МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДВНЗ «УЖГОРОДСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ УПРАВЛЯЮЧИХ СИСТЕМ
ТА ТЕХНОЛОГІЙ

ВСТУП ДО ВЕБТЕХНОЛОГІЙ

Навчальний посібник

Ужгород • РІК-У • 2024

УДК 004.774: 004.43
П 30

Пецко В. І., Беца А. С., Мазютинець Г. В., Міца О. В.

П 30 Вступ до вебтехнологій: навчальний посібник для студентів спеціальності 122 – «Комп'ютерні науки». Ужгород: РІК-У, 2024. 252 с.

ISBN 978-617-8390-10-5

Навчальний посібник «Вступ до вебтехнологій» містить матеріали для ефективного засвоєння сучасних вебтехнологій та суміжних галузей знань. Видання охоплює широке коло тем з основ HTML, CSS та JavaScript. Посібник сприятиме формуванню професійних компетентностей, необхідних для успішної діяльності у галузі розробки вебсайтів та вебзастосунків, а також буде корисним для всіх, хто прагне опанувати сучасні вебтехнології та навчитися створювати функціональні й привабливі вебсайти.

УДК 004.774: 004.43

Розробники:

Пецко В. І., к. т. н., доцент кафедри інформаційних управляючих систем та технологій ДВНЗ «Ужгородський національний університет»;

Беца А. С., аспірант кафедри інформаційних управляючих систем та технологій ДВНЗ «Ужгородський національний університет»;

Мазютинець Г. В., аспірант кафедри інформаційних управляючих систем та технологій ДВНЗ «Ужгородський національний університет»;

Міца О. В., д.т.н., професор, завідувач кафедри інформаційних управляючих систем та технологій ДВНЗ «Ужгородський національний університет».

Рецензенти:

Гече Ф.Е., д.т.н., професор, професор ДВНЗ «Ужгородський національний університет»;

Головач Й.І., д.т.н., професор, професор Закарпатського угорського інституту імені Ференца Ракоці ІІ.

Рекомендовано до друку Вченою радою ДВНЗ "Ужгородський національний університет" (протокол №11 від 18.12.2023 р.)

ISBN 978-617-8390-10-5

© Авторський колектив, 2024
© ТОВ «РІК-У», 2024

ЗМІСТ

ВСТУП 7

1. ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ. ІНТЕРНЕТ, ПЕРЕДАВАННЯ ДАНИХ У МЕРЕЖІ ІНТЕРНЕТ. ВЕБСТОРІНКА ТА ВЕБСАЙТ 9

1.1. Інформаційні технології. Інтернет. Передавання даних у мережі Інтернет 9

1.2. Вебсторінка та вебсайт 11

2. HTML.. 13

2.1. Основи HTML. Базові конструкції HTML 13

2.1.1. Основи HTML. Елементи HTML 13

2.1.2. Структура HTML-документа 14

2.1.3. Задання кольору в HTML-документі 18

2.1.4. Елементи для форматування тексту 19

2.1.5. Блочні елементи 21

2.1.6. Нумеровані та марковані списки 22

2.1.7. Спецсимволи 24

2.1.8. Текстові гіперпосилання 26

2.2. Використання таблиць та iframe на вебсайтах 28

2.2.1. Використання таблиць на вебсайтах 28

2.2.2. Використання iframe на вебсайтах 32

2.3. Графіка, аудіо- та відеоінформація на вебсторінках 34

2.3.1. Використання, розміщення і вирівнювання зображень на вебсторінках 34

2.3.2. Використання мультимедіа на вебсторінках 39

2.4. Форми в HTML 43

2.4.1. Елемент <form> 43

2.4.2. Елемент <input> 46
2.4.3. Елемент <button> 53
2.4.4. Елемент <textarea> 53
2.4.5. Елемент <select> 53
2.4.6. Елемент <label> 54
2.4.7. Елемент <fieldset> 54
3. КАСКАДНІ ТАБЛИЦІ СТИЛІВ 56
3.1. Основи CSS 56
3.2. Синтаксис CSS 59
3.3. Селектори 64
3.4. Спадкування 73
3.5. Каскадування 74
3.6. Форматування тексту та інші важливі CSS-властивості 75
3.7. Адаптивний дизайн 83
3.8. Flexbox 92
4. JAVASCRIPT 110
4.1. Основи синтаксису Javascript 111
4.1.1. Опис змінних 114
4.1.2. Опис констант 116
4.1.3. Ввід та вивід даних. Взаємодія з користувачем 117
4.1.4. Типи даних 119
4.1.5. Перетворення типів 128
4.1.6. Логічні оператори 129
4.1.7. Побітові оператори 130
4.1.8. Умовні оператори *131*
4.2. Цикли 133
4.3. Рядки 136
4.4. Функції. Області видимості функцій 140
4.4.1. Параметри функції, псевдо-масив arguments 141

4.4.2. Передача параметрів функції за значенням та за посиланням 143

4.4.3. Різновиди функцій, які використовуються в JavaScript 144

4.4.4. Глобальний об'єкт, порядок компіляції (ініціалізації), hoisting 147

4.4.5. Область видимості змінних 148

4.4.6. Лексичне оточення, вкладені функції, замикання функції 152

4.4.7. Паттерн модуль 156

4.5. Масиви 157

4.6. Об'єкти. Конструктори об'єктів 166

4.6.1. Об'єкти. Властивості та методи об'єктів 166

4.6.2. Копіювання та порівняння об'єктів 169

4.6.3. Конструктори об'єктів 171

4.6.4. Функція як об'єкт. Методи call, apply, bind 173

4.7. Класи 175

4.7.1. Класи 175

4.7.2. Статичні властивості та методи 178

4.7.3. Приватні властивості та методи 179

4.7.4. Властивості та методи доступу 180

4.7.5. Наслідування 181

4.8. Promise та їх використання 182

4.9. Мережеві запити. Fetch 194

5. БРАУЗЕРНЕ СЕРЕДОВИЩЕ 201

5.1. Структура браузерних об'єктів 201

5.2. Дерево DOM 202

5.3. Навігація по DOM-елементах 204

5.4. Основи роботи з подіями 218

6. ЗАВДАННЯ ДЛЯ ЛАБОРАТОРНИХ РОБІТ 226
6.1. *Лабораторна робота №1* 226
6.2. *Лабораторна робота №2* 229
6.3. *Лабораторна робота №3* 235
6.4. *Лабораторна робота №4* 239
6.5. *Лабораторна робота №5* 241
ПЕРЕЛІК ПИТАНЬ ДЛЯ САМОКОНТРОЛЮ 244
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..................................246
ВІДОМОСТІ ПРО АВТОРІВ 248

ВСТУП

Web-технології займають важливе місце в сучасному інформаційному суспільстві. З їх допомогою переважна більшість людей дізнається про важливі події у світі та країні, звичною стала купівля речей в онлайн-магазинах, активно використовуються системи онлайн-навчання (Moodle) та освітні платформи, такі як Coursera, Edx.com тощо.

Основними завданнями вивчення дисципліни «Вступ до вебтехнологій» є формування у студентів вмінь використовувати сучасні інформаційні технології у мережі Інтернет, створювати сайти з використанням набутих знань HTML5, CSS3 та JavaScript.

Метою цього посібника є ознайомлення з основами функціонування глобальної мережі Інтернет, основними підходами, принципами, методами та засобами розробки сайтів за допомогою мови гіпертекстової розмітки HTML5, каскадних таблиць стилів CSS3 та мови програмування JavaScript.

Цей посібник присвячено теоретичним та практичним аспектам вебтехнологій та вебпрограмуванню. У ньому пропонується повний огляд ключових тем у сфері веброзробки – від основ HTML5 та CSS3 до поглибленого вивчення JavaScript.

У книзі пропонуються зрозумілі пояснення, кожен розділ супроводжується численними прикладами вебсторінок і фрагментами коду, а також завдання для формування навичок, що необхідні для створення власних вебсайтів та вебдодатків. Цей посібник не лише надає базові знання з HTML, CSS і JavaScript, а й розглядає їх взаємодію. Значну увагу приділено використанню flex-технології та медіазапитів для створення адаптивних дизайнів вебсторінок, які працюють на будь-яких пристроях з різними розмірами екранів.

Посібник також пропонує поглиблене вивчення не лише основних аспектів мови програмування JavaScript – логічних та умовних операторів, циклів, рядків та масивів, а й детальне пояснення «підводних каменів» різних видів функцій у JavaScript, області видимості змінних та їх використання.

Особлива увага приділяється питанням, які стосуються об'єктно-орієнтованого програмування, а саме детально

розписано, як працюють в JavaScript класи та об'єкти, наведено багато прикладів їх використання. А також описано використання сучасних інструментів, таких як Promise та Fetch, для роботи з мережевими запитами. Досліджується браузерне середовище і події DOM – для формування розуміння, як взаємодіяти з елементами вебсторінки та створювати динамічні застосунки.

Цей посібник – це не просто набір правил і вказівок, а й інструмент, що надихає до творчості та відкриває нові можливості у світі веброзробки. Разом з нами ви зможете відчути себе впевненими в справі і готовими до викликів, які ставлять сучасні технології, і в майбутньому стати справжніми професіоналами у сфері веброзробки.

3. КАСКАДНІ ТАБЛИЦІ СТИЛІВ

3.1. Основи CSS

Стиль – це набір правил оформлення та форматування, який можна застосувати до різних елементів вебдокумента.

Каскадні таблиці стилів (**CSS**, **Cascading Style Sheets**) містять параметри форматування частини або всього тексту вебсторінки. Таблиці каскадних стилів дають змогу визначити єдиний стиль оформлення для різних сторінок документа і швидко модифікувати його зміною відповідного параметра у таблиці стилів.

Параметрів форматування, які можна задавати за допомогою стилів, досить багато. Це, зокрема:

background – колір тла;
font-family – стиль шрифту (гарнітура);
font-size – розмір шрифту;
font-weight – жирність шрифту;
color – колір шрифту;
text-decoration – оздоблення тексту;
text-align – вирівнювання тексту;
margin-top – відступ від верхнього рядка абзацу;
line-height – міжрядкова відстань.

Є три способи зв’язку каскадних стилів із HTML-документом:

1. Підключення зовнішньої таблиці стилів (пов’язані стилі).
2. Розташування опису стилів у розділі **<head>** документа (глобальні стилі).
3. Задання властивостей стилів безпосередньо в тегах абзаців чи заголовків (вбудовані стилі).

1. Створення та підключення зовнішньої таблиці стилів

Зовнішня таблиця стилів (**External Style Sheet**) – це текстовий файл із розширенням .css. Його підключають до HTML-документа за допомогою тегу **<link>,** який записують у розділі **<head>**, наприклад:

```
<link rel="stylesheet" type="text/css" href="адреса файлу">
```

Значення атрибутів **rel** і **type** залишаються незмінними незалежно від коду, як наведено в цьому прикладі. Значення **href** задає шлях до CSS-файлу, він може бути заданий як відносно, так і абсолютно [12]. Зауважимо, що таким чином можна підключати таблицю стилів, яка знаходиться на іншому сайті. Файл зі стилем не зберігає ніяких даних, крім синтаксису CSS. Підключення зовнішньої таблиці стилів:

```
<link rel="stylesheet" href="mysite.css">
```

Файл зі стилем

```
h3 {
  color: #000080;
  font-size: 200%;
  font-family: Arial, Verdana, sans-serif;
  text-align: center;
}
p {
  padding-left: 20px;
}
```

2. Розташування опису стилів у розділі HEAD документа у блоці, який обмежений тегами <style> та </style>

```
<head>
  <style>
    Тег1 {властивість 11 : значення 11; властивість 12 :
    значення 12; ...; властивість 1n : значення  1n }
    Тег2 {властивість 21 : значення 21; властивість 22 :
    значення 22; ...; властивість 2m : значення 2m}
  </style>
</head>
```

Найпростіша внутрішня таблиця стилів – це послідовність значень тегів, кожне з яких записується, як правило, з нового рядка. Визначення тегу містить його ім’я без кутових дужок, за яким у фігурних дужках через крапку з комою перелічують властивості тегів та їхні значення, розділені двокрапками.

3. **Вбудовані (внутрішні) стилі (Inline Styles) вставляють в теги** (теги заголовків **<h1>...<h6>**, абзацу **<p>**, тіла **<body>**, а також у теги **<div>**, **<img>** тощо) за допомогою атрибута значення. Наприклад:

```
<p style="font-size: 48pt; color: yellow">
```

Визначені у такий спосіб властивості мають найвищий пріоритет порівняно з іншими, оскільки вони визначені безпосередньо у тегу.

Цей підхід використовують для оформлення невеликої кількості елементів.

```
<h2 style="font-size: 48pt; font-family: Arial">Текст...</h2>
```

Приклад:

```
<!DOCTYPE html>
<html>
 <head>
  <link rel="stylesheet" href="mysite.css">
  <style>
   h1 {font-size: 38pt; color: red}
   h2 {font-size: 20pt; color: blue}
  </style>
  <title>Приклад використання CSS</title>
 </head>
 <body>
  <h1>Для заголовка першого рівня визначено розмір 38 pt, а
  колір тексту – червоний </h1>
  <h2>Для заголовка другого рівня визначено розмір 20 pt, а
  колір тексту – синій </h2>
  <h3>Для заголовка 3 рівня визначено з файла mysite.css </h3>
  <h4>Для заголовка 4 рівня визначено з файла mysite.css </h4>
  <h2 style="font-size: 48pt; font-family: Arial; color:
  yellow"> Текст вбудованого стилю (внутрішнього)</h2>
  <p>
    Для цього абзацу стиль не застосовано, для оформлення
    тексту використано атрибути за умовчанням.
  </p>
 </body>
</html>
```

Імпорт CSS.

У поточну стильову таблицю можна імпортувати вміст CSS-файлу за допомогою команди **@import**. Цей метод допускається використовувати спільно з пов'язаними або глобальними стилями (зовнішні таблиці стилів), але ніяк не з внутрішніми стилями. Загальний синтаксис такий:

import url («ім’я файлу») типи носіїв;

import «ім’я файлу» типи носіїв;

Після ключового слова **@import** вказується шлях до стильового файлу одним з двох наведених способів – за допомогою **url** або без нього.

```
@import url(«style/header.css»);
@import «/style/main.css» screen; /* Стиль для виведення результату на монітор */
@import «/style/smart.css» print, handheld; /* Стиль для друку і смартфона */
```

Типи носіїв.

У CSS для вказівки типу носіїв застосовуються команди **@import** і **@media**, за допомогою яких можна визначити стиль для

елементів залежно від того, виводиться документ на екран або на принтер.

- **all** – всі типи. Це значення використовується за умовчанням;
- **aural –** мовні синтезатори, а також програми для відтворення тексту вголос. Сюди, наприклад, можна віднести мовні браузери;
- **braille** – пристрої, засновані на системі Брайля, які призначені для сліпих людей;
- **handheld** – КПК та аналогічні їм апарати;
- **print** – пристрої для друку – на кшталт принтера;
- **projection –** проєктор;
- **screen** – екран монітора;
- **tv** – телевізор.

Команда **@media** дозволяє вказати тип носія для глобальних або пов'язаних стилів і в загальному випадку має такий синтаксис:

```
@media тип носія 1 {
  Опис стилю для типу носія 1
}
@media тип носія 2 {
  Опис стилю для типу носія 2
}
```

3.2. Синтаксис CSS

Таблиця стилів складається з набору правил. Кожне правило, у свою чергу, складається з одного або декількох селекторів і блоку визначень, що відносяться до них. Загальний спосіб запису:

```
Селектор{
    властивість1: значення;
    властивість2: значення;
    …;
    властивість N: значення;
}
```

Селектор – це певне ім'я стилю, для якого додаються параметри форматування. В якості селектора виступають теги, класи та ідентифікатори.

Спочатку пишеться ім’я селектора, наприклад, **table**, це означає, що всі стильові параметри будуть застосовуватися до тегу **<table>**, потім йдуть фігурні дужки, в яких записується визначення властивостей. Визначення властивостей складається із стильової властивості і її значення, що вказується після двокрапки. Визначення властивостей розділяються між собою крапкою з комою, в кінці цей символ можна опустити.

CSS не чутливий до регістру, перенесення рядків, пробілів і символів табуляції, тому форма запису залежить від бажання розробника.

Приклад:

Нижче описані ідентичні стилі.

```
h1 { color: #a6780a; font-weight: normal; border-bottom:2px
solid black}
h1 {
    border-bottom: 2px solid black
    color: #a6780a;
    font-weight: normal;
}
```

Для селектора допускається додавати кожне правило окремо:

```
td {background: olive;}
td {color: white;}
td {border: 1px solid black;}
```

Компактна форма запису:

```
td {
   background: olive;
   color: white;
   border: 1px solid black;
}
```

Якщо для селектора спочатку задається властивість з одним значенням, а потім та ж властивість, але вже з іншим значенням, то застосовуватися буде те значення, яке в коді встановлено нижче

```
p {color: green;}
p {color: red;}
```

У цьому прикладі для селектора **p** колір тексту спочатку задається зеленим, а потім червоним. Оскільки значення **red** розташоване нижче, то воно в підсумку і буде застосовуватися до тексту.

Щоб позначити, що текст є коментарем, застосовують таку конструкцію **/ * …коментар... * /**

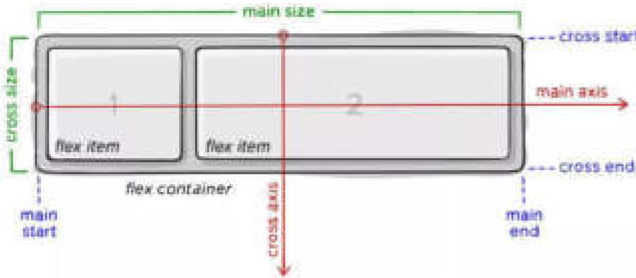
Приклад:

```
/ * Мій стиль */
div {
```

Основи і термінологія

Оскільки **flexbox** – це цілий модуль, а не одна властивість, він включає в себе безліч елементів з набором властивостей. Деякі з них призначені для установки в контейнері (батьківський елемент прийнято називати «flex контейнер»), в той час як інші призначені для установки в дочірніх елементах (так звані «flex елементи»).

Якщо «звичайне» компонування засноване як на блочному, так і на **inline** напрямках, то **flex layout** засновано на «напрямках **flex-flow**». На малюнку внизу відображено основну ідею гнучкого макету.



Елементи будуть розташовані або в напрямку головної осі (**main axis** від **main-start** до **main-end**), або в напрямку поперечної осі (**cross axis** від **cross-start** до cross-end).

- **main axis** – головна вісь **flex**-контейнера, уздовж якої розташовуються **flex**-елементи. Будьте уважні, ця вісь не обов'язкова горизонтальна; це залежить від властивості **flex-direction** (див. нижче).
- **main-start | main-end** – **flex**-елементи поміщаються в контейнер, починаючи з **main-start** і закінчуючи **main-end**.
- **main size** – ширина або висота **flex** елемента, залежно від того, що знаходиться в основному вимірі. Визначається основним розміром **flex**-елементів, тобто властивістю **'width'** або **'height'**, залежно від того, що знаходиться в основному вимірі.
- **cross axis** – вісь, перпендикулярна головній осі, називається поперечною віссю. Її напрямок залежить від напрямку головної осі.
- **cross-start | cross-end** – **flex**-рядки заповнюються елементами і поміщаються в контейнер, починаючи від **cross-start flex** контейнера у напрямку до **cross-end**.

Дивіться на властивість **align-items**, щоб зрозуміти доступні значення.

```
.item {
  align-self: auto | flex-start | flex-end | center | baseline
| stretch;
}
```

Зверніть увагу, що властивості **float**, **clear** і **vertical-align** не впливають на **flex** елементи.

Моменти, які потрібно пам'ятати:

1. Не слід використовувати **flexbox** там, де в цьому немає потреби.
2. Визначення регіонів і зміни порядку контенту у багатьох випадках корисно робити залежними від структури сторінки. Продумайте це.
3. Розберіться у **flexbox** та знайте його основи. Так набагато легше досягти очікуваного результату.
4. Не забувайте про **margin**-и. Вони враховуються при встановленні вирівнювання по осях. Також важливо пам'ятати, що **margin**-и у **flexbox** не "колапсяться", як це відбувається у звичайному потоці.
5. Значення **float** у **flex**-блоків не враховується та не має значення.

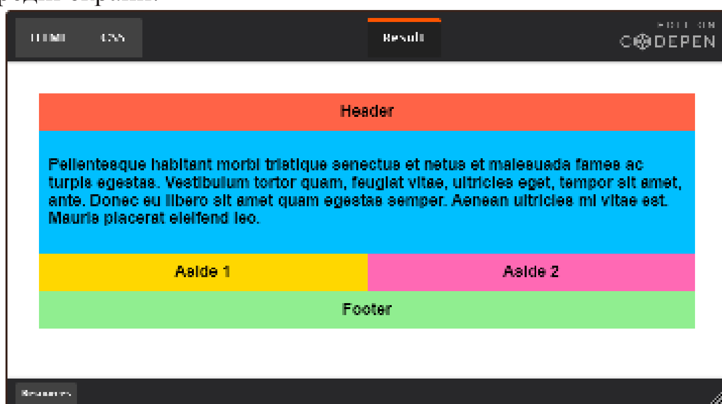
Приклади використання flexbox

Почнімо з дуже простого прикладу, розв'язання майже щоденної проблеми: ідеальне центрування. Найпростіше рішення для цього завдання – це використовувати **flexbox**.

```
<!-- HTML-код -->
<div class="parent">
  <div class = "child">text</div>
</div>
/* CSS-код */
.parent {
  display: flex;
  height: 300px; /* Або що завгодно */
}

.child {
  width: 100px;  /* Або що завгодно */
  height: 100px; /* Або що завгодно */
  margin: auto;  /* Вийшло! */
}
```

Середні екрани:



Маленькі екрани:

```
        return {
            display: function(){
                console.log(obj.greeting);
            }
        }
    })();
    foo.display();  // hello
```

Тут визначено змінну **foo**, яка представляє результат анонімної функції. Всередині цієї функції визначено об'єкт **obj** із певними даними.

Анонімна функція повертає об'єкт, який визначає функцію **display**. Об'єкт, що повертається, визначає загальнодоступний **API**, через який ми можемо звертатися до даних, визначених усередині модуля. Така конструкція дозволяє закрити певний набір даних у рамках функції-модуля та опосередковувати доступ до них через певний **API** – внутрішні функції, що повертаються.

4.5. Масиви

На відміну від деяких мов програмування, масиви в **JavaScript** є складними, впорядкованими неоднорідними типами даних. Тобто в масиві можна зберігати елементи різних типів. Кожен елемент характеризується своїм індексом – унікальним номером.

Синтаксис для створення нового масиву – квадратні дужки зі списком елементів у середині.

```
    var arr = []; //Порожній масив
    var fruits = ["Яблуко", "Апельсин", "Слива"]; // Масив фруктів
з трьома елементами
    var arr = [1, 'Ім\'я', [1,2,3], true];  // різнотипні елементи
```

Враховуючи, що масив є об'єктом, його можна створити за допомогою конструктора **new Array()**. Якщо як параметр передати одне ціле число, то буде створено порожній масив відповідної довжини, в решті випадків буде створено масив, що складається з аргументів конструктора. Виклик конструктора із від’ємним аргументом призводить до помилки.

```
    var arr = new Array(2,3); // еквівалентно arr =  [2, 3]
    arr = new Array(3); // arr = [,,]
    arr = new Array(-3); // Помилка
```

Через **alert** можна вивести масив цілком. При цьому значення його елементів будуть перераховані через кому:

```
    alert(fruits); // Яблуко, Апельсин, Слива
```

Загальне число елементів, збережених в масиві, міститься в його властивості **length**:

```
alert(fruits.length); //3
```

Зазвичай нам не потрібно самостійно міняти **length**. При зменшенні **length** масив вкорочується. Причому цей процес незворотний, тобто якщо потім повернути **length** назад – значення не відновлюються. Найпростіший спосіб очистити масив – присвоїти властивості **length** нульове значення.

Елементи нумеруються, починаючи з нуля. Щоб отримати потрібний елемент з масиву, вказується його індекс у квадратних дужках:

```
alert(fruits [0]); // Яблуко
alert(fruits [2]); // Слива
```

Елемент можна завжди замінити:

```
fruits [2] = 'Груша'; // тепер ["Яблуко", "Апельсин", "Груша"]
```

Або додати:

```
fruits [3] = 'Лимон'; // тепер ["Яблуко", "Апельсин", "Груша", "Лимон"]
```

Слід врахувати, що додавання елемента в позицію, яка перевищує поточну розмірність масиву, призводить до збільшення розмірності. При цьому елементи, значення яких не вказано явно, не займають місце в пам'яті. При виводі масиву через **alert** на їх позиції нічого не відобразиться, але відповідні коми виводяться. При звертанні безпосередньо до них значення вважається рівне **undefined**.

```
var arr[1,2,3];
arr[5] = 5; // в результаті отримаємо масив з 6 елементів [1,2,3,,,5]
alert(a[4]); // undefined
alert(a.length); //6
```

Можна присвоювати цілі масиви, але необхідно врахувати, що при цьому відбувається копіювання вказівника на масив. Це може викликати непередбачувані наслідки, тому доводиться копіювати масиви поелементно:

```
var a = [1,2,3], b = a;
b[1] = 'Ой!';
alert(a[1]); // Ой!, а не як очікувалось 2
var a = [1,2,3], b = [];
for (var i = 0; i < a.length; i++) { b[i] = a[i] };
b[1] = 'Ой!';
alert(a[1]); // 2 як і має бути
```

Приклад. Знайти суму *n* чисел.

```
var n = +prompt('n=', '');
```

```
var a = [];
for (var i = 0; i < n; i++){
     a[i] = +prompt('Введіть число:','');
};
var s = 0;     //шукаємо суму
for (i = 0; i < n; i++){
   document.writeln(a[i]);
   s = s + a[i];
};
document.writeln(' s= ', s );
```

Приклад. Знайти максимальне число.

```
var n =+ prompt('n=', '');
var a = [];
for (var i = 0; i < n; i++){
   a[i]=+prompt('Введіть число:','');
   document.writeln(a[i]);
};
var max=a[0];
for (i=0; i<n; i++){
   if (a[i] >= max) {max = a[i];};
};
document.writeln(' max= ', max );
```

Приклад. Відсортувати масив по спаданню.

3 8 2 4 6

8 6 4 3 2

```
var n =+ prompt('n=', '');
var a=[];
for (var i = 0; i < n; i++){
   a[i] =+ prompt('Введіть число:','');
   document.writeln(a[i]);
};
var max = a[0];
for (i = 0; i < n; i++){
  for (j = i+1; j < n; j++){
     if (a[j] > a[i]) { b = a[i]; a[i] = a[j]; a[j] = b;};
  }
};
document.writeln('<br>');
for (var i=0; i<n; i++){
   document.writeln(a[i]);
};
```

Приклад. Створити скрипт, який серед введених прізвищ знаходить однофамільців.

```
var n =+ prompt('n=','');
var a = [];
for (var i = 0; i < n; i++){
   a[i] = prompt('Введіть прізвище:','');
};
var o = [];
next: for (i = 0; i < n; i++){
   for (var j = i+1; j < n; j++){
```

```
            if (a[i] == a[j] ) {       // знайшли однофамільця
                  for (var l = 0; l < o.length; l++){
                        if(a[i] == o[l]) continue next;  //він вже
є в списку однофамільців
              }; //l
              o[o.length] = a[i];  //додаємо до списку однофамільців
              continue next;
            }; // if
      };  // j
   }; // i
   if (o.length){alert(o)} else {alert('однофамільців немає')};
```

Приклад. Знайти всі прості числа до 1000 та їх суму за допомогою решета Ератосфена.

```
   var c = [];
   const n = 1000;
   for(var i = 1; i < n; i++){c[i] = true}; //вважаємо всі числа
простими
   var p = 2;
   while (p <= Math.sqrt(n)){
      for (i = 2*p; i < n; i = i + p){  //всі числа кратні p вважаємо
не простими
            c[i]=false;
      };
      i = p + 1;  //шукаємо наступне просте число
      while (!c[i]){
            i++;
      };
      p = i;
   };
   var s = 0; //шукаємо суму
   for (i = 1; i < n; i++){
      if (c[i]){
            document.writeln(i);
            s = s + i;
      };
   };
   alert('Сума дорівнює'+s);
```

Масиви в **JavaScript** можуть містити як елементи інші масиви. Цим можна скористатись для створення багатовимірних масивів, наприклад матриць. Звертатись до елемента необхідно, беручи кожен індекс в квадратні дужки.

```
   var matrix = [
       [1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]
   ];
   alert(matrix [1][2]); // елемент в другому рядку, третьому
стовпці рівний 6
```

Якщо матрицю неможливо задати явно, то щоб інтерпретатор зрозумів, з чим має справу, перед її подальшою обробкою

необхідно провести ініціалізацію: спочатку описати матрицю як масив, а потім кожному елементу присвоїти порожній масив.

```
var matrix = [];    // матриця розміром n*m
for(i = 0; i < n; i++){
  matrix[i] = new Array(m); // або matrix[i] = [];
};
```

Приклад. Заповнити матрицю послідовністю натуральних чисел по спіралі (починаючи з верхнього лівого кута, за годинниковою стрілкою). Вивести результат у вигляді таблиці.

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

```
var n = +prompt('Розмірність',''), i, j, k, e=1;
var m = [];        //ініціліазація матриці
for(i = 0; i < n; i++){
  m[i]=[];
};
for ( k = 0;k <=(n -1)/2;k++){
  for (j = k; j < n - k; j++){m[k][j] = (e++)};
  //перший рядок
  for (i = k+1; i < n - k; i++){m[i][n - k - 1] = (e++)};
  //останній стовпець
  for (j = n - k - 2; j >= k; j--){m[n-k-1][j] = (e++)};
  //останній рядок
  for (i = n - k - 2; i > k; i--){m[i][k] = (e++)};
  //перший стовпець
};
//вивід таблиці за допомогою тегів
document.writeln('<table border="2" cellspacing="0">');
for (i = 0; i < n; i++){
  document.write('<tr>');
  for (j = 0; j < n; j++){
    document.write('<td>' + m[i][j] + '</td>');
  };
  document.writeln('</tr>');
};
document.writeln('</table>');
```

Масиви є об’єктами і мають численні методи для полегшення роботи з ними.

split(<u>роздільник</u>) дозволяє перетворити рядок в масив, розбивши його по роздільнику.

```
var names = 'Петя, Марина, Василь';
var arr = names.split (',');    // arr = ['Петя', 'Марина', 'Василь']
for (var i = 0; i < arr.length; i ++) {
  alert(arr [i]);
}
```

Виклик **str.split ('')** розіб'є рядок на літери.

join (<u>роздільник</u>) Протилежний метод до split. Він склеює масив в рядок, використовуючи роздільник.

```
var arr = ['Петя', 'Марина', 'Василь'];
alert(arr.join(';')); //Петя;Марина;Василь
```

splice(позиція, [<u>лічильник видалення</u>, елемент1, елемент2, …, елементN]) – універсальний метод для роботи з масивами. Вміє все: видаляти елементи, вставляти елементи, замінювати елементи – по черзі й одночасно. Він видаляє вказану кількість елементів, починаючи з позиції, а потім вставляє елементи на їх місце. Наступні за видаленими елементами зсуваються, щоб заповнити їх місце. Допускається використання від'ємного номера позиції, яка в цьому випадку відраховується з кінця.

```
var a= [0,1,2,3,4,5,6,7];
a.splice(1, 1); // видалити 1 елемент, починаючи з позиції 1, a=[0,2,3,4,5,6,7];
a.splice(-3,2); //видалити 2 елементи, починаючи з 3 від кінця позиції, a=[0,2,3,4,7]
a.splice(0, 3, 5, 5); // видалити 3 елементи на початку і додати дві 5, a=[5,5,4,7]
a.splice(3, 0, 1); // додати без видалення a=[5,5,4,1,7]
var b=a.splice(1, 2,’видалено 5,4’); //a=[5,’видалено 5,4’,1,7]; b=[5,4];
```

slice(begin, end) копіює частину масиву від **begin** до **end**, не включаючи **end**. Вихідний масив при цьому не змінюється.

```
var arr = ["Чому", "треба", "вчити", "JavaScript"];
var a = arr.slice(1,3); // елементи 1, 2 (не включаючи 3) a=["треба", "вчити"];
```

reverse() змінює порядок елементів у масиві на зворотний.

indexOf(searchElement [, fromIndex]) повертає номер елемента **<u>searchElement</u>** в масиві, або **-1**, якщо його немає. Пошук починається з номера **fromIndex**, якщо він зазначений. Якщо ні – від початку масиву. Для пошуку використовується суворе порівняння ===.

```
var arr = [1, 0, false,1];
```

```
alert(arr.indexOf(0)); // 1
alert(arr.indexOf(false)); // 2
alert(arr.indexOf(null)); // -1
alert(arr.indexOf(1,2)); //3
```

lastIndexOf(searchElement [, fromIndex]) має аналогічні параметри, але шукає останнє входження елементу, переглядаючи масив справа наліво, з кінця або з номера **fromIndex**, якщо він зазначений.

Метод **sort()** сортує масив. Він сортує в порядку зростання, перетворюючи елементи до рядкового типу. Це дає неправильні результати для числових масивів.

```
var arr = [1, 2, 15, 36, 305];
arr.sort();
alert(arr); // 1, 15, 2, 305, 36
```

Метод **sort (функція порівняння)** вміє сортувати будь-які масиви, якщо вказати функцію **fn** від двох елементів, яка вміє порівнювати їх. Вона повинна повертати:

- додатне значення, якщо перший елемент більший;
- від'ємне значення, якщо другий більший;
- якщо рівні, то 0.

```
function compareNumber (a, b) {
    if (a>b) { return 1 };
    if (a<b) { return -1 };
    return 0;
    //а можна просто return a-b
}
var arr = [1, 2, 15, 36, 305];
arr.sort(compareNumber);
alert(arr); // 1, 2, 15, 36, 305
```

Приклад. Скласти скрипт для сортування масиву **arr** порядку спадання.

```
arr.sort(function(a,b){return b-a});
```

Приклад. Скласти скрипт для сортування масиву **arr** в порядку, коли спочатку йдуть всі парні числа, потім – непарні.

```
function evenCompare (a, b) {
    // Числа однієї парності - сортуються звичайним чином
    if (a%2 == b%2) return a - b;
    // Інакше, якщо a - парне, то воно менше
    if (a% 2 == 0) return -1;
    // Лишився один варіант: a - непарне, і b – парне
    return 1;
}
var arr = [5, 2, 1, -10, 8];
arr.sort(evenCompare);
alert(arr); // -10, 2, 8, 1, 5
```

forEach(<u>функція</u>) викликає функцію для кожного елемента масиву. Функція викликається з параметрами **(item, i, arr)**: **item** – значення поточного елемента масиву; **i** – його номер; **arr** – масив, що перебирається. Зміна значення **item** не приводить до зміни відповідного елементу, для цього слід звертатись до **arr[i]**.

<u>Приклад.</u> Скласти скрипт, що виводить кожен елемент масиву в окремий рядок у форматі a[i]=…

```
function print(x,i,arr){
  document.writeln('a[' + i + ']=' + x + '<br>');
}
var a = [2, 3, 5];
a.forEach(print);
alert(a);
```

filter (callback) створює новий масив, в який увійдуть тільки ті елементи вихідного масиву, для яких виклик **callback (item, i, arr)** поверне **true**.

<u>Приклад.</u> Скласти скрипт, що знаходить всі парні елементи на непарних місцях.

```
function condition(x,i,arr){
  return (i%2 != 0) && (x%2 == 0);
}
var b = a.filter(condition);
```

map(callback) створює новий масив, який складатиметься із результатів виклику **callback (item, i, arr)** для кожного елемента **arr**.

<u>Приклад.</u> Скласти скрипт, що збільшує кожен елемент масиву на його індекс;

```
function change(x, i, arr){
  return x + i;
}
alert(a.map(change));
```

every(callback) повертає **true**, якщо виклик **callback** поверне **true** для кожного елемента масиву.

some(callback) повертає **true**, якщо виклик **callback** поверне **true** для якогось елемента масиву.

<u>Приклад.</u> Чи всі елементи масиву додатні, чи є хоча б один додатній.

```
var arr = [1, -1, 2, -2, 3];
function isPositive (number) {return number > 0};
alert(arr.every(isPositive)); // false, не всі додатні
alert(arr.some(isPositive)); // true, є хоча б один додатній
```

reduce(reduceCallback [, initialValue]) застосовує функцію **reduceCallback** по черзі до кожного елементу масиву зліва направо, зберігаючи при цьому проміжний результат.

Аргументи функції **reduceCallback(previousValue, currentItem, index, arr):**

previousValue – останній результат виклику функції, він же проміжний результат;

currentItem – поточний елемент масиву, елементи перебираються по черзі зліва направо;

index – номер поточного елемента;

arr – масив.

Значення **previousValue** при першому виклику одно – **initialValue**. Якщо **initialValue** немає, то воно дорівнює першому елементу масиву, а перебір починається з другого.

Приклад. Скласти скрипт для знаходження суми всіх елементів та кількості додатних елементів у масиві.

```
function kilk(result, x, i, arr){
  if (x>0) { result++ };
  return result;
}
function suma(result, x, i, arr){
  result += x;
  return result;
}
var a=[1, 2, -3, 4, -5];
alert(a.reduce(kilk,0));
alert(a.reduce(suma));
```

Стек і черга

Також в JavaScript реалізовані методи для роботи зі стеком і чергою.

push(<u>елемент</u>) – додає елемент у кінець.

pop() – видаляє і повертає останній елемент.

unshift(<u>елемент</u>) – додає елемент у початок.

shift() – видаляє і повертає перший елемент.

Приклад. Скласти скрипт для перевірки балансу дужок. Алгоритм для перевірки балансу: переглядаємо вираз посимвольно. Якщо трапляється відкриваюча дужка, додаємо її до стеку, якщо трапляється закриваюча – вибираємо зі стеку. Якщо стек порожній і з нього неможливо вибрати – балансу немає. Після перегляду всього виразу стек повинен бути порожнім, інакше – в нас багато відкриваючих дужок.

```
var x = prompt('Вираз', ''), stack = [], ok = true;
x=x.split('') ;
for (var i = 0; i < x.length; i++){
  if (x[i] == '('){ stack.push(x[i])};
```

```
    if (x[i] == ')'){
          if (stack.pop() != '(') {
              ok = false;
              break;
          }
    }
  }// i
  if ((stack.length==0) && ok) { alert('Вірно') } else
{alert('помилка')};
```

Приклад. Дано рядок. Скласти скрипт, який би виводив усі цифри, що в рядку, а потім – решту символів, зберігши їх взаємне розташування. Для зберігання цифр скористатись чергою.

```
  var x=prompt('Вираз',''), msg = '', cifra = [];
  x = x.split('') ;
  for (var i = 0; i < x.length; i++){
    if ((x[i] >= '0') && (x[i] <= '9')){
      cifra.push(x[i])
    } else {
      msg = msg+x[i]
    };
  } //i
  msg = cifra.join('') + msg;
  alert(msg);
```

4.6. Об'єкти. Конструктори об'єктів

4.6.1. Об'єкти. Властивості та методи об'єктів

Асоціативний масив – структура даних, у якій можна зберігати будь-які дані парами ключ-значення. Інша назва для асоціативного масиву – «словник» або «хеш». Кожному ключу відповідає єдине значення, яке можна швидко прочитати, записати або видалити.

В JS асоціативний масив реалізується через об'єкт і є його частковим випадком. Об'єкт може містити будь-які значення, які називаються властивостями об'єкта, і засоби для їх обробки – методи. Об'єкт, який містить тільки властивості, і є асоціативним масивом.

Порожній асоціативний масив (порожній об'єкт) може бути створено за допомогою одного з двох синтаксисів:

1) за допомогою конструктора **Object**:
```
let user = new Object();
```
2) за допомогою літерала (використання фігурних дужок):
```
let user = {};
```

```
    }
};
console.log(tom?.name); // undefined
console.log(bob?.name); // Bob
tom?.sayHi();           // не виконається
bob?.sayHi();           // Hi! I am Bob
```

У цьому випадку звернення до властивості **name** та методу **sayHi()** відбувається лише у тому разі, якщо об'єкти **tom** і **bob** не дорівнюють **null/undefined**. Більше того, далі по ланцюжку викликів перевіряти наявність властивості або методу в об'єкті:

```
obj.val?.prop       // перевірка властивості
obj.arr?.[index]    // перевірка масиву
obj.func?.(args)    // перевірка функції
```

4.7. Класи

4.7.1. Класи

Із впровадженням стандарту ES2015 (ES6) у JavaScript з’явився новий спосіб визначення об’єктів – за допомогою класів. Клас представляє опис об’єкта, його стану та поведінки, а об’єкт є конкретним втіленням або екземпляром класу. Для визначення класу використовується ключове слово **class**:

class Person{}

Після слова **class** іде назва класу (в цьому випадку клас називається **Person**), а потім у фігурних дужках визначається тіло класу. Також можна визначити анонімний або неанонімний клас і присвоїти його змінній або константі:

```
const Person = class{}
const User = class Person{}
```

Після визначення класу ми можемо створити об’єкти класу за допомогою конструктора:

```
class Person{}
const tom = new Person();
const bob = new Person();
```

Для створення об’єкта за допомогою конструктора спочатку встановлюється ключове слово **new** . Потім, власне, йде виклик конструктора, – по суті, виклик функції на ім'я класу. За замовчуванням, класи мають один конструктор без параметрів. Тому в цьому випадку при виклику конструктора до нього не передається жодних аргументів.

Оскільки конструктор класу **Person** має два параметри, відповідно до нього передаються два значення. При цьому конструктор базового класу повинен викликатись до звернення до властивостей поточного об'єкта через **this.**

Перший рядок методу **print()** у класі **Employee** містить виклик реалізації методу **print()** базового класу **Person**:

```
super.print();
```

При наслідуванні варто враховувати, що похідний клас може звертатися до будь-якої функціональності базового класу, крім приватних полів та методів.

4.8. Promise та їх використання

Вступ до промісів

При стандартному виконанні **JavaScript** інструкції виконуються послідовно, одна за одною. Тобто спочатку виконується перша інструкція, потім друга і так далі. Однак буває так, що одна з цих операцій виконується тривалий час, – наприклад, вона виконує якусь високонавантажену роботу типу звернення по мережі або звернення до бази даних. У результаті при послідовному виконанні всі наступні операції чекатимуть виконання цієї операції. Щоб уникнути подібної ситуації, **JavaScript** надає ряд інструментів, щоб подальші операції могли виконуватися, поки виконується тривала операція. Одним із таких інструментів є проміси (**promise**).

Проміс (promise) – це об'єкт, що становить результат успішного чи невдалого завершення асинхронної операції. Асинхронна операція, спрощено кажучи, – це певна дія, яка виконується незалежно від навколишнього коду, у якому вона викликається, а також не блокує виконання коду.

Проміс може бути в одному з таких станів:

- **pending** (стан очікування): початковий стан, проміс створено, але виконання ще не завершено;
- **fulfilled** (успішно завершено): дія, яка представляє проміс, успішно завершена;
- **rejected** (завершено з помилкою): при виконанні дії, що представляє проміс, сталася помилка.

Отримання результату операції в Promise

Раніше ми розглянули, як з функції промісу ми можемо передати результат асинхронної операції у зовнішню область програми:

```
const myPromise = new Promise(function(resolve){
    console.log("Виконання асинхронної операції ");
    resolve("Привіт світ!");
});
```

Тепер отримаємо це значення. Для отримання результату операції промісу застосовується функція **then()** об'єкта **Promise**:

```
then(onFulfilled, onRejected);
```

Перший параметр функції – **onFulfilled** представляє функцію, яка виконується при успішному завершенні промісу, і як параметр отримує передані **resolve()** дані.

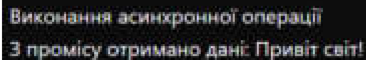
Другий параметр функції – **onRejected** представляє функцію, яка виконується при виникненні помилки, і як параметр отримує передані **reject()** дані.

Функція **then()** повертає об'єкт **Promise**.

Так, отримаємо передані дані:

```
const myPromise = new Promise(function(resolve){
    console.log("Виконання асинхронної операції ");
    resolve("Привіт світ!");
});
myPromise.then(function(value){
    console.log(`Із промісу отримані дані: ${value}`);
})
```

Тобто параметр **value** тут представлятиме рядок "Привіт світ!", який передається в **resolve** ("Привіт світ!"). У результаті консольний вивід виглядатиме так:



При цьому нам необов'язково взагалі передавати в **resolve()** будь-яке значення. Можливо таке, що асинхронна операція просто виконується і не передає жодного результату у зовнішню область.

```
const x = 4;
const y = 8;
const myPromise = new Promise(function(){
    console.log("Виконання асинхронної операції ");
    const z = x + y;
    console.log(`Результат операції: ${z}`)
});
myPromise.then();
```

У цьому випадку функція в промісі обчислює суму чисел ***x*** та

```
sum(25, 4).then(function(value){
  console.log("Результат операції:", value);
});
```

У цьому випадку логіка обробки повторюватиметься. Але оскільки метод **then()** також повертає об'єкт **Promise**, ми можемо зробити так:

```
function sum(x, y){
    return new Promise(function(resolve){
        const result = x + y;
        resolve(result);
    }).then(function(value){
        console.log("Результат операції:", value);
    });
}
sum(3, 5);
sum(25, 4);
```

Гнучке налаштування функції

А якщо ми хочемо, щоб у програміста був вибір: якщо він хоче, то може визначити свій обробник, а якщо ні, то застосовується деякий обробник за замовчуванням? У цьому випадку ми можемо визначити функцію обробника як параметр функції, а якщо його не передано, то встановлювати обробник за замовчуванням:

```
function sum(x, y, func){
    // якщо обробник не встановлений, то встановлюємо обробник
за змовчуванням
    if(func === undefined) func = function(value){
        console.log("Результат операції:", value);
    };
    return new Promise(function(resolve){
        const result = x + y;
        resolve(result);
    }).then(func);
}
sum(3, 5);
sum(25, 4, function(value){
  console.log("Сума:", value);
});
```

При першому виклику функції **sum(3, 5)** буде спрацьовувати обробник за замовчуванням. У другому випадку обробник явно передається через третій параметр, відповідно, він буде задіяний

```
sum(25, 4, function(value){
  console.log("Сума:", value);
})
```

Обробка помилок у Promise

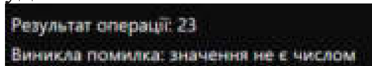
Однією з переваг промісів є простіша обробка помилок. Для отримання та обробки помилки ми можемо використовувати

Другий параметр функції **then()** представляє функцію обробника помилок. За допомогою параметра **error** функції-обробника ми можемо отримати передане в **reject()** значення або інформацію про помилку.

Розглянемо наступний приклад:

```
function generateNumber(str){
    return new Promise(function(resolve, reject){
        const parsed = parseInt(str);
        if (isNaN(parsed))  reject("значення не є числом")
        else resolve(parsed);
    })
    .then(
        function(value){ console.log("Результат операції:",
value);},
        function(error){ console.log("Виникла помилка:",
error);}
    );
}
generateNumber("23");
generateNumber("hello");
```

У цьому випадку для того, щоб у проміс можна було передати різні дані, він визначений як результат функції, що повертається **generateNumber()**. Тобто в цьому випадку консольний вивід буде таким:


Результат операції: 23
Виникла помилка: значення не є числом

Async та await

Впровадження стандарту **ES2017** у **JavaScript** привнесло два нових оператори: **async** і **await** , які покликані спростити роботу з промісами.

Оператор **async** визначає асинхронну функцію, у якій, як передбачається, виконуватиметься одне або кілька асинхронних завдань:

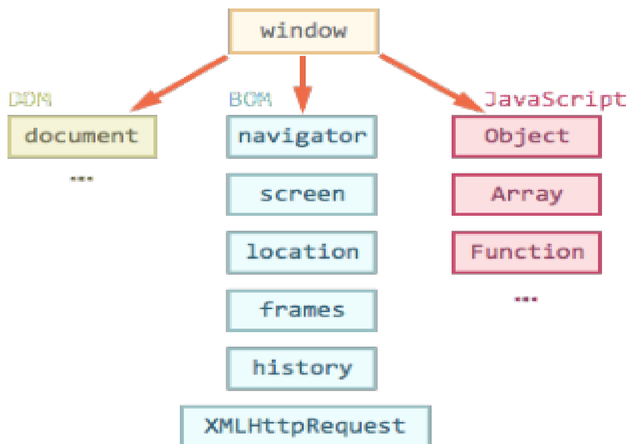
```
async function назва_функції(){
    // асинхронні операції
}
```

Всередині асинхронної функції ми можемо застосувати оператор **await** . Він ставиться перед викликом асинхронної операції, яка представляє об’єкт **Promise:**

```
async function назва_функції(){
    await асинхронна_операція();
```

5. БРАУЗЕРНЕ СЕРЕДОВИЩЕ

5.1. Структура браузерних об'єктів



Є "корінний" об'єкт, що називається **window**. Він має дві ролі:

1. По-перше, це глобальний об'єкт для коду JavaScript у браузері.
2. По-друге, він є "вікном браузера" та надає способи для керування ним.

Приклад:

```
// Відкрити нове вікно/вкладку з URL https://www.uzhnu.edu.ua/
window.open('https://www.uzhnu.edu.ua/');
```

Document Object Model, або скорочено **DOM**, – представляє весь контент сторінки як об'єкти, які можуть бути змінені.

Об'єкт **document** – це головна "точка входу" до сторінки. Ми можемо змінити або створити що-небудь на сторінці, використовуючи цей об'єкт.

Приклад:

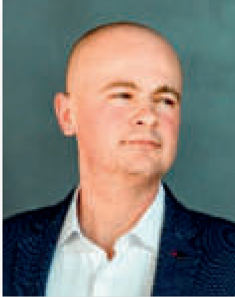
```
// змінити колір фону на червоний
document.body.style.background = "red";
// повернути його назад після 1 секунди
setTimeout(() => document.body.style.background = "", 1000);
```

ПЕРЕЛІК ПИТАНЬ ДЛЯ САМОКОНТРОЛЮ

1. Що таке інформаційні технології?
2. Як здійснюється передача даних в мережі інтернет?
3. Що таке вебсторінка та вебсайт?
4. Які елементи використовують для форматування тексту?
5. Які відмінності між блочними та рядковими елементами?
6. Що таке текстові гіперпосилання та для чого їх використовують?
7. Як задаються таблиці на вебсторінці?
8. Як задаються іфрейми на вебсторінці?
9. Як створити карту посилань?
10. Як задати мультимедіа на вебсторінці?
11. Для чого використовують форми та які атрибути є у форм?
12. Як і для чого використовують такі теги: **<input>**, **<textarea>**, **<select>**?
13. Що таке каскадні таблиці стилів?
14. Які способи зв’язку каскадних стилів із **HTML**-документом існують?
15. Що таке спадкування?
16. Що таке каскадування?
17. Які існують селектори?
18. Що таке клас та яке його застосування?
19. Що таке ідентифікатор та яке його застосування?
20. Що таке контекстний селектор?
21. Яку пріоритетність мають селектори?
22. Що таке псевдокласи та які є їхні різновиди?
23. Які є **CSS**-властивості для стилізації тексту?
24. Що таке позиціонування елементів у **CSS**?
25. Що означають такі CSS-властивості: **position**, **float**, **border**, **margin**, **padding**?
26. Що таке **Flex**-технологія?
27. Що означають такі **CSS**-властивості: **display**, **flex-direction**, **flex-wrap**, **flex-flow**, **justify-content**, **align-items**, **align-content**?
28. Що означають такі **CSS**-властивості: **order**, **flex-grow**, **flex-shrink**, **flex-basis**, **flex**, **align-self**?
29. Які є способи додавання скриптів на сторінку?

30. Які є типи даних у **JS**?
31. Що означає директива **use strict**?
32. Як перевірити, що змінна рівна **NaN**?
33. Порівняйте ключові слова **var**, **let**, **const**.
34. Як працюють оператори присвоєння / порівняння / рядкові / арифметичні / побітові / логічні тощо?
35. Умовні оператори у **JS** та їх використання.
36. Циклічні конструкції у **JS**.
37. Назвіть методи масивів та рокажіть, для чого вони потрібні.
38. Рядки у **JS** та методи рядків.
39. Яка різниця між декларацією функції (**function declaration**) та функціональним виразом (**function expression**)?
40. Що таке анонімна функція?
41. Що таке замикання (**closure**) і які сценарії його використання?
42. Об'єкти, властивості та методи об'єктів.
43. Як здійснюється копіювання та порівняння об'єктів?
44. Що таке конструктор об'єкта?
45. Як працюють такі методи: **call**, **apply**, **bind**?
46. Поняття класів.
47. Статичні властивості та методи класів.
48. Приватні властивості та методи класів.
49. Властивості та методи доступу класів.
50. Наслідування класів.
51. Поясніть, що таке **Promise** та яке його використання.
52. Мережеві запити. **Fetch**.
53. Що таке **DOM**?
54. Як здійснюється навігація по **DOM**-елементах?
55. Як здійснюється пошук елементів на сторінці?
56. Що означають такі властивості як: **innerHTML**, **outerHTML**, **data**, **textContent**?
57. Як здійснюється додавання та видалення вузлів у **JS**?
58. Різновиди події у **JS**.
59. Які є способом призначити обробник події в **JS**?

ВІДОМОСТІ ПРО АВТОРІВ

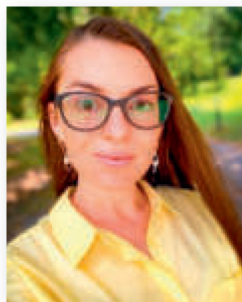


1. **Пецко Василь Іванович** народився 30 грудня 1987 року в с. Керецьки Свалявського району Закарпатської області. У 2003 р. закінчив Керецьківську загальноосвітню школу. З 2003 р. навчався у Свалявському технічному коледжі НУХТ за спеціальністю «Економіка підприємства», який закінчив з відзнакою 2006 року й отримав кваліфікацію економіста з планування. У 2006 р. вступив на математичний факультет Ужгородського національного університету на спеціальність «Математика», який закінчив з відзнакою у 2011 році й отримав кваліфікацію магістра математики, викладача математики та інформатики. З 15.11.2011 по 15.11.2014 навчався в аспірантурі при кафедрі кібернетики та прикладної математики математичного факультету за спеціальністю «Математичне моделювання та обчислювальні методи». З 16.11.2014 – асистент кафедри інформаційних управляючих систем та технологій (ІУСТ) факультету інформаційних технологій Ужгородського національного університету. 29.05.2015 у Тернопільському національному технічному університеті захистив дисертацію «Математичне моделювання оптичних структур та оптимізація їх просторово поляризаційних параметрів при падінні світла під кутом» на здобуття наукового ступеня кандидата технічних наук за спеціальністю 01.05.02. – «Математичне моделювання та обчислювальні методи» (науковий керівник – д.т.н., проф. Міца О. В.). З 01.09.2015 – викладач кафедри ІУСТ, з 01.01.2016 переведений на посаду старшого викладача зазначеної кафедри, а з 01.01.2018 переведений на посаду доцента кафедри ІУСТ. Забезпечує читання курсів «Вступ до вебтехнологій», «Програмування та підтримка вебзастосувань», має багаторічний досвід роботи в ІТ-галузі як Frontend Developer у різних ІТ-компаніях:

SmartGamma (2016 – 2018 рр.), SharpMinds (2018 – 2019 рр.), P-Product (з 2019 р. дотепер).

2. **Беца Анастасія Сергіївна** народилася 18 грудня 1997 року в м. Ужгород Закарпатської області. У 2014 році з відзнакою закінчила Ужгородську загальноосвітню школу І-ІІІ ступенів № 6 ім. В.С. Ґренджі-Донського. У 2015 році була зарахована на навчання на факультет інформаційних технологій ДВНЗ «Ужгородський національний університет» на спеціальність «Інформатика», який закінчила з відзнакою у 2019 році (отримала ступінь бакалавра).

У 2020 році з відзнакою закінчила магістратуру за двома спеціальностями – «Комп'ютерні науки» та «Географія». Того ж року була зарахована в аспірантуру при кафедрі інформаційних управляючих систем та технологій факультету інформаційних технологій Ужгородського національного університету за спеціальністю «Комп'ютерні науки». У 2021 році була прийнята на посаду асистента кафедри ІУСТ факультету інформаційних технологій і дотепер активно займається викладацькою діяльністю в університеті, а також в онлайн-школі програмування для дітей та дорослих з усього світу «TinkerTeens».



3. **Мазютинець Габріела Василівна** народилася 8 травня 1991 року в смт Середнє Ужгородського району Закарпатської області. У 2008 році закінчила Середнянську загальноосвітню школу І-ІІІ ступенів. Того ж року вступила на інженерно-технічний факультет Ужгородського національного університету на спеціальність «Комп'ютерні системи та мережі», який закінчила з відзнакою у 2013 році й здобула кваліфікацію

інженера з комп'ютерних систем. З 2012 по 2014 рік навчалася в Інституті післядипломної освіти та доуніверситетської підготовки УжНУ на спеціальності «Фінанси» й здобула кваліфікацію спеціаліста з фінансів. З 2016 по 2020 рік навчалася в аспірантурі при кафедрі інформаційних управляючих систем та технологій факультету інформаційних технологій УжНУ за спеціальністю «Комп'ютерні науки та інформаційні технології».

Із серпня 2014 року по березень 2015 року працювала у веб-студії «Perspective Studio». З 2015 по 2020 рік працювала в Ужгородському національному університеті на посаді провідного інженера-програміста Центру інформаційних технологій. Також у цей період проводила курси з комп'ютерної грамотності для викладачів університету, була ментором від BrainBasket з вебпрограмування для вчителів інформатики, проводила курси від BrainBasket з мови програмування Scratch для дітей, була ментором з вебпрограмування в IT academy IABS, була ментором в Комп'ютерній Академії IT Step, періодично проводила в школах «Години Коду».

4. **Міца Олександр Володимирович** народився 23 вересня 1977 року в м. Ужгороді Закарпатської області. У 1992 році закінчив Руськівську неповну середню школу. У 1994 році закінчив Ракошинську середню школу і вступив на математичний факультет Ужгородського держуніверситету. Навчаючись на математичному факультеті, двічі (у 1996 та 1998 роках) ставав соросівським стипендіатом. 1999 року закінчив математичний факультет УжДУ з відзнакою та вступив до аспірантури за спеціальністю «Математичне моделювання та обчислювальні методи» (науковий керівник – д.т.н., проф. Головач Й.І.). З 1 листопада 2002 року працював на посаді асистента кафедри кібернетики і прикладної математики УжДУ. 10 червня 2004 року в Тернопільському державному технічному університеті ім. І. Пулюя захистив дисертацію «Математичне

моделювання оптичних шаруватих покриттів та оптимізація їх структури» на здобуття наукового ступеня кандидата технічних наук. У січні 2004 року за значний внесок у розвиток і підтримку обдарованої учнівської молоді нагороджений знаком «Відмінник освіти України». З 1999 року Міца О.В. був членом журі перших трьох етапів Всеукраїнської учнівської олімпіади з інформатики і програмування, а з 2003 року – членом журі всіх етапів Всеукраїнської учнівської олімпіади з інформатики. У 2004 році Міці О.В. присуджено грант Президента України для обдарованої молоді, а у 2006 році – грант для молодих учених України. З 2010 по 2013 рік був головою ради молодих учених Ужгородського національного університету. У 2013 році перейшов працювати на кафедру інформаційних управляючих систем та технологій. З 2014 року – виконувач обов'язків завідувача, а з 2015 року дотепер – завідувач цієї кафедри. 27 квітня 2021 року в Інституті кібернетики ім. В.М. Глушкова НАН України захистив докторську дисертацію на тему «Моделювання та оптимізація спектральних коефіцієнтів шаруватих оптичних систем з неоднорідними границями» (науковий консультант – д.ф.-м.н., с.н.с. Стецюк П.І.). У 2022 році присвоєно звання професора. Міца О. В. опублікував понад 150 наукових праць, зокрема й більше 30 публікацій у періодичних виданнях, які включені до наукометричних баз Scopus або Web of Science.

Міца О.В. є тренером студентських команд з програмування – срібних призерів чемпіонату Південно-Східної Європи (2016, 2017, 2022), переможців Кубку України (2017) та команд, що представляли Україну у фіналі світової студентської першості з програмування й посіли 34 місце в м. Рапід-Сіті (США) 2017 року, 31 місце в м. Пекін (Китай) у 2018 році та команди, яка у 2023 році пройшла у фіналі світової першості з програмування, який пройде в місті Шарм-еш-Шейх (Єгипет).

Забезпечує читання дисциплін «Сучасні технології в програмуванні», «Системний аналіз», «Методика викладання у вищій школі» та «Сучасні методи розв'язання складних оптимізаційних задач».

Навчальне видання

ВСТУП ДО ВЕБТЕХНОЛОГІЙ

Навчальний посібник

Авторський колектив:
Пецко В. І., Беца А. С., Мазютинець Г. В., Міца О. В.

м. Ужгород, вул. Заньковецької, 89А,
ДВНЗ «УжНУ», ФІТ, e-mail: vasyl.petsko@uzhnu.edu.ua

Друкується в авторській редакції

Дизайн обкладинки: Король Р. Я.

Верстка: Кокіна Р. С.

Здано у роботу 20.12.2023 р. Підписано до друку 20.01.2024 р.
Гарнітура Times New Roman. Ум.друк.арк. 14,6. Формат 60x84/16.
Наклад 200 прим. Зам. № 189К.

Оригінал-макет виготовлено та видруковано: ТОВ «РІК-У»
88006, м. Ужгород, вул. Карпатської України, 36, e-mail: print@rik.com.ua
Свідоцтво Серія ДК № 5040 від 21 січня 2016 року