Міністерство освіти і науки України
Житомирський державний університет імені Івана Франка

Матеріали лекційного курсу

# *«Вебтехнології та вебдизайн»*

Конспект лекцій

Житомир 2026

УДК 004.774(075.8)
В 26

*Рекомендовано до друку рішенням вченої ради Житомирського державного університету імені Івана Франка від «29» травня 2026 року, протокол № 11*

**Рецензенти:**

**Міца Олександр** – доктор технічних наук, професор, професор кафедри інформаційних управляючих систем та технологій ДВНЗ "УжНУ".

**Наконечна Оксана** – кандидат технічних наук, доцент, доцент кафедри інформаційних технологій Одеського державного аграрного університету.

**Усата Олена** – кандидат педагогічних наук, доцент, завідувач кафедри комп’ютерних наук та інформаційних технологій Житомирського державного університету імені Івана Франка.

В 26

«Вебтехнологій та вебдизайн: конспект лекцій» / Укладач: Федорчук А. Л. Житомир: Вид-во ЖДУ ім. Івана Франка, 2026. 194 с.

Матеріали лекційного курсу до обов'язкової освітньої компоненти "Вебтехнології та вебдизайн" розроблені відповідно до навчальної та робочої програми для здобувачів першого (бакалаврського) рівня вищої освіти Житомирського державного університету імені Івана Франка.

Конспект лекцій охоплює основи HTML і CSS, сучасні технології верстки CSS Flexbox і CSS Grid, адаптивний дизайн, основи програмування JavaScript та роботу з JS-фреймворками.

УДК 004.774(075.8)

# Зміст

# ВСТУП

Сучасний світ неможливо уявити без вебтехнологій. Вони пронизують усі сфери нашого життя – від освіти та бізнесу до розваг і комунікації. Щодня мільярди людей взаємодіють із вебсайтами та вебдодатками, навіть не замислюючись над складними технологіями, що стоять за їхньою роботою. Саме тому підготовка фахівців, здатних створювати якісні, сучасні та зручні веброзробки, є одним із пріоритетних завдань вищої освіти в галузі інформаційних технологій.

Мета вивчення освітньої компоненти «Вебтехнології та вебдизайн» полягає у підготовці майбутнього фахівця, який володіє засобами проєктування та верстки сучасних сайтів із використанням технологій HTML, CSS, CSS Flex, CSS Grid, JavaScript та вміє пояснювати їх застосування.

Основні завдання курсу:

- розкриття особливостей створення сучасних вебсайтів та інтернет-сервісів;
- навчання працювати з найпопулярнішими FrontEnd-технологіями;
- ознайомлення з популярними бібліотеками та фреймворками;
- формування практичних навичок розробки адаптивних та інтерактивних вебсторінок;
- розвиток здатності самостійно опановувати сучасні інструменти веброзробки.

У лекційному курсі розглядаються базові принципи верстки (семантична розмітка, структура HTML-документа), стильове оформлення сторінок (каскадні таблиці стилів CSS, селектори, властивості), макет CSS, HTML-форми (елементи введення, валідація, обробка даних), технологія CSS Flexbox, технологія CSS Grid, скриптову мову програмування JavaScript (основи мови, робота з DOM, обробка подій, React).

Веброзробка є однією з найдинамічніших галузей IT-індустрії. Технології, що ще кілька років тому вважалися інноваційними, сьогодні стають стандартом. Курс «Вебтехнології та вебдизайн» побудовано з урахуванням сучасних тенденцій:

- Mobile-first підхід – створення сайтів, оптимізованих насамперед для мобільних пристроїв.
- Адаптивний дизайн – коректне відображення на екранах різних

розмірів.

- Семантична верстка – покращення доступності та SEO-оптимізації.
- Інтерактивність – динамічна взаємодія з користувачем засобами JavaScript.

Цей конспект розроблено для здобувачів, які вивчають основи HTML та CSS, технології верстки вебдокументів, скриптову мову програмування JavaScript, JS-фреймворки. Він також стане корисним для тих, хто прагне поглибити свої знання у сфері опанування сучасних вебтехнологій. Навчальне видання стане у нагоді здобувачам першого (бакалаврського) рівня вищої освіти освітньо-професійних програм "Професійна освіта (Цифрові технології)", Середня освіта (Інформатика), "Комп'ютерні науки" та "Інформаційні системи та технології".

Після опанування курсу здобувач зможе:

- створювати семантично правильну HTML-розмітку вебсторінок;
- застосовувати CSS для стилізації та позиціонування елементів;
- використовувати Flexbox і Grid для побудови сучасних макетів;
- розробляти адаптивні сайти для різних пристроїв;
- писати JavaScript-код для забезпечення інтерактивності;
- працювати з DOM-моделлю та обробляти події;
- застосовувати набуті знання для вирішення практичних завдань веброзробки.

Опанування матеріалу цього конспекту закладе міцний фундамент для подальшого професійного розвитку в галузі веброзробки та дозволить впевнено рухатися до вивчення більш складних технологій і фреймворків.

# Модуль І. Основи HTML та CSS

## Тема 1.1. Поняття HTML.

**Мета:** сформувати розуміння природи мови HTML, навчити створювати коректно структурований HTML-документ, ознайомитися з принципами верстки, розрізняти елементи та атрибути, застосовувати теги оформлення тексту, гіперпосилання, зображення, семантичні теги коментарі та спеціальні символи.

**План**

1. Поняття HTML.
2. Структура HTML-документу.
3. Елементи та атрибути HTML.
4. Оформлення тексту.
5. Гіперпосилання та зображення в HTML.
6. Рядкові і блокові елементи.
7. Семантичні теги.
8. Коментарі та спецсимволи.

### 1. Поняття HTML.

HTML (HyperText Markup Language – "мова гіпертекстової розмітки") – мова розмітки, що описує структуру вебсторінки. HTML не є мовою програмування – вона не виконує логіку, а лише визначає, які елементи присутні на сторінці та в якому порядку.

Сучасний стандарт – **HTML5**, підтримується всіма актуальними браузерами. Специфікацію підтримує консорціум W3C та WHATWG.

Три складові сучасної вебсторінки:

| Технологія | Відповідає за |
|---|---|
| HTML | Формує структуру сторінки (тіло) |
| CSS | визначає її зовнішній вигляд (одяг) |
| JavaScript | надає динаміку (поведінка) |

### 2. Структура HTML-документа

Будь-який HTML-документ має обов'язкову базову структуру:

```
<!DOCTYPE html>
<html>
<head>
<title>Назва сторінки</title>
</head>
<body>
<h1>Мій перший заголовок</h1>
<p>Мій перший параграф.</p>
</body>
</html>
```

**Розбір кожного елемента:**

*<!DOCTYPE html>* – декларація типу документа. Повідомляє браузеру, що документ написаний за стандартом HTML5. Обов'язковий перший рядок кожного файлу.

<html> – кореневим елементом HTML сторінки.

*<head>* – службова секція, не відображається на сторінці. Містить метадані: кодування, заголовок, підключення стилів і скриптів.

*<title>* – заголовок сторінки, відображається у вкладці браузера та в результатах пошуку.

*<body>* – основна секція, весь видимий вміст сторінки розміщується тут.

*<h1>* – визначає великий заголовок на HTML сторінці.

* * – визначає абзац (параграф) в HTML документі.

### 3. Елементи та атрибути

**Елемент** – базова одиниця HTML. Складається з відкриваючого тега, вмісту та закриваючого тега:

```
<tagname>Контент йде тут...</tagname>
```

Приклад,

| **Початковий тег** | **Вміст елемента** | **Кінцевий тег** |
|---|---|---|
| <h1> | Мій перший заголовок | </h1> |
|  | Мій перший параграф. |  |

| | | |
|---|---|---|
| <br> | *немає* | *немає* |

Деякі елементи є самозакривними – вони не мають вмісту і не потребують закриваючого тега.

**Атрибути** – додаткові параметри елемента, що уточнюють його поведінку або властивості. Записуються всередині відкриваючого тега у форматі:

***<ім'я_тега ім'я_атрибута = "значення">***

Наприклад,

<a href="https://example.com" target="_blank">Посилання</a>

Тут href і target – атрибути елемента <a>.

У більшості випадків атрибути є необов'язковими і вказуються тільки при необхідності.

**Глобальні атрибути** – працюють з будь-яким елементом:

- id – унікальний ідентифікатор елемента на сторінці
- class – клас для групування елементів, використовується в CSS
- style – інлайн-стилі (не рекомендується для регулярного використання)
- title – підказка при наведенні курсора
- lang – мова вмісту елемента

### 4. Оформлення тексту

Основні теги для структурування та оформлення текстового вмісту:

**Заголовки** – ієрархія від h1 до h6. H1 – головний заголовок сторінки, один на документ. H2–H6 – підзаголовки різних рівнів:

<h1>Головний заголовок</h1>

<h2>Підзаголовок другого рівня</h2>

<h3>Підзаголовок третього рівня</h3>

**Абзац і перенесення рядка:**

Це окремий абзац тексту.

Це інший абзац.<br>Це новий рядок всередині абзацу.

**Виділення тексту:**

Важливий текст (жирний)

Акцентований текст (курсив)

<mark>Позначений текст (підсвічування)</mark>

<del>Видалений текст</del>

<ins>Вставлений текст</ins>

<sub>Підрядковий</sub> та <sup>надрядковий</sup> текст
<code>Код у рядку тексту</code>

**Горизонтальна лінія:**

<hr>

**Списки:**

```
<!-- Невпорядкований список -->
<ul>
  <li>Перший елемент</li>
  <li>Другий елемент</li>
</ul>
<!-- Впорядкований список -->
<ol>
  <li>Перший елемент</li>
  <li>Другий елемент</li>
</ol>
<!-- Список визначень -->
<dl>
  <dt>HTML</dt>
  <dd>Мова розмітки гіпертексту</dd>
</dl>
```

**Таблиці**

Таблиці в HTML призначені для представлення табличних даних – інформації, що має рядки і стовпці. Не використовуються для верстки макету сторінки.

**Базова структура таблиці:**

```
<table>
  <tr>
    <td>Клітинка 1</td>
    <td>Клітинка 2</td>
  </tr>
  <tr>
    <td>Клітинка 3</td>
    <td>Клітинка 4</td>
  </tr>
</table>
```

<table> – контейнер таблиці.

<tr> (table row) – рядок таблиці.

<td> (table data) – клітинка з даними.

<th> (table header) – клітинка заголовка, відображається жирним і по центру за замовчуванням.

**Семантична структура таблиці:**

```
<table>
  <caption>Розклад занять</caption>
  <thead>
    <tr>
      <th>День</th>
      <th>Предмет</th>
      <th>Час</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Понеділок</td>
      <td>HTML та CSS</td>
      <td>10:00</td>
    </tr>
    <tr>
      <td>Середа</td>
      <td>JavaScript</td>
      <td>12:00</td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td colspan="3">Всього: 2 заняття на тиждень</td>
    </tr>
  </tfoot>
</table>
```

<caption> – підпис таблиці, розміщується першим всередині <table>.

<thead> – секція заголовків стовпців.

<tbody> – основна секція з даними.

<tfoot> – підсумкова секція.

**Об'єднання клітинок:**

*<!-- Об'єднання по горизонталі (colspan) -->*

<td colspan="2">Ця клітинка займає 2 стовпці</td>

*<!-- Об'єднання по вертикалі (rowspan) -->*

<td rowspan="3">Ця клітинка займає 3 рядки</td>

colspan – кількість стовпців, які займає клітинка.

rowspan – кількість рядків, які займає клітинка. При об'єднанні клітинки, що поглинаються, прибираються з розмітки.

**5. Гіперпосилання та зображення**

**Гіперпосилання** – елемент <a> з обов'язковим атрибутом href.

Атрибут href вказує адресу призначення посилання

Текст посилання є видимою частиною.

*<!-- Зовнішнє посилання -->*

<a href="https://example.com" target="_blank">Відкрити сайт</a>

*<!-- Внутрішнє посилання (інша сторінка сайту) -->*

<a href="about.html">Про нас</a>

*<!-- Якірне посилання (перехід до елемента на сторінці) -->*

<a href="#contacts">Перейти до контактів</a>

*<!-- Посилання на email -->*

<a href="mailto:info@example.com">Написати нам</a>

*<!-- Посилання на телефон -->*

<a href="tel:+380441234567">Зателефонувати</a>

За замовчуванням посилання буде виглядати так (у всіх браузерах):

- Невідвідуване посилання – підкреслене і синім кольором.
- Відвідуване посилання – підкреслене і фіолетовим кольором.
- Активне посилання – підкреслене і червоним кольором.

Атрибут target вказує, де відкрити пов'язаний документ.

Атрибут target може мати одне із наступних значень:

- _blank – відкриває пов'язаний документ в новому вікні або вкладці.
- _self – відкриває пов'язаний документ в тому ж вікні / вкладці, в якому він був натиснутий (за замовчуванням).
- _parent – відкриває пов'язаний документ в батьківському фреймі.
- _top – відкриває пов'язаний документ в повному тілі вікна.

**Зображення** – самозакривний елемент <img>:

<img src="images/photo.jpg" alt="Опис зображення" width="800" height="600">

Обов'язкові атрибути:

- src – шлях до файлу зображення (відносний або абсолютний)
- alt – альтернативний текст: відображається якщо зображення не завантажилось, читається екранними зчитувачами, враховується пошуковими системами

Відносні шляхи до файлів:

```
<img src="photo.jpg" alt="">          <!-- той самий каталог -->
<img src="images/photo.jpg" alt="">   <!-- підкаталог -->
<img src="../photo.jpg" alt="">       <!-- каталог вище -->
```

### 6. Рядкові і блокові елементи

Усі HTML-елементи за замовчуванням належать до одного з двох типів відображення.

**Блокові елементи** займають всю доступну ширину рядка і починаються з нового рядка. Можуть містити інші блокові та рядкові елементи:

, <p>, <h1>–<h6>, <ul>, <ol>, <li>, <table>, <form>, <header>, <footer>, <section>

**Рядкові елементи** займають лише стільки місця, скільки потребує їх вміст. Розміщуються в межах рядка тексту. Не можуть містити блокові елементи:

, <a>, , , <img>, <input>, <label>, <code>, <br>

Практичне значення: блокові елементи формують структуру сторінки, рядкові – оформлюють вміст усередині блоків.

### 7. Семантичні теги

Семантичні теги описують значення і роль вмісту, а не лише його зовнішній вигляд. Введені в HTML5 для покращення структури документа, доступності та SEO.

**Основні семантичні теги:**

| | |
|---|---|
| <header> | шапка сторінки або секції |
| <nav> | блок навігації |
| <main> | основний вміст сторінки (один на документ) |
| <section> | тематична секція |
| <article> | самостійний незалежний вміст (стаття, пост) |
| <aside> | додатковий вміст (бічна панель, виноски) |
| <footer> | підвал сторінки або секції |
| <figure> | ілюстрація, діаграма або медіаконтент |
| <figcaption> | підпис до figure |

| | |
|---|---|
| <time> | дата або час |
| <address> | контактна інформація |

Приклад типової структури сторінки:

```
<body>
  <header>
    <nav>...</nav>
  </header>
  <main>
    <section>
      <article>...</article>
    </section>
    <aside>...</aside>
  </main>
  <footer>...</footer>
</body>
```

Семантичні теги не замінюють  та  повністю – ці контейнери залишаються корисними там, де немає смислового навантаження і потрібен лише структурний або стильовий контейнер.

### 8. Коментарі та спецсимволи

**Коментарі** – текст, який браузер ігнорує. Використовуються для пояснення коду або тимчасового вимкнення розмітки:

```
<!-- Це коментар -->
<!--
  Багаторядковий коментар.
  Браузер цього не відобразить.
-->
```

**Спецсимволи (HTML-entities)** – спосіб відобразити символи, які мають особливе значення в HTML або відсутні на клавіатурі:

| Символ | Код | Опис |
|---|---|---|
| &lt; | < | Менше (відкриваючий тег) |
| &gt; | > | Більше (закриваючий тег) |

| | | |
|---|---|---|
| &amp; | & | Амперсанд |
| &quot; | " | Подвійні лапки |
|   | (пробіл) | Нерозривний пробіл |
| &copy; | © | Знак авторського права |
| &trade; | ™ | Знак торгової марки |
| &mdash; | — | Довге тире |

```
<p>Ціна товару: 100 грн.</p>
<p>&copy; 2025 Усі права захищені.</p>
<p>5 &lt; 10 &amp;&amp; 10 &gt; 5</p>
```

***Контрольні питання:***

1. Що таке HTML і для чого він використовується?
2. Що таке вебсторінка та з чого вона складається?
3. Яка структура HTML-документу?
4. Яке призначення тегів <html>, <head> та <body>?
5. Що таке HTML-елемент?
6. Що таке атрибут HTML і для чого він використовується?
7. Які основні принципи верстки HTML-сторінок?
8. Які теги використовуються для оформлення тексту?
9. Як створюються гіперпосилання в HTML?
10. Які атрибути використовуються в тегу <a>?
11. Як вставити зображення на вебсторінку?
12. Які атрибути є обов’язковими для тегу <img>?
13. У чому різниця між рядковими та блоковими елементами?
14. Наведіть приклади рядкових елементів.
15. Наведіть приклади блокових елементів.
16. Що таке семантичні теги і навіщо вони потрібні?
17. Назвіть основні семантичні теги HTML.
18. Як створюються коментарі в HTML?
19. Для чого використовуються спеціальні символи?
20. Наведіть приклади спецсимволів у HTML.

# Тема 1.2. Основи CSS.

**Мета:** сформувати розуміння природи та призначення CSS, навчити підключати стилі до HTML-документа, використовувати селектори, розуміти принцип каскадування та специфічності, працювати з одиницями виміру і блоковою моделлю елемента.

**План**

1. Що таке CSS. Місце CSS у веброзробці.
2. Синтаксис CSS.
3. Способи підключення CSS до HTML-сторінки.
4. Селектори та їх групування.
5. Каскадування та специфічність.
6. Одиниці виміру в CSS.
7. Блокова модель елемента.

## 1. Що таке CSS. Місце CSS у веброзробці

CSS (Cascading Style Sheets – "каскадні таблиці стилів") – мова опису зовнішнього вигляду HTML-документа. CSS відповідає за кольори, шрифти, відступи, розміри, розташування елементів та анімації.

Розділення відповідальності між HTML і CSS – базовий принцип веброзробки: HTML описує структуру і зміст, CSS – зовнішній вигляд. Це дозволяє змінювати дизайн сторінки без втручання в її розмітку.

Сучасний стандарт – **CSS3**, який розвивається модульно: кожен аспект (кольори, анімації, сітки) має власну специфікацію.

## 2. Синтаксис CSS

Основна одиниця CSS – **правило**, що складається з селектора і блоку оголошень:

```
селектор {
  властивість: значення;
  властивість: значення;
}
```

Селектор вказує на елемент HTML, який ви хочете стилізувати.

Блок оголошень містить одну або кілька оголошень, розділених крапкою з комою.

Кожне оголошення включає назву CSS-властивості та значення, розділені двокрапкою.

Оголошення CSS завжди закінчюється крапкою з комою, а блоки оголошень заключаються у фігурні дужки.

Приклад:

```
p {
  color: #333333;
  font-size: 16px;
  line-height: 1.5;
}
```

**Складові правила:**

p – селектор, вказує до яких елементів застосовується правило.

color, font-size, line-height – властивості.

#333333, 16px, 1.5 – значення.

**Коментарі в CSS:**

*файл css*

```
/* Це коментар у CSS */
/*
  Багаторядковий коментар.
  Використовується для пояснення блоків стилів.
*/
```

**3. Способи підключення CSS до HTML-сторінки**

Існує три способи підключення стилів, кожен з яких має свою сферу застосування.

**Зовнішній файл** – рекомендований спосіб. Стилі виносяться в окремий файл з розширенням .css і підключаються через тег <link> у секції <head>:

```
<head>
  <link rel="stylesheet" href="styles.css">
</head>
```

Переваги: один файл стилів для всіх сторінок сайту, кешується браузером, легко підтримувати.

**Внутрішні стилі** – розміщуються у тегу <style> у секції <head> в файлі html. Застосовуються лише до поточної сторінки:

```
<head>
  <style>
    p {
      color: navy;
    }
  </style>
```

</head>

Використовується для сторінок з унікальними стилями або під час прототипування.

**Інлайн-стилі** – записуються безпосередньо в атрибуті style конкретного елемента в файлі html:

Текст абзацу

Не рекомендуються для регулярного використання: важко підтримувати, мають найвищу специфічність, змішують структуру і стиль.

| Спосіб | Де записується | Область дії |
|---|---|---|
| Зовнішній файл | Окремий .css файл | Всі підключені сторінки |
| Внутрішній | Тег <style> у <head> | Поточна сторінка |
| Інлайн | Атрибут style тега | Конкретний елемент |

**4. Селектори та їх групування**

Селектор визначає, до яких елементів HTML застосовується CSS-правило.

**Селектор тега** – застосовується до всіх елементів вказаного типу:

p { color: gray; }

h1 { font-size: 32px; }

**Селектор класу** – застосовується до елементів з відповідним атрибутом class. Позначається крапкою:

*файл css*

.highlight { background-color: yellow; }

.btn { padding: 10px 20px; }

*файл html*

Виділений абзац

<a class="btn">Кнопка</a>

Один елемент може мати кілька класів: class="btn btn-primary".

**Селектор ідентифікатора** – застосовується до одного унікального елемента з атрибутом id. Позначається решіткою. Використовується рідко через високу специфічність:

#header { background-color: #f5f5f5; }

*файл html*

<header id="header">...</header>

**Універсальний селектор** – застосовується до всіх елементів:

```
* { box-sizing: border-box; }
```

**Селектор атрибута** – вибирає елементи за наявністю або значенням атрибута:

```
input[type="text"] { border: 1px solid #ccc; }
a[target="_blank"] { color: blue; }
```

**Комбіновані селектори:**

```
/* Нащадок: всі p всередині div */
div p { color: gray; }
/* Дочірній елемент: лише прямі нащадки */
ul > li { list-style: none; }
/* Сусідній елемент: h2 одразу після h1 */
h1 + h2 { margin-top: 0; }
/* Всі наступні сусіди */
h1 ~ p { color: gray; }
```

**Псевдокласи** – вибирають елементи залежно від їх стану:

```
a:hover { color: red; }
input:focus { border-color: blue; }
li:first-child { font-weight: bold; }
li:last-child { border-bottom: none; }
li:nth-child(2n) { background: #f0f0f0; }
```

**Псевдоелементи** – дозволяють стилізувати частину елемента:

```
p::first-line { font-weight: bold; }
p::first-letter { font-size: 2em; }
.item::before { content: "→ "; }
.item::after { content: " ←"; }
```

**Групування селекторів** – кілька селекторів з однаковими правилами об'єднуються через кому:

```
h1, h2, h3 {
  font-family: Arial, sans-serif;
  color: #222;
}
```

**5. Каскадування та специфічність**

**Каскадування** – механізм визначення, яке правило застосовується до елемента, якщо кілька правил конкурують між собою. Браузер враховує три фактори у порядку пріоритету:

1. **Специфічність** – вага селектора.

2. **Порядок оголошення** – пізніше оголошене правило перекриває попереднє.
3. **Походження** – авторські стилі, стилі браузера, користувацькі стилі.

**Специфічність** розраховується за чотирма рівнями:

| Рівень | Що додає | Вага |
|---|---|---|
| Інлайн-стиль | style="..." | 1000 |
| Ідентифікатор | #id | 100 |
| Клас, псевдоклас, атрибут | .class, :hover, [attr] | 10 |
| Тег, псевдоелемент | p, ::before | 1 |

Приклад порівняння специфічності:

```
p { color: black; }          /* специфічність: 1 */
.text { color: blue; }       /* специфічність: 10 */
#intro { color: green; }     /* специфічність: 100 */
```

Якщо до одного елемента <p class="text" id="intro"> застосовані всі три правила – переможе #intro з найвищою специфічністю, колір буде зеленим.

**!important** – примусово підвищує пріоритет правила до максимального. Використовується лише у крайніх випадках, оскільки ускладнює підтримку коду:

```
p { color: red !important; }
```

**Успадкування** – деякі CSS-властивості автоматично успадковуються дочірніми елементами від батьківських. Успадковуються: color, font-family, font-size, line-height. Не успадковуються: margin, padding, border, width, background.

### 6. Одиниці виміру в CSS

**Абсолютні одиниці** – фіксований розмір, не залежить від контексту:

| Одиниця | Опис |
|---|---|
| px | Піксель – основна одиниця екранної верстки |
| pt | Пункт (1pt = 1/72 дюйма) – переважно для друку |

| | |
|---|---|
| cm, mm | Сантиметри, міліметри – для друку |

**Відносні одиниці** – розмір залежить від іншого значення:

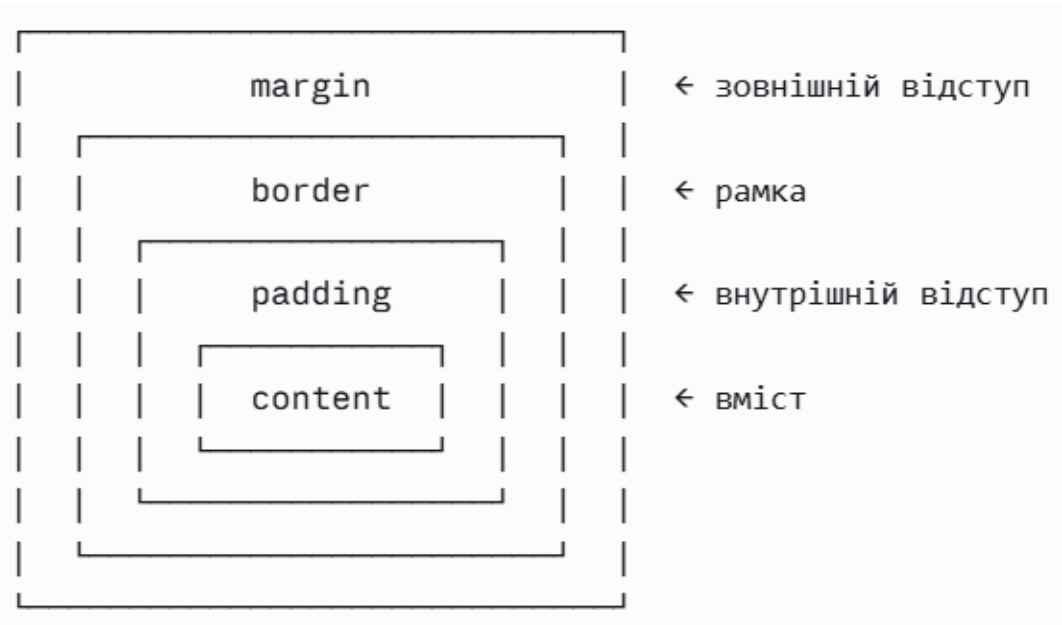
| **Одиниця** | **Відносно чого** |
|---|---|
| % | Відсоток від розміру батьківського елемента |
| em | Розмір шрифту поточного елемента |
| rem | Розмір шрифту кореневого елемента <html> |
| vw | 1% ширини вікна браузера |
| vh | 1% висоти вікна браузера |

Практичні рекомендації:

- px – для точних фіксованих розмірів.
- rem – для розмірів шрифтів і відступів (зручно масштабувати).
- % – для ширини контейнерів.
- vw/vh – для елементів, що мають займати частину екрана.

### 7. Блокова модель елемента

Кожен HTML-елемент у браузері займає прямокутну область, що складається з чотирьох зон. Це і є **блокова модель (box model)**:



**content** – область вмісту елемента. Розмір задається через width та height.

**padding** – внутрішній відступ між вмістом і рамкою. Успадковує фоновий колір елемента:

```
padding: 20px;                  /* всі сторони */
padding: 10px 20px;             /* верх/низ  ліво/право */
padding: 10px 20px 15px 25px;   /* верх право низ ліво */
padding-top: 10px;              /* окрема сторона */
```

**border** – рамка навколо padding. Має три характеристики – товщину, стиль і колір:

```
border: 1px solid #cccccc;
border-top: 2px dashed red;
border-radius: 8px;             /* заокруглення кутів */
```

**margin** – зовнішній відступ між елементом і сусідніми елементами. Прозорий, не успадковує фон:

```
margin: 20px;
margin: 0 auto;                 /* горизонтальне центрування блока */
margin-bottom: 16px;
```

**Властивість box-sizing** – визначає, як браузер розраховує загальний розмір елемента:

```
/* За замовчуванням: width стосується лише content */
box-sizing: content-box;
/* width = content
   реальна ширина = content + padding + border */
/* Рекомендований підхід: width включає padding і border */
box-sizing: border-box;
/* width = content + padding + border
   реальна ширина = width */
```

Практично завжди зручніше використовувати border-box. Типове глобальне правило на початку CSS-файлу:

```
* {
  box-sizing: border-box;
}
```

**Схлопування відступів (margin collapse)** – вертикальні зовнішні відступи сусідніх блокових елементів не підсумовуються, а збігаються – залишається більший з двох:

```
.block-1 { margin-bottom: 30px; }
.block-2 { margin-top: 20px; }
```

/* Відстань між блоками буде 30px, а не 50px */

## Контрольні запитання

1. Що таке CSS і яке його призначення у веброзробці?
2. З яких складових складається CSS-правило?
3. Назвіть три способи підключення CSS до HTML-сторінки та поясніть відмінність між ними.
4. Який спосіб підключення CSS вважається рекомендованим і чому?
5. Що таке селектор? Назвіть основні типи селекторів.
6. Чим відрізняється селектор класу від селектора ідентифікатора?
7. Яка різниця між селектором нащадка та дочірнім селектором?
8. Що таке псевдоклас і псевдоелемент? Наведіть по одному прикладу кожного.
9. Як згрупувати кілька селекторів з однаковими властивостями?
10. Що таке каскадування і які фактори враховує браузер при визначенні пріоритету правила?
11. Що таке специфічність селектора? Розставте за зростанням специфічності: p, .intro, #title, інлайн-стиль.
12. Що таке успадкування в CSS? Наведіть приклад властивості, яка успадковується, і яка не успадковується.
13. Для чого використовується !important і чому його не рекомендується застосовувати регулярно?
14. Чим відрізняються абсолютні одиниці виміру від відносних? Наведіть приклади кожних.
15. Яка різниця між em і rem?
16. З яких чотирьох зон складається блокова модель елемента?
17. Яка різниця між padding і margin?
18. Яка різниця між box-sizing: content-box і box-sizing: border-box?
19. Якщо елемент має width: 200px, padding: 20px, border: 2px при box-sizing: content-box – яка його реальна ширина?
20. Що таке схлопування відступів (margin collapse) і за яких умов воно відбувається?

# Тема 2. Макет CSS

**Мета:** сформувати розуміння механізмів керування розташуванням елементів на сторінці засобами CSS – позиціонування, властивості display та float; навчити обирати відповідний інструмент залежно від задачі верстки.

**План**

1. Нормальний потік документа.
2. Властивість position та її значення.
3. Поняття контексту позиціонування.
4. Властивість display та її значення.
5. Властивість float.

### 1. Нормальний потік документа

Перш ніж розглядати позиціонування, важливо зрозуміти, як браузер розміщує елементи за замовчуванням.

**Нормальний потік (normal flow)** – стандартний порядок відображення елементів на сторінці без застосування будь-яких інструментів позиціонування чи компонування. Браузер розміщує елементи зверху вниз і зліва направо відповідно до їх типу.

*Блокові елементи* ( , <p>, <h1>) займають усю доступну ширину і розміщуються один під одним.

*Рядкові елементи* (, <a>, ) розміщуються в рядку тексту поряд один з одним.

Будь-який інструмент позиціонування або компонування – це відхилення від нормального потоку або його модифікація.

### 2. Властивість position та її значення

Властивість **position** визначає спосіб позиціонування елемента у документі. Разом з нею використовуються властивості top, right, bottom, left для вказівки зміщення та z-index для керування порядком накладання елементів.

**static** – значення за замовчуванням для всіх елементів. Елемент розміщується у нормальному потоці. Властивості top, right, bottom, left та z-index не діють:

*файл css*

```
div {
  position: static; /* поведінка за замовчуванням */
```

```
}
```

**relative** – елемент залишається у нормальному потоці, але зміщується відносно своєї початкової позиції. Місце, яке елемент займав у потоці, зберігається:

*файл css*

```
.box {
  position: relative;
  top: 20px;   /* зміщення вниз від початкової позиції */
  left: 30px;  /* зміщення вправо від початкової позиції */
}
```

Практичне застосування: створення контексту позиціонування для дочірніх елементів з position: absolute.

**absolute** – елемент виймається з нормального потоку і позиціонується відносно найближчого батьківського елемента з position, відмінним від static. Якщо такого батька немає – відносно вікна браузера. Інші елементи не залишають для нього місця:

*файл css*

```
.parent {
  position: relative; /* контекст позиціонування */
}
.child {
  position: absolute;
  top: 0;
  right: 0; /* розміщення у правому верхньому куті батька */
}
```

Практичне застосування: підказки, спливаючі елементи, значки поверх зображення, кнопка закриття модального вікна.

**fixed** – елемент виймається з потоку і позиціонується відносно вікна браузера. Залишається на місці при прокручуванні сторінки:

*файл css*

```
.header {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
}
```

Практичне застосування: фіксована шапка, кнопка «повернутись вгору», панель чату.

**sticky** – гібрид relative і fixed. Елемент поводиться як relative до досягнення заданої позиції при прокручуванні, після чого фіксується як fixed:

*файл css*

```
.nav {
  position: sticky;
  top: 0; /* фіксується, коли досягає верху вікна */
}
```

Практичне застосування: навігація, що прилипає до верху при прокручуванні; заголовки таблиць.

**Властивість z-index** – керує порядком накладання елементів по осі Z (глибина). Працює лише для елементів з position, відмінним від static. Більше значення – елемент відображається поверх:

*файл css*

```
.modal {
  position: fixed;
  z-index: 1000;
}
.overlay {
  position: fixed;
  z-index: 999;
}
```

### 3. Поняття контексту позиціонування

**Контекст позиціонування** – батьківський елемент, відносно якого позиціонується дочірній елемент з position: absolute.

Правило визначення контексту: браузер шукає найближчого батьківського елемента, у якого властивість position має значення, відмінне від static – relative, absolute, fixed або sticky. Якщо такого батька не знайдено – контекстом стає початковий блок (<html>).

*файл css*

```
/* Приклад 1: контекст – .card */
.card {
  position: relative; /* ← контекст позиціонування */
}
.card__badge {
```

```
  position: absolute;
  top: 10px;
  right: 10px; /* розміщується у куті .card */
}
```

*файл css*

```
/* Приклад 2: контекст – <html>, бо немає відносного батька */
.badge {
  position: absolute;
  top: 10px;
  right: 10px; /* розміщується у куті вікна браузера */
}
```

Розуміння контексту позиціонування – ключ до передбачуваної поведінки position: absolute. Найпоширеніша помилка – абсолютно позиційований елемент розміщується не там, де очікувалось, через відсутність або неправильний контекст.

**4. Властивість display та її значення**

Властивість display визначає тип відображення елемента – як він поводиться у потоці документа і як розміщуються його дочірні елементи.

**block** – елемент відображається як блоковий: займає всю доступну ширину, починається з нового рядка, можна задати width, height, margin, padding:

*файл css*

```
span {
  display: block; /* рядковий span стає блоковим */
}
```

**inline** – елемент відображається як рядковий: займає лише необхідну ширину, не починає новий рядок. Властивості width, height та вертикальні margin не діють:

*файл css*

```
div {
  display: inline; /* блоковий div стає рядковим */
}
```

**inline-block** – поєднує властивості обох типів: розміщується в рядку як рядковий елемент, але приймає width, height, margin та padding як блоковий:

*файл css*

```
.btn {
  display: inline-block;
```

```
  width: 120px;
  padding: 10px 20px;
}
```

Практичне застосування: кнопки, елементи навігації, іконки з текстом поряд.

**none** – елемент повністю прихований і не займає місця у потоці документа. Відрізняється від visibility: hidden, яке ховає елемент, але зберігає його місце:

*файл css*

```
.hidden {
  display: none; /* елемент відсутній на сторінці */
}
.invisible {
  visibility: hidden; /* елемент невидимий, але місце зберігається */
}
```

Практичне застосування display: none: приховування елементів для мобільних пристроїв, реалізація випадаючих меню, модальних вікон, вкладок.

Зведена таблиця значень:

| **Значення** | **Новий рядок** | **width/height** | **Поруч з іншими** |
|---|---|---|---|
| block | Так | Так | Ні |
| inline | Ні | Ні | Так |
| inline-block | Ні | Так | Так |
| none | – | – | Відсутній |

**5. Властивість float**

float – властивість, що виштовхує елемент до лівого або правого краю контейнера і дозволяє тексту та рядковим елементам обтікати його.

*файл css*

```
img {
  float: left;  /* зображення притискається вліво, текст обтікає справа */
  margin: 0 16px 16px 0;
}
```

Значення: left, right, none (за замовчуванням).

**Проблема float** – плаваючі елементи виймаються з нормального потоку, через що батьківський контейнер може «не бачити» їх висоти і стискатись до нуля. Для вирішення застосовувався прийом **clearfix**:

*файл css*

```
.clearfix::after {
  content: "";
  display: block;
  clear: both;
}
```

**Важливо:** float є застарілим інструментом для побудови макетів. У сучасній верстці його замінили Flexbox і Grid. Знання float необхідне для розуміння і підтримки legacy-коду, написаного до 2015–2017 років. У нових проєктах float використовується лише для обтікання текстом зображень або медіаконтенту – це його початкове і досі актуальне призначення.

## Контрольні питання

1. Що таке нормальний потік документа і як у ньому розміщуються блокові та рядкові елементи?
2. Для чого призначена властивість position? Які значення вона приймає?
3. Чим відрізняється position: relative від position: static?
4. Що відбувається з елементом у потоці документа при застосуванні position: absolute?
5. Відносно чого позиціонується елемент з position: absolute, якщо жоден батьківський елемент не має явно заданого position?
6. Чим position: fixed відрізняється від position: absolute?
7. Як поводиться елемент з position: sticky при прокручуванні сторінки?
8. Наведіть практичний приклад застосування кожного значення властивості position.
9. Що таке контекст позиціонування і як браузер його визначає?
10. Яка найпоширеніша помилка при роботі з position: absolute і як її уникнути?
11. Для чого використовується властивість z-index? За яких умов вона діє?
12. Що визначає властивість display?
13. Чим display: inline-block відрізняється від display: inline?
14. Чому до елемента з display: inline не можна застосувати width і height?

15. Яка різниця між display: none і visibility: hidden?
16. Наведіть практичний приклад використання display: inline-block.
17. Для чого призначена властивість float і які значення вона приймає?
18. Чому float вважається застарілим інструментом для побудови макетів?
19. Що таке clearfix і для вирішення якої проблеми він застосовується?
20. У яких випадках використання float залишається актуальним у сучасній верстці?

# Тема 3. Шрифти, меню та псевдокласи

**Мета:** сформувати вміння працювати зі шрифтами у CSS, підключати зовнішні шрифти, будувати різні типи навігаційних меню засобами HTML і CSS, застосовувати псевдокласи та псевдоелементи для стилізації елементів сторінки.

**План**

1. Властивості шрифтів у CSS.
2. Підключення зовнішніх шрифтів.
3. Типи меню та їх побудова.
4. Псевдокласи.
5. Псевдоелементи.

## 1. Властивості шрифтів у CSS

CSS надає повний набір властивостей для керування зовнішнім виглядом тексту.

**font-family** – визначає гарнітуру шрифту. Задається у вигляді списку – браузер використовує перший доступний шрифт. Останнім у списку вказується загальна сім'я шрифтів як запасний варіант:

*файл css*

```
p {
  font-family: 'Roboto', Arial, sans-serif;
}
```

Загальні сім'ї шрифтів: serif (з засічками), sans-serif (без засічок), monospace (моноширинний), cursive (рукописний), fantasy (декоративний).

**font-size** – розмір шрифту. Може задаватись в абсолютних і відносних одиницях:

*файл css*

```
h1 { font-size: 32px; }
p  { font-size: 1rem; }
small { font-size: 0.875em; }
```

**font-weight** – насиченість (жирність) шрифту. Числові значення від 100 до 900 або ключові слова:

*файл css*

```
p { font-weight: normal; }    /* 400 */
b { font-weight: bold; }      /* 700 */
h1 { font-weight: 800; }
```

**font-style** – стиль накреслення:

*файл css*

```
em { font-style: italic; }
p  { font-style: normal; }
```

**line-height** – міжрядковий інтервал. Рекомендоване значення для основного тексту – 1.4–1.6 без одиниць виміру:

*файл css*

```
p { line-height: 1.5; }
```

**letter-spacing та word-spacing** – відстань між символами та словами:

*файл css*

```
h1 { letter-spacing: 0.05em; }
p  { word-spacing: 0.1em; }
```

**text-transform** – перетворення регістра:

*файл css*

```
h2 { text-transform: uppercase; }   /* ВЕЛИКІ ЛІТЕРИ */
p  { text-transform: capitalize; }  /* Перша Велика */
```

**text-decoration** – оформлення тексту лінією:

*файл css*

```
a    { text-decoration: none; }
del  { text-decoration: line-through; }
ins  { text-decoration: underline; }
```

**text-align** – вирівнювання тексту по горизонталі:

*файл css*

```
h1 { text-align: center; }
p  { text-align: justify; }
```

**Скорочений запис font:**

*файл css*

```
p {
  font: italic 400 16px/1.5 'Roboto', sans-serif;
  /*   стиль вага розмір/висота рядка сім'я */
}
```

**2. Підключення зовнішніх шрифтів**

Системні шрифти обмежені і відрізняються залежно від операційної системи. Зовнішні шрифти дозволяють використовувати будь-яку гарнітуру незалежно від системи користувача.

**Google Fonts** – найпоширеніший безкоштовний сервіс шрифтів. Підключення через тег <link> у секції <head>:

*файл html*

```
<head>
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link
href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;700&displa
y=swap" rel="stylesheet">
</head>
```

Після підключення шрифт використовується у CSS як звичайний:

*файл css*

```
body {
  font-family: 'Roboto', sans-serif;
}
```

**Підключення через CSS** – альтернативний спосіб через @import на початку CSS-файлу. Менш рекомендований через затримку завантаження:

*файл css*

```
@import
url('https://fonts.googleapis.com/css2?family=Roboto&display=swap');
```

**Локальні шрифти через @font-face** – підключення шрифтового файлу, розміщеного на власному сервері. Дає повний контроль над завантаженням:

*файл css*

```
@font-face {
  font-family: 'MyFont';
  src: url('fonts/myfont.woff2') format('woff2'),
      url('fonts/myfont.woff') format('woff');
  font-weight: normal;
  font-style: normal;
}
body {
  font-family: 'MyFont', sans-serif;
}
```

Сучасний формат шрифтів – **WOFF2**, забезпечує найкраще стиснення. WOFF – як запасний варіант для старіших браузерів.

### 3. Типи меню та їх побудова

Навігаційне меню – один з найважливіших елементів інтерфейсу. В основі будь-якого меню лежить семантична HTML-структура: елемент <nav> зі списком <ul> і посиланнями <a>.

**Горизонтальне меню** – елементи розміщуються в рядок. Найпоширеніший тип для десктопної навігації:

*файл html*

```
<nav>
  <ul class="menu">
    <li><a href="#">Головна</a></li>
    <li><a href="#">Про нас</a></li>
    <li><a href="#">Послуги</a></li>
    <li><a href="#">Контакти</a></li>
  </ul>
</nav>
```

*файл css*

```
.menu {
  list-style: none;
  margin: 0;
  padding: 0;
  display: flex;
  gap: 24px;
}
.menu a {
  text-decoration: none;
  color: #333;
  font-weight: 500;
}
.menu a:hover {
  color: #0066cc;
}
```

**Вертикальне меню** – елементи розміщуються один під одним. Використовується для бічних панелей:

*файл css*

```
.menu-vertical {
  list-style: none;
  padding: 0;
  width: 200px;
}
.menu-vertical a {
  display: block;
```

```
  padding: 10px 16px;
  text-decoration: none;
  color: #333;
  border-bottom: 1px solid #eee;
}
.menu-vertical a:hover {
  background-color: #f5f5f5;
  color: #0066cc;
}
```

**Випадаюче меню (dropdown)** – при наведенні на пункт відкривається підменю. Реалізується через вкладений список і псевдоклас :hover:

*файл html*

```
<nav>
  <ul class="menu">
    <li>
      <a href="#">Послуги</a>
      <ul class="submenu">
        <li><a href="#">Дизайн</a></li>
        <li><a href="#">Розробка</a></li>
        <li><a href="#">Підтримка</a></li>
      </ul>
    </li>
  </ul>
</nav>
```

*файл css*

```
.menu li {
  position: relative;
}
.submenu {
  display: none;
  position: absolute;
  top: 100%;
  left: 0;
  list-style: none;
  background: #fff;
  box-shadow: 0 4px 12px rgba(0,0,0,0.1);
  min-width: 160px;
```

```
}
.menu li:hover .submenu {
  display: block;
}
.submenu a {
  display: block;
  padding: 10px 16px;
  text-decoration: none;
  color: #333;
}
.submenu a:hover {
  background-color: #f0f0f0;
}
```

**Активний пункт меню** – виділення поточної сторінки:

*файл css*

```
.menu a.active {
  color: #0066cc;
  border-bottom: 2px solid #0066cc;
}
```

*файл html*

```
<li><a href="#" class="active">Головна</a></li>
```

**4. Псевдокласи**

Псевдоклас – ключове слово, що додається до селектора і вибирає елемент залежно від його стану або позиції у документі. Записується через одне двокрапку.

**Псевдокласи стану:**

*файл css*

```
a:hover   { color: red; }          /* при наведенні курсора */
a:active  { color: darkred; }      /* в момент натискання */
a:visited { color: purple; }       /* відвідане посилання */
input:focus { border-color: blue; } /* елемент у фокусі */
input:disabled { opacity: 0.5; }   /* заблокований елемент */
input:checked { accent-color: green; } /* вибраний checkbox/radio */
```

**Псевдокласи структури – вибір за позицією:**

*файл css*

```
li:first-child  { font-weight: bold; }   /* перший дочірній елемент */
li:last-child   { border: none; }        /* останній дочірній елемент */
```

```
li:nth-child(2) { color: red; }          /* другий елемент */
li:nth-child(odd)  { background: #f9f9f9; } /* непарні */
li:nth-child(even) { background: #fff; }    /* парні */
li:nth-child(3n)   { color: blue; }         /* кожен третій */
p:only-child    { font-size: 1.2em; }    /* єдиний дочірній елемент */
```

**Псевдоклас :not()** – вибирає елементи, що не відповідають заданому селектору:

*файл css*

```
li:not(:last-child) {
  border-bottom: 1px solid #eee; /* лінія між усіма, крім останнього */
}
input:not([type="submit"]) {
  border: 1px solid #ccc;
}
```

**Псевдокласи форм:**

*файл css*

```
input:valid   { border-color: green; }  /* коректне значення */
input:invalid { border-color: red; }    /* некоректне значення */
input:required { background: #fffbe6; } /* обов'язкове поле */
```

**5. Псевдоелементи**

Псевдоелемент – ключове слово, що дозволяє стилізувати певну частину елемента або створити віртуальний елемент без додавання HTML. Записується через подвійну двокрапку.

**::before та ::after** – створюють віртуальний елемент до або після вмісту елемента. Обов'язково потребують властивості content:

*файл css*

```
.item::before {
  content: "→ ";
  color: #0066cc;
}
.required::after {
  content: " *";
  color: red;
}
/* Декоративний елемент без тексту */
.divider::after {
  content: "";
```

```
  display: block;
  width: 60px;
  height: 3px;
  background: #0066cc;
  margin-top: 8px;
}
```

**::first-line** – стилізує перший рядок тексту:

*файл css*

```
p::first-line {
  font-weight: bold;
  font-size: 1.1em;
}
```

**::first-letter** – стилізує першу літеру тексту. Використовується для декоративної буквиці:

*файл css*

```
p::first-letter {
  font-size: 3em;
  font-weight: bold;
  float: left;
  margin-right: 8px;
  color: #0066cc;
}
```

**::selection** – стилізує виділений користувачем текст:

*файл css*

```
::selection {
  background-color: #0066cc;
  color: #ffffff;
}
```

**::placeholder** – стилізує текст-підказку в полях введення:

*файл css*

```
input::placeholder {
  color: #aaa;
  font-style: italic;
}
```

**Практичний приклад – оформлення списку через ::before замість стандартних маркерів:**

*файл css*

```
ul {
  list-style: none;
  padding: 0;
}
ul li::before {
  content: "✓";
  color: green;
  font-weight: bold;
  margin-right: 8px;
}
```

**Контрольні питання:**

1. Які CSS-властивості відповідають за зовнішній вигляд шрифту? Назвіть не менше п'яти.
2. Що таке загальна сім'я шрифтів і навіщо вона вказується останньою у значенні font-family?
3. Яка різниця між font-size заданим у px і заданим у rem?
4. Чим відрізняється line-height: 1.5 від line-height: 1.5px?
5. Які способи підключення зовнішніх шрифтів існують? Який з них рекомендований і чому?
6. Для чого використовується правило @font-face?
7. Який формат шрифтових файлів є сучасним стандартом і чому?
8. Яка семантична HTML-структура лежить в основі будь-якого навігаційного меню?
9. Чим відрізняється горизонтальне меню від вертикального з точки зору CSS?
10. Як реалізується випадаюче підменю засобами CSS без JavaScript?
11. Чому для випадаючого меню батьківський елемент <li> потребує position: relative?
12. Як виділити активний пункт меню засобами CSS?
13. Що таке псевдоклас? Чим він відрізняється від звичайного класу?
14. Яка різниця між псевдокласами :hover і :focus?
15. Що робить псевдоклас :not() і наведіть приклад його практичного застосування.
16. Як за допомогою :nth-child() вибрати непарні рядки таблиці?
17. Що таке псевдоелемент і чим він відрізняється від псевдокласу?
18. Навіщо псевдоелементам ::before і ::after обов'язково потрібна властивість content?

19. Як за допомогою ::before або ::after створити декоративний елемент без видимого тексту?
20. Наведіть два практичних приклади використання псевдоелементів у верстці.

# Тема 4. HTML-форми

**Мета:** сформувати вміння створювати HTML-форми різної складності, використовувати різні типи полів введення та їх атрибути, грамотно групувати елементи форми і стилізувати їх засобами CSS.

**План**

1. Призначення форм та базова структура.
2. Елементи та атрибути форми.
3. Типи полів введення.
4. Групування елементів форми.
5. Стилізація форм засобами CSS.

## 1. Призначення форм та базова структура.

HTML-форма – основний інструмент взаємодії користувача з вебсайтом. Форми використовуються для введення даних реєстрації та авторизації, пошукових запитів, замовлень, коментарів, завантаження файлів.

Базова структура форми:

*файл html*

```
<form action="submit.php" method="POST">
  <!-- елементи форми -->
  <button type="submit">Надіслати</button>
</form>
```

**Атрибути тега <form>:**

*action* – URL-адреса, на яку надсилаються дані форми. Якщо не вказано – дані надсилаються на поточну сторінку.

*method* – HTTP-метод передачі даних:

- GET – дані передаються у рядку URL. Використовується для пошуку та фільтрації.
- POST – дані передаються у тілі запиту. Використовується для реєстрації, авторизації, надсилання файлів.

*enctype* – тип кодування даних. Обов'язковий при завантаженні файлів:

*файл html*

```
<form action="upload.php" method="POST" enctype="multipart/form-data">
```

*novalidate* – вимикає вбудовану HTML5-валідацію браузера. Використовується коли валідація реалізована засобами JavaScript.

*autocomplete* – керує автозаповненням полів браузером: on або off.

**2. Елементи та атрибути форми**

**Тег <label>** – підпис до поля введення. Покращує доступність і зручність: клік на підпис переводить фокус у відповідне поле. Зв'язується з полем через атрибут for, що відповідає id поля:

*файл html*

```
<label for="username">Ім'я користувача</label>
<input type="text" id="username" name="username">
```

Альтернативний спосіб – обгорнути поле всередину <label>:

*файл html*

```
<label>
  Ім'я користувача
  <input type="text" name="username">
</label>
```

**Тег <input>** – універсальне поле введення. Тип визначається атрибутом type. Самозакривний елемент.

**Тег <textarea>** – багаторядкове текстове поле. Розмір задається атрибутами rows і cols або через CSS:

*файл html*

```
<label for="message">Повідомлення</label>
<textarea id="message" name="message" rows="5" placeholder="Введіть текст..."></textarea>
```

**Тег <select>** – випадаючий список. Елементи списку задаються тегом <option>:

*файл html*

```
<label for="city">Місто</label>
<select id="city" name="city">
  <option value="">Оберіть місто</option>
  <option value="kyiv">Київ</option>
  <option value="lviv">Львів</option>
  <option value="kharkiv">Харків</option>
</select>
```

Атрибут multiple дозволяє вибрати кілька значень. Атрибут selected позначає вибраний за замовчуванням пункт.

**Тег <button>** – кнопка. Типи: submit – надсилає форму, reset – скидає значення полів, button – без стандартної поведінки, використовується з JavaScript:

*файл html*

<button type="submit">Надіслати</button>
<button type="reset">Очистити</button>
<button type="button">Дія через JS</button>

**Загальні атрибути полів введення:**

| Атрибут | Призначення |
|---|---|
| name | Ім'я поля, надсилається на сервер разом зі значенням |
| id | Унікальний ідентифікатор для зв'язку з <label> |
| value | Початкове або фіксоване значення поля |
| placeholder | Текст-підказка, зникає при введенні |
| required | Поле обов'язкове для заповнення |
| disabled | Поле заблоковане, не надсилається на сервер |
| readonly | Поле лише для читання, надсилається на сервер |
| autofocus | Автоматичний фокус при завантаженні сторінки |
| autocomplete | Керування автозаповненням конкретного поля |
| tabindex | Порядок переходу між полями клавішею Tab |

### 3. Типи полів введення

Тип поля задається атрибутом type тега <input> і визначає зовнішній вигляд, поведінку та вбудовану валідацію.

**Текстові поля:**

*файл html*

```
<!-- Однорядкове текстове поле -->
<input type="text" name="name" placeholder="Ваше ім'я">
<!-- Пароль – символи приховуються -->
<input type="password" name="password">
<!-- Email – валідація формату email -->
<input type="email" name="email" placeholder="example@mail.com">
```

```
<!-- Телефон -->
<input type="tel" name="phone" placeholder="+380991234567">

<!-- URL – валідація формату посилання -->
<input type="url" name="website" placeholder="https://example.com">
<!-- Пошук – з кнопкою очищення -->
<input type="search" name="query" placeholder="Пошук...">
```

**Числові поля:**

*файл html*

```
<!-- Число з обмеженнями -->
<input type="number" name="age" min="1" max="120" step="1">
<!-- Повзунок діапазону -->
<input type="range" name="volume" min="0" max="100" value="50">
```

**Дата та час:**

*файл html*

```
<input type="date" name="birthday">
<input type="time" name="meeting-time">
<input type="datetime-local" name="event">
<input type="month" name="period">
<input type="week" name="week">
```

**Вибір та перемикачі:**

*файл html*

```
<!-- Прапорець (checkbox) – вибір кількох варіантів -->
<input type="checkbox" id="agree" name="agree" value="yes">
<label for="agree">Погоджуюсь з умовами</label>
<!-- Перемикач (radio) – вибір одного з варіантів -->
<!-- Групуються однаковим значенням атрибута name -->
<input type="radio" id="male" name="gender" value="male">
<label for="male">Чоловік</label>
<input type="radio" id="female" name="gender" value="female">
<label for="female">Жінка</label>
```

**Спеціальні типи:**

*файл html*

```
<!-- Завантаження файлу -->
<input type="file" name="avatar" accept="image/*">
<!-- Вибір кольору -->
<input type="color" name="theme-color" value="#0066cc">
```

```
<!-- Приховане поле – не відображається, але надсилається -->
<input type="hidden" name="token" value="abc123">
<!-- Кнопка-зображення -->
<input type="image" src="btn.png" alt="Надіслати">
```

**Атрибути валідації:**

*файл html*

```
<input type="text" required minlength="2" maxlength="50">
<input type="number" min="0" max="100">
<input type="text" pattern="[A-Za-z]{3,}" title="Лише літери, мінімум 3">
```

**4. Групування елементів форми**

**Тег <fieldset>** – об'єднує логічно пов'язані елементи форми у групу, візуально обмежену рамкою.

**Тег <legend>** – підпис групи, розміщується першим всередині <fieldset>:

*файл html*

```
<form>
  <fieldset>
    <legend>Особисті дані</legend>
    <label for="firstname">Ім'я</label>
    <input type="text" id="firstname" name="firstname">
    <label for="lastname">Прізвище</label>
    <input type="text" id="lastname" name="lastname">
  </fieldset>
  <fieldset>
    <legend>Контактна інформація</legend>
    <label for="email">Email</label>
    <input type="email" id="email" name="email">
    <label for="phone">Телефон</label>
    <input type="tel" id="phone" name="phone">
  </fieldset>
  <button type="submit">Зареєструватись</button>
</form>
```

**Тег <datalist>** – підказки для текстового поля. Користувач може обрати зі списку або ввести власне значення:

*файл html*

```
<label for="browser">Браузер</label>
<input type="text" id="browser" name="browser" list="browsers">
<datalist id="browsers">
```

```
  <option value="Chrome">
  <option value="Firefox">
  <option value="Safari">
  <option value="Edge">
</datalist>
```

**Тег <optgroup>** – групування пунктів у <select>:

*файл html*

```
<select name="country">
  <optgroup label="Європа">
    <option value="ua">Україна</option>
    <option value="pl">Польща</option>
  </optgroup>
  <optgroup label="Америка">
    <option value="us">США</option>
    <option value="ca">Канада</option>
  </optgroup>
</select>
```

### 5. Стилізація форм засобами CSS

Браузери мають власні стилі для елементів форм, які відрізняються між собою. Для отримання однакового вигляду на всіх браузерах стилі скидаються і задаються вручну.

**Скидання стандартних стилів:**

*файл css*

```
input,
textarea,
select,
button {
  box-sizing: border-box;
  font-family: inherit;
  font-size: inherit;
}
```

**Стилізація текстових полів:**

*файл css*

```
.form-input {
  width: 100%;
  padding: 10px 14px;
  border: 1px solid #ccc;
```

```
  border-radius: 6px;
  font-size: 1rem;
  color: #333;
  background-color: #fff;
  outline: none;
  transition: border-color 0.2s;
}
.form-input:focus {
  border-color: #0066cc;
  box-shadow: 0 0 0 3px rgba(0, 102, 204, 0.15);
}
.form-input::placeholder {
  color: #aaa;
}
```

**Стилізація кнопки:**

*файл css*

```
.btn {
  display: inline-block;
  padding: 12px 28px;
  background-color: #0066cc;
  color: #fff;
  border: none;
  border-radius: 6px;
  font-size: 1rem;
  cursor: pointer;
  transition: background-color 0.2s;
}
.btn:hover {
  background-color: #0052a3;
}
.btn:active {
  background-color: #003d7a;
}
```

**Стилізація станів валідації:**

*файл css*

```
.form-input:valid {
  border-color: #28a745;
```

```
}
.form-input:invalid:not(:placeholder-shown) {
  border-color: #dc3545;
}
```

**Стилізація <fieldset> та <legend>:**

*файл css*

```
fieldset {
  border: 1px solid #ddd;
  border-radius: 8px;
  padding: 20px;
  margin-bottom: 24px;
}
legend {
  font-weight: 600;
  color: #333;
  padding: 0 8px;
}
```

**Макет форми через CSS:**

*файл css*

```
/* Вертикальний макет */
.form-group {
  display: flex;
  flex-direction: column;
  gap: 6px;
  margin-bottom: 16px;
}
/* Горизонтальний макет: підпис і поле в рядок */
.form-group-horizontal {
  display: flex;
  align-items: center;
  gap: 16px;
  margin-bottom: 16px;
}
.form-group-horizontal label {
  width: 140px;
  flex-shrink: 0;
}
```

**Стилізація checkbox та radio:**

*файл css*

```
input[type="checkbox"],
input[type="radio"] {
  accent-color: #0066cc;
  width: 18px;
  height: 18px;
  cursor: pointer;
}
```

## Контрольні запитання:

1. Для чого призначені HTML-форми? Наведіть три приклади їх використання на вебсайтах.
2. Які атрибути має тег <form> і за що кожен з них відповідає?
3. Яка різниця між методами передачі даних GET і POST? Коли застосовується кожен?
4. Для чого призначений тег <label> і як він пов'язується з полем введення?
5. Назвіть два способи зв'язування <label> з полем введення. Наведіть приклад кожного.
6. Чим відрізняється атрибут disabled від readonly?
7. Які атрибути є обов'язковими для правильної роботи форми і чому?
8. Назвіть не менше п'яти типів поля <input> та поясніть призначення кожного.
9. Яка різниця між <input type="checkbox"> і <input type="radio">? Як групуються перемикачі radio?
10. Для чого використовується <input type="hidden">?
11. Які атрибути HTML5 використовуються для вбудованої валідації полів? Наведіть приклади.
12. Чим <textarea> відрізняється від <input type="text">?
13. Як влаштований елемент <select>? Як позначити пункт вибраним за замовчуванням?
14. Для чого використовується тег <datalist> і чим він відрізняється від <select>?
15. Що таке <fieldset> і <legend>? Коли доцільно їх використовувати?
16. Навіщо при стилізації форм скидати стандартні стилі браузера?

17. Які CSS-властивості використовуються для стилізації поля у стані фокусу?
18. Як за допомогою CSS візуально відобразити коректний і некоректний стан поля введення?
19. Як побудувати горизонтальний макет форми, де підпис і поле розміщуються в один рядок?
20. Що робить властивість accent-color і до яких елементів форми вона застосовується?

# Модуль ІІ. Сучасні CSS-технології та адаптивна верстка

## Тема 5. Технологія CSS Flexbox

**Мета:** сформувати розуміння концепції флексбокс-компонування, навчити оголошувати flex-контейнер, керувати напрямком і вирівнюванням елементів за допомогою властивостей контейнера та flex-елементів, застосовувати flexbox для побудови типових компонентів вебсторінки.

**План**

1. Поняття flexbox та його роль у сучасній верстці.
2. Оголошення flex-контейнера. Основна та поперечна осі.
3. Властивості flex-контейнера.
4. Властивості flex-елементів.
5. Практичне застосування flexbox.

### 1. Поняття flexbox та його роль у сучасній верстці

**Flexbox** (Flexible Box Layout) – модуль CSS, призначений для розміщення елементів в одному напрямку – по рядку або по стовпцю – з гнучким розподілом простору між ними.

До появи flexbox розміщення елементів будувалось на float і позиціонуванні – підходах, що не були призначені для компонування макетів і вимагали складних обхідних рішень. Flexbox вирішує більшість типових задач верстки елегантно і передбачувано.

Flexbox найкраще підходить для:

- навігаційних панелей і меню;
- карткових сіток з рівними висотами;
- центрування елементів по вертикалі та горизонталі;
- розміщення елементів форми;
- будь-яких компонентів з одновимірним розміщенням.

### 2. Оголошення flex-контейнера. Основна та поперечна осі

Для активації flexbox достатньо задати елементу-контейнеру display: flex або display: inline-flex:

*файл css*

```
.container {
  display: flex; /* блоковий flex-контейнер */
}
.container {
```

```
  display: inline-flex; /* рядковий flex-контейнер */
}
```

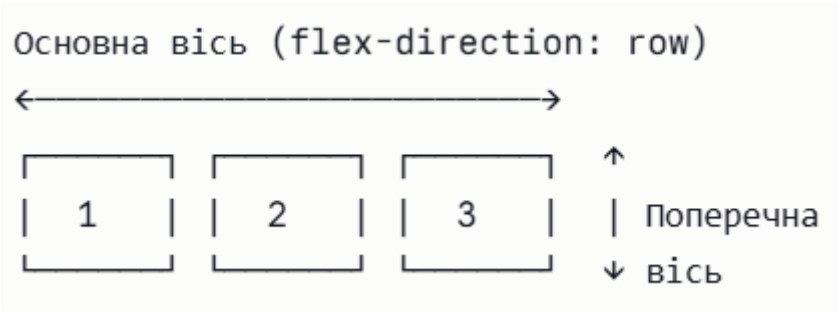
Після цього всі прямі дочірні елементи контейнера стають **flex-елементами** і підпорядковуються правилам flexbox.

**Основна вісь (main axis)** – вісь, вздовж якої розміщуються flex-елементи. За замовчуванням – горизонтальна, зліва направо.

**Поперечна вісь (cross axis)** – перпендикулярна до основної. За замовчуванням – вертикальна, зверху вниз.

Напрямок осей визначається властивістю flex-direction. Розуміння осей критично важливе – всі властивості вирівнювання працюють відносно них.

Основна вісь (flex-direction: row)




### 3. Властивості flex-контейнера

**flex-direction** – визначає напрямок основної осі:

*файл css*

```
.container { flex-direction: row; }            /* → за замовчуванням */
.container { flex-direction: row-reverse; }    /* ← */
.container { flex-direction: column; }         /* ↓ */
.container { flex-direction: column-reverse; } /* ↑ */
```

**flex-wrap** – визначає поведінку елементів при переповненні контейнера:

*файл css*

```
.container { flex-wrap: nowrap; }       /* в один рядок, за замовчуванням */
.container { flex-wrap: wrap; }         /* перенесення на новий рядок */
.container { flex-wrap: wrap-reverse; } /* перенесення у зворотному порядку */
```

**flex-flow** – скорочений запис flex-direction і flex-wrap:

*файл css*

```
.container { flex-flow: row wrap; }
```

**gap** – відстань між flex-елементами. Сучасна заміна відступів через margin:

*файл css*

.container { gap: 16px; }  /* однаковий відступ по обох осях */

.container { gap: 16px 24px; }  /* рядковий і стовпцевий відступи */

.container { row-gap: 16px; }

.container { column-gap: 24px; }

**justify-content** – вирівнювання елементів вздовж **основної** осі:

*файл css*

.container { justify-content: flex-start;  /* до початку – за замовчуванням */

.container { justify-content: flex-end; }  /* до кінця */

.container { justify-content: center; }  /* по центру */

.container { justify-content: space-between; } /* рівні відступи між елементами */

.container { justify-content: space-around; }  /* рівні відступи навколо елементів */

.container { justify-content: space-evenly; }  /* рівні відступи між і навколо */

**align-items** – вирівнювання елементів вздовж **поперечної** осі в межах рядка:

*файл css*

.container { align-items: stretch; }  /* розтягнути – за замовчуванням */

.container { align-items: flex-start; }  /* до початку поперечної осі */

.container { align-items: flex-end; }  /* до кінця поперечної осі */

.container { align-items: center; }  /* по центру поперечної осі */

.container { align-items: baseline; }  /* по базовій лінії тексту */

**align-content** – вирівнювання рядків flex-елементів вздовж поперечної осі. Діє лише при flex-wrap: wrap і кількох рядках:

*файл css*

.container { align-content: flex-start; }

.container { align-content: center; }

.container { align-content: space-between; }

.container { align-content: stretch; }

**4. Властивості flex-елементів**

**flex-grow** – коефіцієнт збільшення елемента відносно вільного простору. За замовчуванням 0 – елемент не розтягується:

*файл css*

.item { flex-grow: 1; } /* займає весь вільний простір рівномірно */

.item-wide { flex-grow: 2; } /* займає вдвічі більше простору ніж flex-grow: 1 */

**flex-shrink** – коефіцієнт зменшення елемента при нестачі простору. За замовчуванням 1 – елемент може стискатись:

*файл css*

```
.item { flex-shrink: 0; } /* елемент не стискається */
.item { flex-shrink: 1; } /* елемент стискається пропорційно */
```

**flex-basis** – базовий розмір елемента до розподілу вільного простору. Аналог width для flex-елементів:

*файл css*

```
.item { flex-basis: 200px; } /* початкова ширина 200px */
.item { flex-basis: 33.33%; } /* третина контейнера */
.item { flex-basis: auto; }   /* за вмістом – за замовчуванням */
```

**flex** – скорочений запис flex-grow, flex-shrink, flex-basis. Рекомендований спосіб запису:

*файл css*

```
.item { flex: 1; }        /* flex: 1 1 0 */
.item { flex: auto; }     /* flex: 1 1 auto */
.item { flex: none; }     /* flex: 0 0 auto */
.item { flex: 1 1 200px; }
```

**order** – порядок відображення елемента. За замовчуванням 0. Менше значення – раніше відображається:

*файл css*

```
.item-first { order: -1; } /* відображається першим */
.item-last  { order: 1; }  /* відображається останнім */
```

**align-self** – індивідуальне вирівнювання конкретного елемента по поперечній осі. Перекриває align-items контейнера:

*файл css*

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

## 5. Практичне застосування flexbox

**Горизонтальне і вертикальне центрування:**

*файл css*

```
.centered {
  display: flex;
  justify-content: center;
  align-items: center;
```

```
  min-height: 100vh;
}
```

**Навігаційна панель:**

*файл css*

```
.nav {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 0 24px;
  height: 64px;
}
.nav__logo { font-weight: bold; }
.nav__links {
  display: flex;
  gap: 24px;
  list-style: none;
}
```

**Картки рівної висоти:**

*файл css*

```
.cards {
  display: flex;
  flex-wrap: wrap;
  gap: 24px;
}
.card {
  flex: 1 1 280px;
  display: flex;
  flex-direction: column;
}
.card__content { flex-grow: 1; }  /* розтягує вміст */
.card__footer { margin-top: auto; } /* притискає підвал до низу */
```

**Макет із бічною панеллю:**

*файл css*

```
.layout {
  display: flex;
  gap: 24px;
  align-items: flex-start;
```

```
}
.sidebar { flex: 0 0 260px; }  /* фіксована ширина, не стискається */
.main    { flex: 1; }          /* займає залишковий простір */
```

**Форма з полями в рядок:**

*файл css*

```
.form-row {
  display: flex;
  gap: 16px;
}
.form-row .form-group {
  flex: 1;
}
```

**Зведена таблиця властивостей:**

| Властивість | Застосовується до | Призначення |
|---|---|---|
| flex-direction | Контейнер | Напрямок основної осі |
| flex-wrap | Контейнер | Перенесення елементів |
| gap | Контейнер | Відступи між елементами |
| justify-content | Контейнер | Вирівнювання по основній осі |
| align-items | Контейнер | Вирівнювання по поперечній осі |
| align-content | Контейнер | Вирівнювання рядків |
| flex-grow | Елемент | Коефіцієнт розтягування |
| flex-shrink | Елемент | Коефіцієнт стискання |
| flex-basis | Елемент | Базовий розмір |
| order | Елемент | Порядок відображення |
| align-self | Елемент | Індивідуальне вирівнювання |

### *Контрольні питання*

1. Що таке flexbox і які задачі верстки він вирішує краще за попередні підходи?
2. Як оголосити flex-контейнер? Чим display: flex відрізняється від display: inline-flex?
3. Що таке основна та поперечна осі flexbox? Як вони розташовані за замовчуванням?
4. Як властивість flex-direction впливає на напрямок основної осі? Назвіть усі її значення.
5. Що відбувається з flex-елементами при переповненні контейнера за замовчуванням і як це змінити?
6. Для чого використовується властивість gap і чим вона зручніша за margin?
7. За вирівнювання вздовж якої осі відповідає justify-content? Назвіть його основні значення.
8. Яка різниця між justify-content: space-between і justify-content: space-evenly?
9. За вирівнювання вздовж якої осі відповідає align-items? Назвіть його основні значення.
10. Яка різниця між align-items і align-content? За яких умов діє align-content?
11. Що таке flex-grow? Що означає значення 0 і що означає значення 1?
12. Для чого використовується flex-shrink: 0? Наведіть практичний приклад.
13. Яка різниця між flex-basis і width?
14. Що означає скорочений запис flex: 1? Які значення він задає?
15. Як за допомогою властивості order змінити порядок відображення елементів?
16. Чим align-self відрізняється від align-items і коли його застосовують?
17. Як за допомогою flexbox відцентрувати елемент по горизонталі і вертикалі одночасно?
18. Як реалізувати макет із фіксованою бічною панеллю і гнучкою основною зоною?
19. Як зробити картки рівної висоти з кнопкою, притиснутою до низу кожної картки?
20. Коли доцільно використовувати flexbox, а коли – CSS Grid?

# Тема 6. Технологія CSS Grid

**Мета:** сформувати розуміння концепції grid-компонування, навчити оголошувати grid-контейнер, визначати колонки та рядки сітки, розміщувати елементи у сітці вручну та через іменовані зони, застосовувати вирівнювання, будувати адаптивні сітки та поєднувати Grid і Flexbox в одному макеті.

**План**

1. Поняття CSS Grid та його роль у верстці.
2. Оголошення grid-контейнера. Колонки та рядки.
3. Розміщення елементів у сітці.
4. Іменовані зони.
5. Вирівнювання елементів.
6. Адаптивні сітки.
7. Поєднання Grid і Flexbox. Порівняння підходів.

## 1. Поняття CSS Grid та його роль у верстці

**CSS Grid** – модуль CSS для двовимірного компонування: елементи розміщуються одночасно по рядках і колонках. На відміну від flexbox, який працює в одному напрямку, Grid керує розміщенням по обох осях одночасно.

Grid найкраще підходить для:

- побудови загального макету сторінки;
- складних сіток з елементами різного розміру;
- розміщення елементів у точно визначених позиціях;
- симетричних карткових сіток.

## 2. Оголошення grid-контейнера. Колонки та рядки

Для активації Grid задається display: grid або display: inline-grid:

*файл css*

```
.container {
  display: grid;
}
```

**grid-template-columns** – визначає кількість і ширину колонок:

*файл css*

```
.container {
  grid-template-columns: 200px 200px 200px; /* три колонки по 200px */
```

```
  grid-template-columns: 1fr 1fr 1fr;       /* три рівні колонки */
  grid-template-columns: 200px 1fr;         /* фіксована + гнучка */
  grid-template-columns: 30% 70%;           /* у відсотках */
}
```

**grid-template-rows** – визначає кількість і висоту рядків:

*файл css*

```
.container {
  grid-template-rows: 80px 1fr 60px; /* шапка, вміст, підвал */
}
```

**Одиниця fr** – дробова частина вільного простору контейнера після відрахування фіксованих розмірів:

*файл css*

```
.container {
  grid-template-columns: 250px 1fr 2fr;
  /* 250px фіксовано, решта ділиться: 1/3 і 2/3 */
}
```

**Функція repeat()** – скорочений запис для повторюваних значень:

*файл css*

```
.container {
  grid-template-columns: repeat(3, 1fr);       /* три рівні колонки */
  grid-template-columns: repeat(4, 200px);     /* чотири по 200px */
  grid-template-columns: repeat(3, 1fr 2fr);   /* чергування */
}
```

**gap** – відступи між колонками і рядками:

*файл css*

```
.container {
  gap: 24px;              /* однаковий відступ */
  gap: 16px 24px;         /* рядковий і стовпцевий */
  row-gap: 16px;
  column-gap: 24px;
}
```

**Неявна сітка** – якщо елементів більше ніж визначено в шаблоні, браузер автоматично створює нові рядки. Їх розміром керує grid-auto-rows:

*файл css*

```
.container {
  grid-template-columns: repeat(3, 1fr);
  grid-auto-rows: 200px;         /* висота автоматичних рядків */
```

```
  grid-auto-rows: minmax(150px, auto); /* мінімум і максимум */
}
```

### 3. Розміщення елементів у сітці

За замовчуванням елементи розміщуються автоматично – по одному у кожну клітинку. Для точного розміщення використовуються властивості grid-column і grid-row.

**Лінії сітки** – grid ділить простір лініями. Для сітки з трьох колонок існує чотири вертикальні лінії (1, 2, 3, 4):

```
1   2   3   4
│ col1 │ col2 │ col3 │
```

**grid-column** – розміщення елемента по горизонталі через номери ліній:

*файл css*

```
.item {
  grid-column: 1 / 3;  /* від лінії 1 до лінії 3 – займає 2 колонки */
  grid-column: 2 / 4;  /* від лінії 2 до лінії 4 */
  grid-column: 1 / -1; /* від першої до останньої лінії – вся ширина */
}
```

**grid-row** – розміщення елемента по вертикалі:

*файл css*

```
.item {
  grid-row: 1 / 3; /* займає 2 рядки */
  grid-row: 2 / 4;
}
```

**Скорочення через span** – кількість клітинок замість номерів ліній:

*файл css*

```
.item {
  grid-column: span 2; /* займає 2 колонки від поточної позиції */
  grid-row: span 3;    /* займає 3 рядки */
}
```

**grid-area** – скорочений запис: grid-row-start / grid-column-start / grid-row-end / grid-column-end:

*файл css*

```
.item {
  grid-area: 1 / 1 / 3 / 4; /* рядок 1-3, колонка 1-4 */
}
```

### 4. Іменовані зони

Іменовані зони – найнаочніший спосіб побудови макету сторінки. Кожній зоні дається ім'я, а макет описується текстовою схемою.

**grid-template-areas** – визначає схему макету. Кожен рядок у лапках – рядок сітки. Крапка (.) позначає порожню клітинку:

*файл css*

```
.container {
  display: grid;
  grid-template-columns: 250px 1fr;
  grid-template-rows: 80px 1fr 60px;
  grid-template-areas:
    "header  header"
    "sidebar main"
    "footer  footer";
  gap: 16px;
  min-height: 100vh;
}
```

Елементам призначаються зони через grid-area:

*файл css*

```
.header  { grid-area: header; }
.sidebar { grid-area: sidebar; }
.main    { grid-area: main; }
.footer  { grid-area: footer; }
```

html

```
<div class="container">
  <header class="header">Шапка</header>
  <aside class="sidebar">Бічна панель</aside>
  <main class="main">Основний вміст</main>
  <footer class="footer">Підвал</footer>
</div>
```

Приклад з порожньою клітинкою:

*файл css*

```
grid-template-areas:
  "header header header"
  "sidebar main  ."
  "footer footer footer";
```

### 5. Вирівнювання елементів

Grid має два рівні вирівнювання: вирівнювання самих елементів у клітинках і вирівнювання сітки у контейнері.

**Вирівнювання елементів у клітинках:**

*файл css*

```
/* По горизонталі (вісь рядка) */
.container { justify-items: start; }   /* до лівого краю клітинки */
.container { justify-items: end; }     /* до правого краю */
.container { justify-items: center; }  /* по центру */
.container { justify-items: stretch; } /* розтягнути – за замовчуванням */
/* По вертикалі (вісь колонки) */
.container { align-items: start; }
.container { align-items: end; }
.container { align-items: center; }
.container { align-items: stretch; }
```

**Індивідуальне вирівнювання окремого елемента:**

*файл css*

```
.item { justify-self: center; }
.item { align-self: end; }
```

**Вирівнювання сітки у контейнері** – якщо сітка менша за контейнер:

*файл css*

```
.container { justify-content: center; }
.container { justify-content: space-between; }
.container { align-content: center; }
```

### 6. Адаптивні сітки

Grid має вбудовані інструменти для побудови адаптивних сіток без медіазапитів.

**auto-fit** – створює стільки колонок, скільки вміщується, і розтягує їх на всю ширину:

*файл css*

```
.container {
  grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
}
```

**auto-fill** – також заповнює рядок колонками, але залишає порожні клітинки якщо елементів не вистачає:

*файл css*

```
.container {
  grid-template-columns: repeat(auto-fill, minmax(250px, 1fr));
```

```
}
```

**Функція minmax()** – задає мінімальний і максимальний розмір колонки або рядка:

*файл css*

```
.container {
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
  /* кожна колонка: мінімум 200px, максимум – рівна частка простору */
}
.container {
  grid-auto-rows: minmax(150px, auto);
  /* рядок: мінімум 150px, максимум – за вмістом */
}
```

Поєднання repeat(auto-fit, minmax()) – найпоширеніший підхід до адаптивних карткових сіток без медіазапитів:

*файл css*

```
.cards {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(280px, 1fr));
  gap: 24px;
}
```

### 7. Поєднання Grid і Flexbox. Порівняння підходів

**Поєднання в одному макеті** – Grid і Flexbox не конкурують, а доповнюють одне одного. Типовий підхід: Grid будує загальний макет сторінки, Flexbox – вирівнює вміст всередині компонентів:

*файл css*

```
/* Grid – загальний макет */
.page {
  display: grid;
  grid-template-areas:
    "header"
    "main"
    "footer";
  grid-template-rows: auto 1fr auto;
  min-height: 100vh;
}
/* Flexbox – всередині шапки */
.header {
```

```
  grid-area: header;
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 0 24px;
}
/* Flexbox – картка всередині grid-сітки */
.card {
  display: flex;
  flex-direction: column;
}
.card__content { flex: 1; }
```

**Порівняння підходів:**

| Критерій | Flexbox | Grid |
|---|---|---|
| Вимірність | Одновимірний (рядок або стовпець) | Двовимірний (рядки і стовпці) |
| Керування | Від вмісту до контейнера | Від контейнера до вмісту |
| Типові задачі | Навігація, кнопки, форми, картки | Макет сторінки, складні сітки |
| Адаптивність | Через flex-wrap | Через auto-fit/minmax |
| Точне розміщення | Обмежено | Повний контроль |

**Правило вибору:**

- якщо розміщення одновимірне (елементи в рядок або стовпець) – **Flexbox**
- якщо потрібен контроль по рядках і колонках одночасно – **Grid**
- для макету сторінки в цілому – **Grid**
- для вмісту всередині компонентів – **Flexbox**

## Контрольні запитання

1. Що таке CSS Grid і чим він принципово відрізняється від Flexbox?

2. Як оголосити grid-контейнер? Чим display: grid відрізняється від display: inline-grid?
3. Для чого використовуються властивості grid-template-columns і grid-template-rows?
4. Що таке одиниця fr і як вона розраховується?
5. Що робить функція repeat() і які аргументи вона приймає?
6. Що таке неявна сітка і як керувати розміром її рядків?
7. Що таке лінії сітки? Скільки вертикальних ліній має сітка з чотирьох колонок?
8. Як розмістити елемент так, щоб він займав дві колонки? Наведіть два способи запису.
9. Що означає запис grid-column: 1 / -1?
10. Що таке іменовані зони і як вони оголошуються?
11. Як призначити елементу іменовану зону сітки?
12. Що позначає крапка (.) у значенні grid-template-areas?
13. Яка різниця між justify-items і justify-content у Grid?
14. Для чого використовуються justify-self і align-self? Чим вони відрізняються від justify-items і align-items?
15. Яка різниця між auto-fit і auto-fill у функції repeat()?
16. Що робить функція minmax() і як вона використовується разом з auto-fit?
17. Як побудувати адаптивну карткову сітку без медіазапитів?
18. Як поєднуються Grid і Flexbox в одному макеті? Наведіть практичний приклад.
19. Коли доцільно використовувати Grid, а коли Flexbox?
20. Опишіть структуру типового макету сторінки з шапкою, бічною панеллю, основним вмістом і підвалом за допомогою grid-template-areas.

# Тема 7. Адаптивний дизайн

**Мета:** сформувати розуміння принципів адаптивного дизайну, навчити використовувати метатег viewport, писати медіазапити, обирати між підходами mobile-first і desktop-first, застосовувати відносні одиниці виміру, адаптивні зображення та типографіку, поєднувати медіазапити з Flexbox і Grid для побудови повністю адаптивних макетів.

**План**

1. Поняття адаптивного дизайну.
2. Метатег viewport.
3. Медіазапити: синтаксис та точки зламу.
4. Підходи mobile-first і desktop-first.
5. Відносні одиниці виміру в контексті адаптивності.
6. Адаптивні зображення та типографіка.
7. Практичне застосування медіазапитів з Flexbox і Grid.

## 1. Поняття адаптивного дизайну

**Адаптивний дизайн (Responsive Web Design)** – підхід до створення вебсайтів, за якого сторінка коректно відображається і зручно використовується на будь-якому пристрої – смартфоні, планшеті, ноутбуці, десктопі – незалежно від розміру екрана.

Основні принципи адаптивного дизайну:

- гнучкі сітки на основі відносних одиниць
- гнучкі зображення, що масштабуються разом з контейнером
- медіазапити для зміни стилів залежно від характеристик пристрою

**Чому це важливо:** більше половини вебтрафіку у світі припадає на мобільні пристрої. Google враховує мобільну версію сайту як основну при індексації (Mobile First Indexing).

## 2. Метатег viewport

Без метатегу viewport мобільний браузер відображає сторінку так, ніби екран має ширину десктопа – зменшує масштаб і робить текст нечитабельним.

*файл html*

<meta name="viewport" content="width=device-width, initial-scale=1.0">

**Атрибути:**

width=device-width – ширина області перегляду дорівнює фізичній ширині екрана пристрою.

initial-scale=1.0 – початковий масштаб сторінки – 1:1, без збільшення чи зменшення.

Додаткові значення, які не рекомендується використовувати:

*файл html*

<!-- Забороняє масштабування – погіршує доступність -->

<meta name="viewport" content="width=device-width, initial-scale=1.0, user-scalable=no">

Метатег viewport є обов'язковим елементом будь-якої адаптивної сторінки і розміщується у секції <head>.

### 3. Медіазапити: синтаксис та точки зламу

**Медіазапит (@media)** – правило CSS, що застосовує стилі лише за умови відповідності характеристик пристрою заданим умовам.

**Базовий синтаксис:**

*файл css*

```
@media тип_медіа and (умова) {
  /* CSS-правила */
}
```

**Типи медіа:**

*файл css*

```
@media screen { }  /* екрани – за замовчуванням */
@media print { }   /* друк */
@media all { }     /* всі пристрої */
```

**Умови (media features):**

*файл css*

```
/* Ширина вікна браузера */
@media (min-width: 768px) { }   /* від 768px і ширше */
@media (max-width: 767px) { }   /* до 767px включно */
@media (width: 375px) { }       /* точно 375px */
/* Висота вікна */
@media (min-height: 600px) { }
/* Орієнтація пристрою */
@media (orientation: landscape) { } /* горизонтальна */
@media (orientation: portrait) { }  /* вертикальна */
/* Роздільна здатність */
@media (min-resolution: 2dppx) { }  /* Retina-дисплеї */
```

**Логічні оператори:**

*файл css*

```
/* AND – обидві умови виконуються */
@media screen and (min-width: 768px) and (max-width: 1199px) { }
/* OR – хоча б одна умова (кома) */
@media (max-width: 480px), (orientation: portrait) { }
/* NOT – заперечення */
@media not print { }
/* Діапазон (сучасний синтаксис CSS Media Queries Level 4) */
@media (768px <= width <= 1199px) { }
```

**Точки зламу (breakpoints)** – значення ширини екрана, при яких змінюється макет сторінки. Типові точки зламу:

| Назва | Ширина | Пристрої |
|---|---|---|
| xs | до 576px | Смартфони |
| sm | 576px–767px | Великі смартфони |
| md | 768px–991px | Планшети |
| lg | 992px–1199px | Ноутбуки |
| xl | 1200px і більше | Десктопи |

Точки зламу обираються не під конкретні пристрої, а під вміст – там де макет починає «ламатись».

## 4. Підходи mobile-first і desktop-first

**Mobile-first** – базові стилі пишуться для мобільних пристроїв, медіазапити з min-width додають стилі для ширших екранів:

*файл css*

```
/* Базові стилі – мобільний */
.container {
  padding: 16px;
}
.cards {
  display: flex;
  flex-direction: column;
```

```
  gap: 16px;
}
/* Планшет */
@media (min-width: 768px) {
  .container { padding: 24px; }
  .cards { flex-direction: row; flex-wrap: wrap; }
}
/* Десктоп */
@media (min-width: 1200px) {
  .container { max-width: 1140px; margin: 0 auto; }
}
```

**Desktop-first** – базові стилі пишуться для десктопа, медіазапити з max-width адаптують для вужчих екранів:

*файл css*

```
/* Базові стилі – десктоп */
.cards {
  display: flex;
  gap: 24px;
}
/* Планшет */
@media (max-width: 991px) {
  .cards { flex-wrap: wrap; }
}
/* Мобільний */
@media (max-width: 767px) {
  .cards { flex-direction: column; }
}
```

**Порівняння підходів:**

| **Критерій** | **Mobile-first** | **Desktop-first** |
|---|---|---|
| Пріоритет | Мобільні пристрої | Десктопні пристрої |
| Оператор | min-width | max-width |
| Рекомендований | Так | Ні |

| | | |
|---|---|---|
| Причина | Більше мобільного трафіку, кращий SEO | Використовується для legacy-проєктів |

Mobile-first є рекомендованим стандартом сучасної веброзробки.

**5. Відносні одиниці виміру в контексті адаптивності**

Відносні одиниці – основа адаптивної верстки. Вони масштабуються залежно від контексту, на відміну від фіксованих px.

**em** – відносно розміру шрифту батьківського елемента. Зручний для відступів, що мають залежати від розміру тексту:

*файл css*

```
.btn {
  font-size: 16px;
  padding: 0.75em 1.5em; /* 12px і 24px відносно 16px */
}
```

**rem** – відносно розміру шрифту кореневого елемента <html>. Передбачуваніший за em, рекомендований для розмірів шрифтів і відступів:

*файл css*

```
html { font-size: 16px; }

h1 { font-size: 2rem; }    /* 32px */
p  { font-size: 1rem; }    /* 16px */
.section { padding: 2rem; } /* 32px */
```

Масштабування через зміну кореневого шрифту:

*файл css*

```
html { font-size: 16px; }

@media (max-width: 767px) {
  html { font-size: 14px; }
  /* всі rem-значення зменшаться пропорційно */
}
```

**%** – відносно розміру батьківського елемента. Використовується для ширини контейнерів:

*файл css*

```
.container { width: 90%; max-width: 1200px; margin: 0 auto; }
.column    { width: 50%; }
```

**vw та vh** – відносно розмірів вікна браузера:

*файл css*

```
.hero    { height: 100vh; }   /* повна висота екрана */
.banner  { width: 100vw; }    /* повна ширина екрана */
```

**Функція clamp()** – задає значення між мінімумом і максимумом з гнучким середнім:

*файл css*

```
h1 { font-size: clamp(1.5rem, 4vw, 3rem); }
/* мінімум 1.5rem, максимум 3rem, між ними – 4vw */

.container { width: clamp(320px, 90%, 1200px); }
```

## 6. Адаптивні зображення та типографіка

**Адаптивні зображення:**

Базовий підхід – зображення не виходить за межі контейнера:

*файл css*

```
img {
  max-width: 100%;
  height: auto; /* пропорційне масштабування */
}
```

**Елемент <picture>** – різні зображення для різних умов:

*файл html*

```
<picture>
  <source media="(min-width: 1200px)" srcset="image-large.jpg">
  <source media="(min-width: 768px)"  srcset="image-medium.jpg">
  <img src="image-small.jpg" alt="Опис зображення">
</picture>
```

**Атрибут srcset** – набір зображень різної роздільної здатності, браузер обирає найоптимальніше:

*файл html*

```
<img
  src="image.jpg"
  srcset="image-400.jpg 400w, image-800.jpg 800w, image-1200.jpg 1200w"
  sizes="(max-width: 767px) 100vw, (max-width: 1199px) 50vw, 33vw"
  alt="Опис">
```

**Адаптивна типографіка:**

Фіксовані розміри шрифтів через медіазапити:

*файл css*

```
h1 { font-size: 2rem; }

@media (min-width: 768px)  { h1 { font-size: 2.5rem; } }
@media (min-width: 1200px) { h1 { font-size: 3rem; } }
```

Плавна типографіка через clamp() без медіазапитів:

*файл css*

```
h1 { font-size: clamp(1.75rem, 3vw + 1rem, 3rem); }
p  { font-size: clamp(1rem, 1.5vw, 1.25rem); }
```

## 7. Практичне застосування медіазапитів з Flexbox і Grid

**Адаптивна навігація:**

*файл css*

```
.nav {
  display: flex;
  flex-direction: column;
  gap: 8px;
}
@media (min-width: 768px) {
  .nav {
    flex-direction: row;
    justify-content: space-between;
    align-items: center;
  }
}
```

**Адаптивний макет сторінки через Grid:**

*файл css*

```
.layout {
  display: grid;
  grid-template-areas:
    "header"
    "main"
    "sidebar"
    "footer";
  gap: 16px;
}
@media (min-width: 768px) {
  .layout {
    grid-template-columns: 1fr 260px;
```

```
    grid-template-areas:
      "header  header"
      "main    sidebar"
      "footer  footer";
  }
}
@media (min-width: 1200px) {
  .layout {
    grid-template-columns: 220px 1fr 260px;
    grid-template-areas:
      "header  header  header"
      "nav     main    sidebar"
      "footer  footer  footer";
    max-width: 1400px;
    margin: 0 auto;
  }
}
.header  { grid-area: header; }
.main    { grid-area: main; }
.sidebar { grid-area: sidebar; }
.footer  { grid-area: footer; }
```

**Адаптивна карткова сітка:**

*файл css*

```
/* Mobile-first: одна колонка */
.cards {
  display: grid;
  gap: 16px;
}
/* Планшет: дві колонки */
@media (min-width: 768px) {
  .cards {
    grid-template-columns: repeat(2, 1fr);
  }
}
/* Десктоп: три колонки */
@media (min-width: 1200px) {
  .cards {
```

```
    grid-template-columns: repeat(3, 1fr);
    gap: 24px;
  }
}
```

Або без медіазапитів через auto-fit:

*файл css*

```
.cards {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(280px, 1fr));
  gap: 24px;
}
```

## Контрольні запитання

1. Що таке адаптивний дизайн і які три принципи лежать в його основі?
2. Для чого призначений метатег viewport і що відбувається якщо його не вказати?
3. Що означають атрибути width=device-width і initial-scale=1.0?
4. Що таке медіазапит і яку задачу він вирішує?
5. Запишіть синтаксис медіазапиту, що застосовує стилі для екранів шириною від 768px.
6. Які логічні оператори використовуються у медіазапитах і що кожен з них означає?
7. Що таке точки зламу і за яким принципом вони обираються?
8. У чому різниця між підходами mobile-first і desktop-first?
9. Який оператор використовується у медіазапитах при підході mobile-first, а який – при desktop-first?
10. Чому mobile-first вважається рекомендованим стандартом сучасної веброзробки?
11. Яка різниця між одиницями em і rem? Коли доцільно використовувати кожну?
12. Як зміна розміру шрифту кореневого елемента <html> впливає на всі rem-значення на сторінці?
13. Що робить функція clamp() і які аргументи вона приймає?
14. Як зробити зображення адаптивним за допомогою CSS?
15. Для чого використовується елемент <picture> і чим він відрізняється від атрибута srcset?
16. Назвіть два способи реалізації адаптивної типографіки.
17. Як побудувати адаптивну карткову сітку через Grid без медіазапитів?

18. Як змінити порядок зон макету через grid-template-areas для мобільного і десктопного розміщення?
19. Як адаптувати навігаційне меню з вертикального на горизонтальне за допомогою Flexbox і медіазапитів?
20. Які одиниці виміру найкраще підходять для ширини контейнерів, розмірів шрифтів і висоти секцій відповідно?

# Модуль ІІІ. Скриптова мова програмування JavaScript

## Тема 8. Синтаксис та базові конструкції JS.

**Мета:** сформувати розуміння природи JavaScript як мови програмування, навчити підключати скрипти до HTML-сторінки, оголошувати змінні, працювати з типами даних та операторами, виконувати перетворення типів і читати структуру JS-коду.

**План**

1. Загальні поняття JS. Стандарт ECMAScript.
2. Підключення скрипту до HTML-сторінки.
3. Введення та виведення даних.
4. Структура коду, інструкції та коментарі.
5. Оголошення змінних: var, let, const.
6. Типи даних та оператор typeof.
7. Оператори JS.
8. Перетворення типів. Пріоритетність операторів.

### 1. Загальні поняття JS. Стандарт ECMAScript

**JavaScript** – скриптова мова програмування, що виконується у браузері та забезпечує інтерактивність вебсторінок. На відміну від HTML і CSS, JavaScript є повноцінною мовою програмування з умовами, циклами, функціями та об'єктами.

JavaScript виконується у двох середовищах: браузер (клієнтська сторона) та Node.js (серверна сторона). У цьому курсі розглядається браузерний JavaScript.

**ECMAScript** – офіційний стандарт мови JavaScript, що підтримується організацією ECMA International. Версії стандарту:

- ES5 (2009) – широка підтримка, основа сумісного коду
- ES6 / ES2015 – ключове оновлення: let, const, стрілкові функції, класи, модулі
- ES2017+ – async/await, нові методи масивів і об'єктів
- ESNext – актуальна версія, що постійно оновлюється

Сучасний JS-код пишеться за стандартом ES6+.

### 2. Підключення скрипту до HTML-сторінки

Існує три способи підключення JavaScript до HTML-документа.

**Зовнішній файл** – рекомендований спосіб. Скрипт виноситься в окремий файл з розширенням .js:

*файл html*

```
<!-- Підключення перед закриваючим тегом </body> -->
<body>
  <!-- вміст сторінки -->
  <script src="script.js"></script>
</body>
```

Розміщення перед </body> гарантує, що HTML завантажиться раніше, ніж скрипт почне виконуватись.

**Атрибути завантаження скрипту:**

*файл html*

```
<!-- defer: скрипт завантажується паралельно, виконується після парсингу HTML -->
<script src="script.js" defer></script>
<!-- async: скрипт завантажується і виконується паралельно, порядок не гарантований -->
<script src="script.js" async></script>
```

defer – рекомендований атрибут при підключенні у <head>.

**Внутрішній скрипт** – JS-код розміщується безпосередньо в HTML у тегу <script>:

*файл html*

```
<script>
  console.log('Привіт, світе!');
</script>
```

**Інлайн-обробник** – JS-код у атрибуті HTML-елемента. Не рекомендується:

*файл html*

```
<button onclick="alert('Натиснуто')">Клік</button>
```

### 3. Введення та виведення даних

**Виведення у консоль браузера** – основний інструмент розробника для перевірки коду:

*файл javascript*

```
console.log('Звичайне повідомлення');
console.warn('Попередження');
console.error('Помилка');
console.table([1, 2, 3]); /* виведення у вигляді таблиці */
```

Консоль браузера відкривається клавішею F12 → вкладка Console.

**Діалогові вікна браузера:**

javascript

```
/* Виведення повідомлення */
alert('Привіт!');
/* Введення даних користувачем – повертає рядок або null */
let name = prompt('Як вас звати?');
let name = prompt('Як вас звати?', 'Іван'); /* друге значення – за замовчуванням */
/* Підтвердження – повертає true або false */
let answer = confirm('Ви впевнені?');
```

**Виведення на сторінку:**

*файл javascript*

```
/* Запис у HTML-елемент */
document.getElementById('result').textContent = 'Результат';
document.getElementById('result').innerHTML = '<b>Результат</b>';
/* Запис у документ – лише під час завантаження */
document.write('Текст');
```

## 4. Структура коду, інструкції та коментарі

**Інструкція** – окрема команда, що виконує певну дію. Інструкції відокремлюються крапкою з комою:

*файл javascript*

```
let x = 5;
let y = 10;
console.log(x + y);
```

Крапка з комою в JS формально необов'язкова – діє механізм автоматичного вставлення (ASI). Однак явне використання крапки з комою є рекомендованою практикою.

**Блок коду** – кілька інструкцій, об'єднаних фігурними дужками:

*файл javascript*

```
{
  let a = 1;
  let b = 2;
  console.log(a + b);
}
```

**Суворий режим 'use strict'** – вмикає суворий режим виконання, що забороняє потенційно небезпечні конструкції. Розміщується першим рядком файлу або функції:

*файл javascript*

```
'use strict';
x = 5; /* помилка – змінна не оголошена */
```

**Коментарі:**

*файл javascript*

```
// Однорядковий коментар

/*
  Багаторядковий коментар.
  Використовується для пояснення блоків коду.
*/
```

**Регістрочутливість** – JS розрізняє великі і малі літери:

*файл javascript*

```
let name = 'Іван';
let Name = 'Петро'; /* інша змінна */
```

## 5. Оголошення змінних: var, let, const

**var** – застарілий спосіб оголошення змінної. Має функційну область видимості та підлягає hoisting (підняттю). У сучасному коді не використовується:

*файл javascript*

```
var x = 10;
```

**let** – сучасне оголошення змінної з блоковою областю видимості. Значення можна змінювати:

*файл javascript*

```
let age = 20;
age = 21; /* дозволено */
```

**const** – оголошення константи з блоковою областю видимості. Значення не можна перепризначити після оголошення. Рекомендується використовувати за замовчуванням:

*файл javascript*

```
const PI = 3.14159;
PI = 3; /* помилка */
const user = { name: 'Іван' };
```

user.name = 'Петро'; /* *дозволено – змінюється вміст об'єкта, не сам об'єкт* */

**Hoisting (підняття)** – механізм, за яким оголошення змінних і функцій переміщуються на початок своєї області видимості під час компіляції. var підіймається з ініціалізацією undefined, let і const – підіймаються, але залишаються у тимчасовій мертвій зоні до рядка оголошення:

*файл javascript*

```
console.log(a); /* undefined – var підіймається */
var a = 5;
console.log(b); /* помилка – let у мертвій зоні */
let b = 5;
```

**Правила іменування змінних:**

- можуть містити літери, цифри, _ та $;
- не можуть починатись з цифри;
- не можуть збігатись з ключовими словами (let, const, if тощо);
- регістрочутливі;
- угода: camelCase для змінних і функцій, UPPER_CASE для констант.

*файл javascript*

```
let userName = 'Іван';
let userAge = 25;
const MAX_SIZE = 100;
```

## 6. Типи даних та оператор typeof

JavaScript – мова з динамічною типізацією: тип змінної визначається її значенням і може змінюватись.

**Примітивні типи:**

*файл javascript*

```
let num = 42;            // number
let pi = 3.14;           // number
let name = 'Іван';       // string
let isActive = true;     // boolean
let nothing = null;      // null
let notDefined;          // undefined
let id = Symbol('id');   // symbol
let bigNum = 9007199254740991n; // bigint
```

**Посилальні типи:**

*файл javascript*

```
let arr = [1, 2, 3];          // object (масив)
```

```
let obj = { name: 'Іван' };      // object
let fn = function() {};          // function
```

**Оператор typeof** – повертає рядок з назвою типу:

*файл javascript*

```
typeof 42          // 'number'
typeof 'hello'     // 'string'
typeof true        // 'boolean'
typeof undefined   // 'undefined'
typeof null        // 'object' – відома особливість JS
typeof []          // 'object'
typeof {}          // 'object'
typeof function(){} // 'function'
```

**Особливі числові значення:**

*файл javascript*

```
let inf = Infinity;   // нескінченність
let negInf = -Infinity;
let notNum = NaN;          // Not a Number – результат некоректної математичної операції
console.log(0 / 0);   // NaN
console.log(1 / 0);   // Infinity
isNaN(NaN);           // true
```

## 7. Оператори JS

**Арифметичні оператори:**

*файл javascript*

```
let a = 10, b = 3;

a + b   // 13 – додавання
a - b   // 7  – віднімання
a * b   // 30 – множення
a / b   // 3.333... – ділення
a % b   // 1  – остача від ділення
a ** b  // 1000 – піднесення до степеня

/* Інкремент та декремент */
let x = 5;
x++;    // постфіксний: повертає 5, потім x = 6
++x;    // префіксний: x = 7, повертає 7
```

```
x--;    // постфіксний декремент
--x;    // префіксний декремент
```

**Оператори присвоєння:**

*файл javascript*

```
let x = 10;
x += 5;   // x = x + 5 → 15
x -= 3;   // x = x - 3 → 12
x *= 2;   // x = x * 2 → 24
x /= 4;   // x = x / 4 → 6
x %= 4;   // x = x % 4 → 2
x **= 3;  // x = x ** 3 → 8
```

**Оператори порівняння:**

*файл javascript*

```
5 == '5'   // true  – нестрога рівність (з приведенням типів)
5 === '5'  // false – сувора рівність (без приведення типів)
5 != '5'   // false – нестрога нерівність
5 !== '5'  // true  – сувора нерівність
5 > 3      // true
5 < 3      // false
5 >= 5     // true
5 <= 4     // false
```

Рекомендується завжди використовувати === і !==.

**Логічні оператори:**

*файл javascript*

```
true && false  // false – AND: обидва мають бути true
true || false  // true  – OR: хоча б один true
!true          // false – NOT: заперечення

/* Скорочене обчислення */
let name = '' || 'Гість';    // 'Гість' – повертає перше truthy
let user = null && 'Іван';   // null    – повертає перше falsy
```

**Тернарний оператор:**

*файл javascript*

```
let age = 20;
let status = age >= 18 ? 'дорослий' : 'неповнолітній';
/* умова ? значення_якщо_true : значення_якщо_false */
```

**Оператори ?. та ??:**

*файл javascript*

```
/* Optional chaining – безпечний доступ до властивостей */
let user = null;
console.log(user?.name);   // undefined, а не помилка
/* Nullish coalescing – значення за замовчуванням для null/undefined */
let name = null ?? 'Гість'; // 'Гість'
let count = 0 ?? 10;        // 0 –  бо 0 не є null/undefined
```

## 8. Перетворення типів. Пріоритетність операторів

**Явне перетворення типів:**

*файл javascript*

```
/* До числа */
Number('42')      // 42
Number('3.14')    // 3.14
Number('')        // 0
Number('abc')     // NaN
Number(true)      // 1
Number(false)     // 0
Number(null)      // 0
Number(undefined) // NaN
parseInt('42px')  // 42
parseFloat('3.5em') // 3.5
/* До рядка */
String(42)        // '42'
String(true)      // 'true'
String(null)      // 'null'
(42).toString()   // '42'
/* До булевого */
Boolean(0)        // false
Boolean('')       // false
Boolean(null)     // false
Boolean(undefined)// false
Boolean(NaN)      // false
Boolean(1)        // true
Boolean('hello')  // true
Boolean([])       // true
```

**Truthy та falsy значення:**

*файл javascript*

```
/* Falsy – приводяться до false */
false, 0, '', null, undefined, NaN
/* Truthy – все інше, включаючи */
true, 1, 'hello', [], {}, function(){}
```

**Неявне перетворення типів (коерція):**

*файл javascript*

```
'5' + 3      // '53'  – число приводиться до рядка
'5' - 3      // 2     – рядок приводиться до числа
'5' * '3'    // 15    – обидва до числа
true + 1     // 2
false + 1    // 1
null + 1     // 1
undefined + 1 // NaN
```

**Пріоритетність операторів** – від вищого до нижчого:

| **Пріоритет** | **Оператори** |
|---|---|
| Найвищий | () дужки |
| Високий | ++, --, ! |
| Середній | ** |
| Середній | *, /, % |
| Середній | +, - |
| Нижчий | <, >, <=, >= |
| Нижчий | ==, ===, !=, !== |
| Нижчий | && |
| Нижчий | \|\|, ?? |
| Найнижчий | =, +=, -= тощо |

*файл javascript*

```
2 + 3 * 4      // 14 – множення першим
(2 + 3) * 4   // 20 – дужки змінюють порядок
true || false && false // true – && має вищий пріоритет ніж ||
```

**Контрольні запитання**

1. Що таке JavaScript і яку роль він відіграє у веброзробці поряд з HTML та CSS?
2. Що таке стандарт ECMAScript і яке значення мало оновлення ES6?
3. Назвіть три способи підключення JavaScript до HTML-сторінки. Який є рекомендованим?
4. Яка різниця між атрибутами defer і async? Коли застосовується кожен?
5. Які методи консолі браузера ви знаєте і чим вони відрізняються?
6. Яка різниця між alert(), prompt() і confirm()? Що повертає кожен?
7. Що таке інструкція і блок коду? Навіщо використовується крапка з комою?
8. Що робить директива 'use strict' і навіщо її використовувати?
9. Яка різниця між var, let і const? Яке оголошення рекомендується використовувати за замовчуванням?
10. Що таке hoisting? Як він поводиться з var і як з let?
11. Назвіть примітивні типи даних JavaScript. Чим вони відрізняються від посилальних?
12. Що повертає typeof null і чому це вважається особливістю мови?
13. Яка різниця між операторами == і ===? Який рекомендується використовувати?
14. Що таке скорочене обчислення логічних операторів && і ||?
15. Яка різниця між оператором ?? і оператором ||? Наведіть приклад де вони дають різний результат.
16. Що таке truthy і falsy значення? Назвіть усі falsy значення JavaScript.
17. Яка різниця між явним і неявним перетворенням типів? Наведіть приклад коерції.
18. Що поверне вираз '5' + 3 і що поверне '5' - 3? Поясніть чому.
19. Що поверне Number(null), Number(undefined) і Number('')?
20. Що означає пріоритетність операторів? Обчисліть результат виразу 2 + 3 * 4 === 20.

# Тема 9. Керуючі конструкції.

**Мета:** сформувати вміння керувати потоком виконання програми засобами JavaScript – використовувати умовні оператори, оператор вибору, логічні операції та цикли, застосовувати оператори break і continue, відлагоджувати код за допомогою покрокового виконання у консолі браузера.

**План**

1. Умовний оператор if.
2. Оператор вибору switch.
3. Логічні операції.
4. Цикли: for, while, do...while, for...of.
5. Оператори break і continue.
6. Покрокове виконання коду у консолі браузера.

## 1. Умовний оператор if

Умовний оператор if виконує блок коду якщо задана умова є істинною.

**Проста форма:**

*файл javascript*

```
if (умова) {
  // виконується якщо умова true
}
```

*файл javascript*

```
let age = 20;
if (age >= 18) {
  console.log('Доступ дозволено');
}
```

**Форма if...else:**

*файл javascript*

```
if (умова) {
  // виконується якщо умова true
} else {
  // виконується якщо умова false
}
```

*файл javascript*

```
let age = 15;
```

```
if (age >= 18) {
  console.log('Доступ дозволено');
} else {
  console.log('Доступ заборонено');
}
```

**Форма if...else if...else:**

*файл javascript*

```
let score = 75;
if (score >= 90) {
  console.log('Відмінно');
} else if (score >= 75) {
  console.log('Добре');
} else if (score >= 60) {
  console.log('Задовільно');
} else {
  console.log('Незадовільно');
}
```

**Умова та приведення до булевого типу:**

Умова в if приводиться до булевого типу. Будь-яке значення є або truthy або falsy:

*файл javascript*

```
let name = '';
if (name) {
  console.log('Ім\'я задано');
} else {
  console.log('Ім\'я порожнє'); // виконається – порожній рядок falsy
}
```

**Тернарний оператор як коротка форма if...else:**

*файл javascript*

```
let age = 20;
let status = age >= 18 ? 'дорослий' : 'неповнолітній';
```

Тернарний оператор доцільний для простих умов з одним значенням. Вкладені тернарні оператори погіршують читабельність – краще використовувати if...else.

## 2. Оператор вибору switch

**switch** порівнює значення виразу з кількома варіантами і виконує відповідний блок. Використовується замість довгих ланцюжків if...else if коли перевіряється одне значення:

*файл javascript*

```
switch (вираз) {
  case значення1:
    // код
    break;
  case значення2:
    // код
    break;
  default:
    // виконується якщо жоден case не підійшов
}
```

*файл javascript*

```
let day = 3;
switch (day) {
  case 1:
    console.log('Понеділок');
    break;
  case 2:
    console.log('Вівторок');
    break;
  case 3:
    console.log('Середа');
    break;
  default:
    console.log('Інший день');
}
```

**Оператор break** у switch завершує виконання блоку. Без break виконання «провалюється» до наступного case:

*файл javascript*

```
let month = 2;
switch (month) {
  case 12:
  case 1:
```

```
  case 2:
    console.log('Зима'); // спрацює для 12, 1 або 2
    break;
  case 3:
  case 4:
  case 5:
    console.log('Весна');
    break;
  default:
    console.log('Інший сезон');
}
```

**Порівняння switch та if:**

| Критерій | if...else if | switch |
|---|---|---|
| Тип порівняння | Будь-який вираз | Строга рівність (===) |
| Читабельність | Краще для складних умов | Краще для багатьох фіксованих значень |
| Провалювання | Відсутнє | Є без break |

### 3. Логічні операції

Логічні оператори повертають не обов'язково true або false – вони повертають одне з операндів значень.

**AND (&&)** – повертає перше falsy значення або останнє якщо всі truthy:

*файл javascript*

```
console.log(1 && 2 && 3);     // 3 – всі truthy, повертає останнє
console.log(1 && 0 && 3);     // 0 – перше falsy
console.log('' && 'привіт');  // '' – перше falsy
```

**OR (||)** – повертає перше truthy значення або останнє якщо всі falsy:

*файл javascript*

```
console.log(0 || '' || 'Гість'); // 'Гість' – перше truthy
console.log(1 || 'резерв');      // 1 – перше truthy
console.log(0 || null || '');    // '' – всі falsy, повертає останнє
```

Практичне застосування – значення за замовчуванням:

*файл javascript*

```
let userName = prompt('Ім\'я?') || 'Гість';
```

**NOT (!)** – заперечення, приводить значення до boolean і інвертує:

*файл javascript*

```
!true     // false
!false    // true
!0        // true
!''       // true
!'hello'  // false

/* Подвійне заперечення – явне приведення до boolean */
!!0       // false
!!'hello' // true
```

**Nullish coalescing (??)** – повертає праве значення лише якщо ліве є null або undefined. На відміну від || не реагує на 0 і '':

*файл javascript*

```
let count = 0;
console.log(count || 10);  // 10 – бо 0 є falsy
console.log(count ?? 10);  // 0  – бо 0 не є null/undefined
let name = '';
console.log(name || 'Гість');  // 'Гість'
console.log(name ?? 'Гість');  // '' – порожній рядок не є null/undefined
```

## 4. Цикли

Цикли дозволяють виконувати блок коду багаторазово.

**for** – цикл з лічильником. Використовується коли кількість ітерацій відома заздалегідь:

*файл javascript*

```
for (ініціалізація; умова; крок) {
  // тіло циклу
}
javascript
for (let i = 0; i < 5; i++) {
  console.log(i); // 0, 1, 2, 3, 4
}
/* Перебір масиву */
let fruits = ['яблуко', 'банан', 'апельсин'];
```

```
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
```

**while** – цикл з перевіркою умови на початку. Виконується поки умова true. Кількість ітерацій наперед невідома:

*файл javascript*

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

*файл javascript*

```
/* Практичний приклад */
let number = 1;
while (number <= 100) {
  number *= 2;
}
console.log(number); // перше число більше 100
```

**do...while** – тіло циклу виконується принаймні один раз, умова перевіряється після:

*файл javascript*

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

*файл javascript*

```
/* Практичний приклад – запит до введення коректного значення */
let input;
do {
  input = prompt('Введіть число від 1 до 10:');
} while (input < 1 || input > 10);
```

**for...of** – перебір ітерованих об'єктів: масивів, рядків. Сучасний і читабельніший спосіб перебору порівняно з for:

*файл javascript*

```
let fruits = ['яблуко', 'банан', 'апельсин'];
```

```
for (let fruit of fruits) {
  console.log(fruit);
}
/* Перебір рядка */
let word = 'привіт';
for (let char of word) {
  console.log(char); // п, р, и, в, і, т
}
```

**Порівняння циклів:**

| Цикл | Умова | Коли використовувати |
|---|---|---|
| for | На початку | Відома кількість ітерацій |
| while | На початку | Невідома кількість, умова перевіряється першою |
| do...while | Наприкінці | Тіло має виконатись хоча б раз |
| for...of | – | Перебір масиву або рядка |

## 5. Оператори break і continue

**break** – негайно завершує виконання циклу або switch:

*файл javascript*

```
for (let i = 0; i < 10; i++) {
  if (i === 5) break;
  console.log(i); // 0, 1, 2, 3, 4
}
```

*файл javascript*

```
/* Пошук першого елемента – вихід після знаходження */
let numbers = [3, 7, 1, 9, 4, 6];
let target = 9;
for (let num of numbers) {
  if (num === target) {
    console.log('Знайдено:', num);
    break;
  }
}
```

**continue** – пропускає поточну ітерацію і переходить до наступної:

*файл javascript*

```
for (let i = 0; i < 10; i++) {
  if (i % 2 === 0) continue; // пропустити парні
  console.log(i); // 1, 3, 5, 7, 9
}
```

*файл javascript*

```
/* Виведення лише непорожніх значень */
let names = ['Іван', '', 'Петро', '', 'Олена'];
for (let name of names) {
  if (!name) continue;
  console.log(name);
}
```

## 6. Покрокове виконання коду у консолі браузера

Браузерні DevTools надають інструменти для відлагодження JS-коду – покрокового виконання і інспекції значень змінних.

**Відкриття DevTools:** клавіша F12 або Ctrl+Shift+I → вкладка **Sources**.

**Точка зупину (breakpoint)** – позначка на рядку коду, де виконання зупиниться:

- клік на номер рядка у вкладці Sources – встановлює breakpoint;
- виконання зупиняється на цьому рядку;
- можна інспектувати значення змінних у панелі Scope.

**Оператор debugger** – програмна точка зупину прямо у коді:

*файл javascript*

```
let x = 5;
let y = 10;
debugger; // виконання зупиниться тут якщо DevTools відкриті
let sum = x + y;
console.log(sum);
```

**Кнопки керування виконанням:**

| Кнопка | Клавіша | Дія |
|---|---|---|
| Resume | F8 | Продовжити до наступного breakpoint |
| Step over | F10 | Виконати поточний рядок, не заходити у функцію |

| | | |
|---|---|---|
| Step into | F11 | Зайти всередину функції |
| Step out | Shift+F11 | Вийти з поточної функції |

**Панель Scope** – показує всі змінні поточної та глобальної областей видимості з їх актуальними значеннями.

**Панель Watch** – дозволяє відстежувати конкретні вирази під час виконання:

*файл javascript*

```
/* Додати у Watch: */
x + y
fruits.length
typeof name
```

**Консольне виведення для відлагодження:**

*файл javascript*

```
console.log('x =', x, 'y =', y);
console.log({ x, y }); /* виведення об'єктом – зручно бачити назви */
console.table(arr);    /* масив у вигляді таблиці */
```

## Контрольні запитання

1. Які форми має умовний оператор if? Наведіть приклад кожної.
2. Коли доцільно використовувати if...else if замість кількох окремих if?
3. Яке значення умови вважається хибним у JS? Назвіть усі falsy значення.
4. Коли доцільно використовувати тернарний оператор замість if...else?
5. Що таке оператор switch і в яких випадках він зручніший за if...else if?
6. Що відбувається у switch якщо не вказати break? Наведіть приклад де це корисно.
7. Як порівнює значення оператор switch – суворо чи нестрого?
8. Що повертає оператор && і що повертає ||? Наведіть приклади з нелогічними типами.
9. Яка різниця між операторами || і ??? Наведіть приклад де вони дають різний результат.
10. Що робить подвійне заперечення !! і коли воно застосовується?
11. З яких трьох частин складається цикл for і за що кожна відповідає?
12. Яка різниця між циклами while і do...while? Коли використовується кожен?

13. Для перебору яких структур даних призначений for...of? Наведіть два приклади.
14. Чим for...of зручніший за звичайний for при переборі масиву?
15. Що робить оператор break у циклі? Наведіть практичний приклад його застосування.
16. Що робить оператор continue? Чим він відрізняється від break?
17. Що таке точка зупину (breakpoint) і як її встановити у DevTools?
18. Що робить оператор debugger і за якої умови він спрацьовує?
19. Яка різниця між кнопками Step over і Step into у режимі покрокового виконання?
20. Для чого використовується панель Watch у DevTools і що до неї можна додавати?

# Тема 10. Структури даних: масиви і рядки.

**Мета:** сформувати вміння оголошувати масиви та рядки, працювати з їх методами, застосовувати оператор spread та деструктуризацію масивів, використовувати шаблонні рядки та рядкові методи для вирішення практичних задач.

**План**

1. Оголошення та індексація масивів.
2. Методи масиву: додавання і видалення.
3. Методи масиву: пошук і сортування.
4. Методи масиву: перебір та трансформація.
5. Оператор spread та деструктуризація масивів.
6. Оголошення рядків. Шаблонні рядки.
7. Рядкові методи.
8. Деструктуризація рядків.

## 1. Оголошення та індексація масивів

**Масив** – впорядкована колекція елементів будь-яких типів. Елементи зберігаються за числовими індексами починаючи з нуля.

**Оголошення масиву:**

*файл javascript*

```
/* Літеральний синтаксис – рекомендований */
let fruits = ['яблуко', 'банан', 'апельсин'];
let numbers = [1, 2, 3, 4, 5];
let mixed = [1, 'привіт', true, null]; /* різні типи */
let empty = []; /* порожній масив */
/* Конструктор – рідко використовується */
let arr = new Array(3);        /* масив з 3 порожніх слотів */
let arr2 = new Array(1, 2, 3); /* [1, 2, 3] */
```

**Індексація – доступ до елементів:**

*файл javascript*

```
let fruits = ['яблуко', 'банан', 'апельсин'];
fruits[0]  // 'яблуко'  – перший елемент
fruits[1]  // 'банан'
fruits[2]  // 'апельсин'
fruits[3]  // undefined – елемент відсутній
```

*/* Метод at() – підтримує від'ємні індекси */*

```
fruits.at(0)   // 'яблуко'
fruits.at(-1)  // 'апельсин' – останній елемент
fruits.at(-2)  // 'банан'
```

**Властивість length:**

*файл javascript*

```
let fruits = ['яблуко', 'банан', 'апельсин'];
fruits.length     // 3
fruits[fruits.length - 1] // 'апельсин' – останній елемент
```

**Зміна елементів:**

*файл javascript*

```
fruits[1] = 'груша'; // ['яблуко', 'груша', 'апельсин']
```

**Багатовимірні масиви:**

*файл javascript*

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
matrix[0][0] // 1
matrix[1][2] // 6
matrix[2][1] // 8
```

## 2. Методи масиву: додавання і видалення

**Додавання елементів:**

*файл javascript*

```
let arr = [1, 2, 3];
arr.push(4);        // додає в кінець → [1, 2, 3, 4]
arr.push(5, 6);     // кілька елементів → [1, 2, 3, 4, 5, 6]
arr.unshift(0);     // додає на початок → [0, 1, 2, 3, 4, 5, 6]
```

**Видалення елементів:**

*файл javascript*

```
let arr = [1, 2, 3, 4, 5];
arr.pop();          // видаляє і повертає останній → 5, arr = [1, 2, 3, 4]
arr.shift();        // видаляє і повертає перший → 1, arr = [2, 3, 4]
```

**Метод splice – видалення, вставка, заміна:**

*файл javascript*

```
let arr = [1, 2, 3, 4, 5];
```

```
/* splice(початок, кількість) – видалення */
arr.splice(1, 2);       // видаляє 2 елементи з індексу 1 → [1, 4, 5]
/* splice(початок, 0, елементи) – вставка */
arr.splice(1, 0, 'a', 'b'); // вставляє на місце 1 → [1, 'a', 'b', 4, 5]
/* splice(початок, кількість, елементи) – заміна */
arr.splice(1, 2, 'x');   // замінює 2 елементи одним → [1, 'x', 4, 5]
```

**Метод slice – копіювання частини масиву (не змінює оригінал):**

*файл javascript*

```
let arr = [1, 2, 3, 4, 5];
arr.slice(1, 3)  // [2, 3] – від 1 до 3 (не включаючи 3)
arr.slice(2)     // [3, 4, 5] – від 2 до кінця
arr.slice(-2)    // [4, 5] – останні два
arr.slice()      // [1, 2, 3, 4, 5] – копія всього масиву
```

**Об'єднання масивів:**

*файл javascript*

```
let a = [1, 2];
let b = [3, 4];
let combined = a.concat(b);     // [1, 2, 3, 4]
let combined2 = a.concat(b, [5, 6]); // [1, 2, 3, 4, 5, 6]
```

### 3. Методи масиву: пошук і сортування

**Пошук елементів:**

*файл javascript*

```
let arr = [1, 2, 3, 2, 1];
arr.indexOf(2)       // 1 – індекс першого входження
arr.lastIndexOf(2)   // 3 – індекс останнього входження
arr.indexOf(5)       // -1 – не знайдено
arr.includes(3)      // true
arr.includes(5)      // false
/* find – повертає перший елемент що відповідає умові */
let users = [
  { id: 1, name: 'Іван' },
  { id: 2, name: 'Петро' }
];
users.find(user => user.id === 2)      // { id: 2, name: 'Петро' }
users.findIndex(user => user.id === 2) // 1 – індекс знайденого елемента
```

**Перевірка елементів:**

*файл javascript*

```
let numbers = [1, 2, 3, 4, 5];
numbers.some(n => n > 4)    // true – хоча б один більше 4
numbers.every(n => n > 0)   // true – всі більше 0
numbers.every(n => n > 2)   // false – не всі більше 2
```

**Сортування:**

*файл javascript*

```
let fruits = ['банан', 'яблуко', 'апельсин'];
fruits.sort(); // ['апельсин', 'банан', 'яблуко'] – алфавітний порядок
/* Числове сортування – потребує функції порівняння */
let numbers = [10, 1, 5, 2, 8];
numbers.sort((a, b) => a - b); // [1, 2, 5, 8, 10] – за зростанням
numbers.sort((a, b) => b - a); // [10, 8, 5, 2, 1] – за спаданням
/* reverse – обертає порядок елементів */
numbers.reverse(); // змінює оригінальний масив
```

**join – об'єднання у рядок:**

*файл javascript*

```
let fruits = ['яблуко', 'банан', 'апельсин'];
fruits.join(', ')  // 'яблуко, банан, апельсин'
fruits.join(' - ') // 'яблуко - банан - апельсин'
fruits.join('')    // 'яблукобананапельсин'
```

## 4. Методи масиву: перебір та трансформація

**forEach – перебір без повернення значення:**

*файл javascript*

```
let fruits = ['яблуко', 'банан', 'апельсин'];
fruits.forEach((fruit, index) => {
  console.log(index, fruit);
});
/* 0 яблуко
   1 банан
   2 апельсин */
```

**map – трансформація: повертає новий масив:**

*файл javascript*

```
let numbers = [1, 2, 3, 4, 5];
let doubled = numbers.map(n => n * 2);
// [2, 4, 6, 8, 10]

let users = [{ name: 'Іван' }, { name: 'Петро' }];
```

```
let names = users.map(user => user.name);
// ['Іван', 'Петро']
```

**filter – фільтрація: повертає новий масив з елементами що відповідають умові:**

*файл javascript*

```
let numbers = [1, 2, 3, 4, 5, 6];
let even = numbers.filter(n => n % 2 === 0);
// [2, 4, 6]
let users = [
  { name: 'Іван', age: 17 },
  { name: 'Петро', age: 25 },
  { name: 'Олена', age: 20 }
];
let adults = users.filter(user => user.age >= 18);
// [{ name: 'Петро', age: 25 }, { name: 'Олена', age: 20 }]
```

**reduce – зведення масиву до одного значення:**

*файл javascript*

```
let numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce((accumulator, current) => accumulator + current, 0);
// 15
let max = numbers.reduce((acc, cur) => cur > acc ? cur : acc, numbers[0]);
// 5
/* Підрахунок суми цін */
let cart = [
  { name: 'Хліб', price: 30 },
  { name: 'Молоко', price: 45 },
  { name: 'Сир', price: 120 }
];
let total = cart.reduce((sum, item) => sum + item.price, 0);
// 195
```

**flat та flatMap:**

*файл javascript*

```
let nested = [1, [2, 3], [4, [5, 6]]];

nested.flat()    // [1, 2, 3, 4, [5, 6]] – один рівень
nested.flat(2)   // [1, 2, 3, 4, 5, 6]  – два рівні
```

```
nested.flat(Infinity) // повністю розгортає
let sentences = ['Привіт світе', 'JavaScript клас'];
sentences.flatMap(s => s.split(' '));
// ['Привіт', 'світе', 'JavaScript', 'клас']
```

**5. Оператор spread та деструктуризація масивів**

**Оператор spread (...)** – розгортає масив у список значень:

*файл javascript*

```
let a = [1, 2, 3];
let b = [4, 5, 6];
/* Об'єднання масивів */
let combined = [...a, ...b];       // [1, 2, 3, 4, 5, 6]
let withExtra = [...a, 0, ...b];   // [1, 2, 3, 0, 4, 5, 6]
/* Копіювання масиву */
let copy = [...a];                 // [1, 2, 3] – нова копія
/* Передача у функцію */
let numbers = [3, 1, 4, 1, 5];
Math.max(...numbers);              // 5
```

**Деструктуризація масивів** – розпаковування значень масиву у змінні:

*файл javascript*

```
let fruits = ['яблуко', 'банан', 'апельсин'];
let [first, second, third] = fruits;
// first = 'яблуко', second = 'банан', third = 'апельсин'
/* Пропуск елементів */
let [a, , c] = fruits;
// a = 'яблуко', c = 'апельсин'
/* Значення за замовчуванням */
let [x = 0, y = 0, z = 0, w = 0] = [1, 2];
// x = 1, y = 2, z = 0, w = 0
/* Rest-елемент */
let [head, ...tail] = [1, 2, 3, 4, 5];
// head = 1, tail = [2, 3, 4, 5]
/* Обмін значень змінних */
let m = 1, n = 2;
[m, n] = [n, m];
// m = 2, n = 1
```

**6. Оголошення рядків. Шаблонні рядки**

**Оголошення рядків:**

*файл javascript*

```
let single = 'Одинарні лапки';
let double = "Подвійні лапки";
let template = `Зворотні лапки`;
```

Одинарні і подвійні лапки рівнозначні. Зворотні лапки – шаблонні рядки з розширеними можливостями.

**Спеціальні символи:**

*файл javascript*

```
let str = 'Рядок з\nновим рядком';
let tab = 'Колонка1\tКолонка2';
let quote = 'Він сказав: \'Привіт\'"; /* екранування */
let backslash = 'C:\\Users\\name';
```

**Незмінність рядка** – рядки в JS незмінні. Будь-який метод повертає новий рядок, не змінює оригінал:

*файл javascript*

```
let str = 'привіт';
str[0] = 'П'; /* не змінює рядок */
console.log(str); // 'привіт'
```

**Шаблонні рядки (template literals):**

*файл javascript*

```
let name = 'Іван';
let age = 25;
/* Вбудовані вирази через ${} */
let greeting = `Привіт, ${name}! Тобі ${age} років.`;
// 'Привіт, Іван! Тобі 25 років.'
/* Вирази у шаблоні */
let a = 5, b = 3;
console.log(`Сума: ${a + b}`);          // Сума: 8
console.log(`Більше: ${a > b ? a : b}`); // Більше: 5
/* Багаторядковий рядок */
let html = `
  <div>
    <h1>${name}</h1>
    <p>Вік: ${age}</p>
  </div>
`;
```

**7. Рядкові методи**

**Довжина та доступ до символів:**

*файл javascript*

```
let str = 'привіт';
str.length      // 6
str[0]          // 'п'
str.at(-1)      // 'т' – останній символ
str.charAt(2)   // 'и'
```

**Пошук:**

*файл javascript*

```
let str = 'JavaScript – це круто';
str.indexOf('це')        // 15 – індекс першого входження
str.indexOf('python')    // -1 – не знайдено
str.lastIndexOf('a')     // індекс останньої 'a'
str.includes('круто')    // true
str.startsWith('Java')   // true
str.endsWith('круто')    // true
str.search(/\d+/)        // індекс першого збігу з RegExp
```

**Вилучення частини рядка:**

*файл javascript*

```
let str = 'JavaScript';
str.slice(0, 4)    // 'Java' – від 0 до 4 (не включаючи)
str.slice(4)       // 'Script' – від 4 до кінця
str.slice(-6)      // 'Script' – останні 6 символів
str.slice(-6, -3)  // 'Scr'
str.substring(0, 4) // 'Java' – аналог slice, не підтримує від'ємні
```

**Заміна:**

*файл javascript*

```
let str = 'Привіт, світе!';
str.replace('світе', 'JS')      // 'Привіт, JS!' – перший збіг
str.replaceAll('і', 'і')        // замінює всі входження
str.replace(/\s/g, '_')         // заміна через RegExp
```

**Зміна регістра:**

*файл javascript*

```
let str = 'JavaScript';
str.toUpperCase() // 'JAVASCRIPT'
str.toLowerCase() // 'javascript'
```

**Обрізання пробілів:**

*файл javascript*

```
let str = '   привіт   ';
str.trim()       // 'привіт' – обрізає з обох боків
str.trimStart()  // 'привіт   ' – лише зліва
str.trimEnd()    // '   привіт' – лише справа
```

**Розбиття та повторення:**

*файл javascript*

```
let str = 'яблуко,банан,апельсин';
str.split(',')    // ['яблуко', 'банан', 'апельсин']
str.split('')     // масив окремих символів
str.split(',', 2) // ['яблуко', 'банан'] – обмеження кількості
'ha'.repeat(3)    // 'hahaha'
'5'.padStart(4, '0') // '0005' – доповнення зліва
'5'.padEnd(4, '0')   // '5000' – доповнення справа
```

## 8. Деструктуризація рядків

Рядок є ітерованим об'єктом, тому до нього можна застосувати деструктуризацію і spread:

*файл javascript*

```
let str = 'Іван';
let [a, b, c, d] = str;
// a = 'І', b = 'в', c = 'а', d = 'н'
/* Rest у деструктуризації рядка */
let [first, ...rest] = 'привіт';
// first = 'п', rest = ['р', 'и', 'в', 'і', 'т']
/* Spread рядка у масив символів */
let chars = [...'привіт'];
// ['п', 'р', 'и', 'в', 'і', 'т']
/* Порівняння з split */
'привіт'.split('') // ['п', 'р', 'и', 'в', 'і', 'т'] – аналогічний результат
[...'привіт']      // ['п', 'р', 'и', 'в', 'і', 'т'] – але коректно з Unicode
```

**Практичний приклад – підрахунок символів:**

*файл javascript*

```
let str = 'javascript';
let charCount = [...str].reduce((acc, char) => {
  acc[char] = (acc[char] || 0) + 1;
  return acc;
}, {});
```

```
// { j: 1, a: 3, v: 1, s: 1, c: 1, r: 1, i: 1, p: 1, t: 1 }
```

## Контрольні запитання

1. Що таке масив у JavaScript? Як оголосити порожній масив і масив з елементами?
2. З якого індексу починається нумерація елементів масиву? Що повернеться при зверненні до неіснуючого індексу?
3. Яка різниця між методами at() і зверненням через []? Наведіть приклад з від'ємним індексом.
4. Яка різниця між методами push/pop і shift/unshift?
5. Що робить метод splice? Чим він відрізняється від slice?
6. Які методи масиву використовуються для пошуку елементів? Чим відрізняється indexOf від find?
7. Яка різниця між some і every? Наведіть приклад кожного.
8. Чому для сортування чисел потрібна функція порівняння? Що поверне [10, 1, 5].sort() без неї?
9. Що робить метод forEach? Чим він відрізняється від map?
10. Що повертає метод filter? Наведіть практичний приклад його застосування.
11. Що робить метод reduce і які аргументи він приймає? Наведіть приклад підрахунку суми.
12. Для чого використовується оператор spread з масивами? Наведіть два приклади застосування.
13. Що таке деструктуризація масиву? Як пропустити елемент при деструктуризації?
14. Що робить rest-елемент при деструктуризації масиву? Наведіть приклад.
15. Яка різниця між одинарними лапками і зворотними лапками при оголошенні рядка?
16. Що таке шаблонний рядок і які переваги він має над звичайним рядком?
17. Яка різниця між методами slice і substring для рядків?
18. Які методи рядка використовуються для пошуку підрядка? Чим відрізняється includes від indexOf?
19. Які методи використовуються для обрізання пробілів у рядку? Чим відрізняється trim від trimStart і trimEnd?
20. Чому для розбиття рядка на символи з урахуванням Unicode краще використовувати [...str] замість split('')?

# Тема 11. Функції та асинхронне програмування.

**Мета:** сформувати розуміння механізмів оголошення та виклику функцій, навчити працювати з параметрами, замиканнями та колбеками, застосовувати стрілкові функції, будувати асинхронний код через Promise та async/await, обробляти помилки за допомогою try...catch.

**План**

1. Function Declaration і Function Expression.
2. Параметри функції та значення за замовчуванням.
3. Замикання та callback.
4. Стрілкові функції.
5. Асинхронне програмування. Event Loop.
6. Promise: створення та ланцюжок.
7. Async/await.
8. Обробка помилок через try...catch.

## 1. Function Declaration і Function Expression

**Функція** – іменований блок коду, що виконує певну задачу і може бути викликаний багаторазово.

**Function Declaration – оголошення функції:**

*файл javascript*

```
function greet(name) {
  return `Привіт, ${name}!`;
}
greet('Іван'); // 'Привіт, Іван!'
```

**Function Expression – функція як значення:**

*файл javascript*

```
const greet = function(name) {
  return `Привіт, ${name}!`;
};
greet('Іван'); // 'Привіт, Іван!'
```

**Ключова різниця – hoisting:**

*файл javascript*

```
/* Function Declaration піднімається – можна викликати до оголошення */
greet('Іван'); // працює
function greet(name) { return `Привіт, ${name}!`; }
```

```
/* Function Expression не піднімається */
greet('Іван'); // помилка – greet ще не визначено
const greet = function(name) { return `Привіт, ${name}!`; };
```

**Повернення значення через return:**

*файл javascript*

```
function sum(a, b) {
  return a + b; /* виконання функції завершується */
  console.log('Цей код не виконається');
}
function greet(name) {
  if (!name) return 'Привіт, Гість!'; /* ранній вихід */
  return `Привіт, ${name}!`;
}
```

Функція без return або з порожнім return повертає undefined.

**Область видимості змінних:**

*файл javascript*

```
let globalVar = 'глобальна';
function example() {
  let localVar = 'локальна';
  console.log(globalVar); /* доступна – функція бачить зовнішні змінні */
  console.log(localVar);  /* доступна */
}
console.log(localVar); /* помилка – локальна змінна недоступна зовні */
```

## 2. Параметри функції та значення за замовчуванням

**Параметри та аргументи:**

*файл javascript*

```
function sum(a, b) {  /* a, b – параметри */
  return a + b;
}
sum(3, 5); /* 3, 5 – аргументи */
```

Якщо аргументів менше ніж параметрів – відсутні отримують undefined.

**Значення за замовчуванням:**

*файл javascript*

```
function greet(name = 'Гість', greeting = 'Привіт') {
  return `${greeting}, ${name}!`;
}
```

```
greet();              // 'Привіт, Гість!'
greet('Іван');        // 'Привіт, Іван!'
greet('Іван', 'Добрий день'); // 'Добрий день, Іван!'
```

Значенням за замовчуванням може бути будь-який вираз:

```
javascript
function createId(prefix = 'user', timestamp = Date.now()) {
  return `${prefix}_${timestamp}`;
}
```

**Rest-параметри – довільна кількість аргументів:**

*файл javascript*

```
function sum(...numbers) {
  return numbers.reduce((acc, n) => acc + n, 0);
}
sum(1, 2, 3)       // 6
sum(1, 2, 3, 4, 5) // 15
```

Rest-параметр завжди останній:

```
javascript
function log(level, ...messages) {
  console.log(`[${level}]`, ...messages);
}
log('INFO', 'Завантаження', 'завершено'); // [INFO] Завантаження завершено
```

**Оператор spread при виклику функції:**

*файл javascript*

```
let numbers = [3, 1, 4, 1, 5];
Math.max(...numbers)    // 5
Math.min(...numbers)    // 1
function sum(a, b, c) { return a + b + c; }
sum(...[1, 2, 3])       // 6
```

### 3. Замикання та callback

**Замикання (closure)** – функція, що зберігає доступ до змінних зовнішньої функції навіть після того як зовнішня функція завершила виконання:

*файл javascript*

```
function makeCounter() {
  let count = 0; /* змінна у зовнішній функції */
```

```
  return function() {
    count++;
    return count;
  };
}
const counter = makeCounter();
counter(); // 1
counter(); // 2
counter(); // 3
const counter2 = makeCounter(); /* незалежний лічильник */
counter2(); // 1
```

Практичне застосування замикань:

*файл javascript*

```
function multiply(factor) {
  return function(number) {
    return number * factor;
  };
}
const double = multiply(2);
const triple = multiply(3);
double(5); // 10
triple(5); // 15
```

**Callback-функції** – функції, що передаються як аргументи і викликаються всередині іншої функції:

*файл javascript*

```
function processUser(name, callback) {
  const user = { name, id: Date.now() };
  callback(user);
}
processUser('Іван', function(user) {
  console.log(`Створено користувача: ${user.name}`);
});
```

Callback у методах масиву:

javascript

```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(function(n) { console.log(n); });
numbers.map(function(n) { return n * 2; });
```

numbers.filter(function(n) { return n > 2; });

**4. Стрілкові функції**

**Стрілкові функції (arrow functions)** – коротший синтаксис Function Expression:

*файл javascript*

```
/* Звичайна функція */
const greet = function(name) {
  return `Привіт, ${name}!`;
};
/* Стрілкова функція */
const greet = (name) => {
  return `Привіт, ${name}!`;
};
/* Скорочений запис – один параметр, одна інструкція */
const greet = name => `Привіт, ${name}!`;
/* Без параметрів */
const sayHello = () => 'Привіт!';
/* Кілька параметрів */
const sum = (a, b) => a + b;
/* Повернення об'єкта – у дужках */
const getUser = name => ({ name, id: 1 });
```

**Стрілкові функції у методах масиву:**

*файл javascript*

```
let numbers = [1, 2, 3, 4, 5];
numbers.map(n => n * 2);
numbers.filter(n => n % 2 === 0);
numbers.reduce((acc, n) => acc + n, 0);
```

**Ключова відмінність – this:**

*файл javascript*

```
/* Звичайна функція – this залежить від контексту виклику */
const obj = {
  name: 'Іван',
  greet: function() {
    console.log(this.name); // 'Іван'
  }
};
/* Стрілкова функція – this береться з навколишнього контексту */
```

```
const obj2 = {
  name: 'Іван',
  greet: () => {
    console.log(this.name); // undefined – this є window/undefined
  }
};
```

Стрілкові функції не мають власного this – не підходять для методів об'єктів, але зручні у колбеках.

### 5. Асинхронне програмування. Event Loop

JavaScript – однопотокова мова. Код виконується у єдиному потоці – по одній операції за раз.

**Event Loop** – механізм виконання асинхронного коду:

Call Stack → Web APIs → Callback Queue → Event Loop → Call Stack

- **Call Stack** – стек поточних викликів функцій
- **Web APIs** – браузерні API: setTimeout, fetch, DOM-події
- **Callback Queue** – черга готових до виконання колбеків
- **Event Loop** – постійно перевіряє: якщо стек порожній – переміщує колбек з черги у стек

*файл javascript*

```
console.log('1 – синхронно');
setTimeout(() => {
  console.log('3 – асинхронно, через 0мс');
}, 0);
console.log('2 – синхронно');
/* Порядок виведення: 1, 2, 3 */
```

### 6. Promise: створення та ланцюжок

**Promise** – об'єкт що представляє результат асинхронної операції. Має три стани: pending (очікування), fulfilled (виконано), rejected (відхилено).

**Створення Promise:**

*файл javascript*

```
const promise = new Promise((resolve, reject) => {
  /* асинхронна операція */
  const success = true;

  if (success) {
    resolve('Дані отримано'); /* переводить у fulfilled */
```

```
  } else {
    reject('Помилка з\'єднання'); /* переводить у rejected */
  }
});
```

**Симуляція асинхронності через setTimeout:**

*файл javascript*

```
function loadData(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (id > 0) {
        resolve({ id, name: 'Іван' });
      } else {
        reject('Невірний id');
      }
    }, 1000);
  });
}
```

**Обробка результату:**

*файл javascript*

```
loadData(1)
  .then(data => console.log(data))    /* fulfilled */
  .catch(error => console.log(error)) /* rejected */
  .finally(() => console.log('Завершено')); /* завжди */
```

**Ланцюжок Promise:**

*файл javascript*

```
loadData(1)
  .then(user => {
    console.log(user);
    return loadData(2); /* повертає новий Promise */
  })
  .then(user2 => {
    console.log(user2);
    return user2.name;
  })
  .then(name => console.log(`Ім'я: ${name}`))
  .catch(error => console.log(error));
```

**Статичні методи Promise:**

*файл javascript*

```
/* Promise.all – чекає всі, відхиляє якщо хоча б один відхилений */
Promise.all([loadData(1), loadData(2), loadData(3)])
  .then(results => console.log(results))
  .catch(error => console.log(error));
/* Promise.allSettled – чекає всі незалежно від результату */
Promise.allSettled([loadData(1), loadData(-1)])
  .then(results => results.forEach(r => console.log(r.status)));
/* Promise.race – повертає результат першого завершеного */
Promise.race([loadData(1), loadData(2)])
  .then(first => console.log(first));
```

**7. Async/await**

async/await – синтаксичний цукор над Promise, що дозволяє писати асинхронний код у синхронному стилі.

**async** – позначає функцію як асинхронну. Функція завжди повертає Promise:

*файл javascript*

```
async function fetchUser() {
  return { name: 'Іван' }; /* автоматично загортається у Promise */
}
fetchUser().then(user => console.log(user));
```

**await** – призупиняє виконання async-функції до вирішення Promise. Використовується лише всередині async-функції:

javascript

```
async function loadUser(id) {
  const user = await loadData(id); /* чекає вирішення Promise */
  console.log(user);
  return user;
}
```

**Порівняння Promise і async/await:**

*файл javascript*

```
/* Promise */
function getUser(id) {
  return loadData(id)
    .then(user => getUserPosts(user.id))
    .then(posts => console.log(posts))
    .catch(error => console.log(error));
```

```
}
/* async/await – читабельніший варіант */
async function getUser(id) {
  const user = await loadData(id);
  const posts = await getUserPosts(user.id);
  console.log(posts);
}
```

**Паралельне виконання з async/await:**

*файл javascript*

```
async function loadAll() {
  /* Послідовно – повільніше */
  const user1 = await loadData(1);
  const user2 = await loadData(2);
  /* Паралельно через Promise.all – швидше */
  const [u1, u2] = await Promise.all([loadData(1), loadData(2)]);
}
```

## 8. Обробка помилок через try...catch

**try...catch** – механізм перехоплення помилок під час виконання коду:

*файл javascript*

```
try {
  /* код що може викинути помилку */
  const result = riskyOperation();
  console.log(result);
} catch (error) {
  /* виконується якщо сталась помилка */
  console.log('Помилка:', error.message);
} finally {
  /* виконується завжди */
  console.log('Завершено');
}
```

**Об'єкт помилки:**

*файл javascript*

```
try {
  null.property; /* TypeError */
} catch (error) {
  console.log(error.name);    /* 'TypeError' */
  console.log(error.message); /* Cannot read properties of null */
```

```
  console.log(error.stack);   /* стек викликів */
}
```

**Власні помилки через throw:**

*файл javascript*

```
function divide(a, b) {
  if (b === 0) throw new Error('Ділення на нуль');
  return a / b;
}
try {
  console.log(divide(10, 0));
} catch (error) {
  console.log(error.message); // 'Ділення на нуль'
}
```

**Обробка помилок у async/await:**

*файл javascript*

```
async function loadUser(id) {
  try {
    const user = await loadData(id);
    const posts = await getUserPosts(user.id);
    return { user, posts };
  } catch (error) {
    console.log('Помилка завантаження:', error.message);
    return null;
  } finally {
    console.log('Запит завершено');
  }
}
```

**Типи вбудованих помилок:**

| Тип | Причина |
|---|---|
| Error | Загальна помилка |
| TypeError | Невірний тип значення |
| ReferenceError | Звернення до неоголошеної змінної |

| | |
|---|---|
| SyntaxError | Синтаксична помилка |
| RangeError | Значення поза допустимим діапазоном |

**Контрольні запитання**

1. Яка ключова різниця між Function Declaration і Function Expression з точки зору hoisting?
2. Що повертає функція якщо в ній відсутній оператор return?
3. Яка різниця між параметром і аргументом функції?
4. Що відбувається якщо функція викликається з меншою кількістю аргументів ніж параметрів?
5. Як задати значення за замовчуванням для параметра функції? Наведіть приклад.
6. Що таке rest-параметр і чим він відрізняється від оператора spread?
7. Що таке замикання? Наведіть практичний приклад його застосування.
8. Що таке callback-функція? Наведіть приклад передачі callback як аргументу.
9. Який синтаксис має стрілкова функція? Запишіть три варіанти – з блоком, скорочений, без параметрів.
10. У чому ключова відмінність стрілкової функції від звичайної щодо this?
11. Чому стрілкові функції не підходять для методів об'єкта але зручні у колбеках?
12. Що таке Event Loop і яку проблему він вирішує в однопотоковому JS?
13. У якому порядку виконається код: синхронні інструкції чи колбек setTimeout з затримкою 0мс?
14. Що таке Promise і які три стани він має?
15. Яка різниця між методами .then(), .catch() і .finally()?
16. Яка різниця між Promise.all() і Promise.allSettled()?
17. Що означає ключове слово async перед функцією? Що вона повертає?
18. Що робить await і де його можна використовувати?
19. Як виглядає обробка помилок у async/await функції? Наведіть приклад з try...catch.
20. Які типи вбудованих помилок є в JS? Чим TypeError відрізняється від ReferenceError?

# Тема 12. Об'єктно-орієнтоване програмування.

**Мета:** сформувати розуміння об'єктно-орієнтованого підходу в JavaScript, навчити оголошувати об'єкти та класи, працювати з властивостями та методами, застосовувати успадкування, статичні та приватні поля, перевіряти належність об'єкта до класу.

**План**

1. Оголошення об'єктів та доступ до властивостей.
2. Перебір та деструктуризація об'єктів.
3. Ключове слово this.
4. Класи: оголошення та конструктор.
5. Успадкування: extends та super.
6. Статичні та приватні поля і методи.
7. Перевірка instanceof.

## 1. Оголошення об'єктів та доступ до властивостей

**Об'єкт** – колекція пов'язаних даних і функціональності у вигляді пар ключ-значення.

**Оголошення через літерал – рекомендований спосіб:**

*файл javascript*

```
const user = {
  name: 'Іван',
  age: 25,
  isActive: true,
  address: {
    city: 'Київ',
    street: 'Хрещатик'
  }
};
```

**Доступ до властивостей:**

*файл javascript*

```
/* Через крапку – основний спосіб */
user.name    // 'Іван'
user.age     // 25
/* Через дужки – для динамічних ключів */
user['name'] // 'Іван'
let key = 'age';
```

```
user[key]    // 25 – доступ через змінну
/* Вкладені властивості */
user.address.city    // 'Київ'
user['address']['city'] // 'Київ'
```

**Зміна та додавання властивостей:**

*файл javascript*

```
user.age = 26;          /* зміна існуючої */
user.email = 'i@ua.com'; /* додавання нової */
```

**Видалення властивостей:**

*файл javascript*

```
delete user.isActive;
```

**Перевірка існування властивості:**

*файл javascript*

```
'name' in user       // true
'phone' in user      // false
user.phone !== undefined // альтернативний спосіб
```

**Скорочений синтаксис властивостей:**

*файл javascript*

```
const name = 'Іван';
const age = 25;
/* Якщо ім'я змінної збігається з ключем */
const user = { name, age };
/* еквівалентно { name: name, age: age } */
```

**Обчислювані властивості:**

*файл javascript*

```
const field = 'name';
const user = {
  [field]: 'Іван',      /* ключ з виразу */
  [`get${field}`]: function() { return this[field]; }
};
```

**Методи об'єкта:**

*файл javascript*

```
const user = {
  name: 'Іван',
  /* Звичайний метод */
  greet: function() {
    return `Привіт, ${this.name}!`;
```

```
  },
  /* Скорочений синтаксис методу */
  sayBye() {
    return `До побачення, ${this.name}!`;
  }
};
user.greet();   // 'Привіт, Іван!'
user.sayBye();  // 'До побачення, Іван!'
```

**Копіювання об'єктів:**

*файл javascript*

```
const original = { name: 'Іван', age: 25 };
/* Поверхневе копіювання */
const copy1 = Object.assign({}, original);
const copy2 = { ...original };
/* Глибоке копіювання */
const deep = JSON.parse(JSON.stringify(original));
const deep2 = structuredClone(original); /* сучасний спосіб */
```

**Spread для об'єктів:**

*файл javascript*

```
const base = { name: 'Іван', age: 25 };
const extended = { ...base, email: 'i@ua.com', age: 26 };
/* { name: 'Іван', age: 26, email: 'i@ua.com' } */
/* пізніші значення перезаписують попередні */
```

## 2. Перебір та деструктуризація об'єктів

**Перебір властивостей:**

*файл javascript*

```
const user = { name: 'Іван', age: 25, city: 'Київ' };
/* for...in – перебирає власні та успадковані ключі */
for (let key in user) {
  console.log(key, user[key]);
}
/* Object.keys() – масив ключів */
Object.keys(user)    // ['name', 'age', 'city']
/* Object.values() – масив значень */
Object.values(user)  // ['Іван', 25, 'Київ']

/* Object.entries() – масив пар [ключ, значення] */
```

```
Object.entries(user) // [['name','Іван'],['age',25],['city','Київ']]
/* Перебір через entries */
for (let [key, value] of Object.entries(user)) {
  console.log(`${key}: ${value}`);
}
```

**Деструктуризація об'єктів:**

*файл javascript*

```
const user = { name: 'Іван', age: 25, city: 'Київ' };
/* Базова деструктуризація */
const { name, age } = user;
// name = 'Іван', age = 25
/* Перейменування */
const { name: userName, age: userAge } = user;
// userName = 'Іван', userAge = 25
/* Значення за замовчуванням */
const { name, phone = 'не вказано' } = user;
// phone = 'не вказано'
/* Rest при деструктуризації */
const { name, ...rest } = user;
// name = 'Іван', rest = { age: 25, city: 'Київ' }
/* Вкладена деструктуризація */
const { address: { city, street } } = {
  address: { city: 'Київ', street: 'Хрещатик' }
};
/* Деструктуризація у параметрах функції */
function greet({ name, age = 0 }) {
  return `${name}, ${age} років`;
}
greet(user); // 'Іван, 25 років'
```

### 3. Ключове слово this

this – посилання на об'єкт у контексті якого викликається функція. Значення this залежить від способу виклику.

**this у методі об'єкта:**

*файл javascript*

```
const user = {
  name: 'Іван',
  greet() {
```

```
    console.log(this.name); // 'Іван' – this є user
  }
};
user.greet();
```

**Втрата контексту:**

*файл javascript*

```
const user = {
  name: 'Іван',
  greet() {
    console.log(this.name);
  }
};
const fn = user.greet;
fn(); /* undefined – this є window або undefined у strict mode */
```

**Явна передача контексту:**

*файл javascript*

```
function greet(greeting) {
  return `${greeting}, ${this.name}!`;
}
const user = { name: 'Іван' };
greet.call(user, 'Привіт');   /* виклик з контекстом і аргументами */
greet.apply(user, ['Привіт']); /* аргументи у масиві */
const boundGreet = greet.bind(user); /* створює нову функцію з фіксованим this */
boundGreet('Привіт');
```

**Геттери та сеттери:**

*файл javascript*

```
const user = {
  firstName: 'Іван',
  lastName: 'Петренко',
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
  set fullName(value) {
    [this.firstName, this.lastName] = value.split(' ');
  }
};
```

```
user.fullName;              // 'Іван Петренко'
user.fullName = 'Олег Іваненко';
user.firstName;             // 'Олег'
```

## 4. Класи: оголошення та конструктор

**Клас** – шаблон для створення об'єктів з однаковою структурою та поведінкою.

**Оголошення класу:**

*файл javascript*

```
class User {
  /* Конструктор – викликається при створенні екземпляра */
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  /* Метод екземпляра */
  greet() {
    return `Привіт, ${this.name}!`;
  }
  /* Геттер */
  get info() {
    return `${this.name}, ${this.age} років`;
  }
}
/* Створення екземпляра */
const user1 = new User('Іван', 25);
const user2 = new User('Петро', 30);
user1.greet(); // 'Привіт, Іван!'
user1.info;    // 'Іван, 25 років'
```

**Публічні поля класу (ES2022):**

*файл javascript*

```
class User {
  role = 'user';       /* публічне поле з початковим значенням */
  createdAt = new Date();
  constructor(name) {
    this.name = name;
  }
}
```

**Class Expression:**

*файл javascript*

```
const User = class {
  constructor(name) {
    this.name = name;
  }
};
```

## 5. Успадкування: extends та super

**extends** – дозволяє класу успадкувати властивості та методи іншого класу:

*файл javascript*

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    return `${this.name} видає звук`;
  }
  toString() {
    return `Animal: ${this.name}`;
  }
}
class Dog extends Animal {
  constructor(name, breed) {
    super(name); /* виклик конструктора батьківського класу – обов'язковий */
    this.breed = breed;
  }
  /* Перевизначення методу батьківського класу */
  speak() {
    return `${this.name} гавкає`;
  }
  /* Виклик методу батьківського класу через super */
  toString() {
    return `${super.toString()}, порода: ${this.breed}`;
  }
}
```

```
const dog = new Dog('Рекс', 'вівчарка');
dog.speak();    // 'Рекс гавкає'
dog.toString(); // 'Animal: Рекс, порода: вівчарка'
```

**Багаторівневе успадкування:**

*файл javascript*

```
class GuideDog extends Dog {
  constructor(name, breed, owner) {
    super(name, breed);
    this.owner = owner;
  }
  guide() {
    return `${this.name} веде ${this.owner}`;
  }
}
const guide = new GuideDog('Граф', 'лабрадор', 'Іван');
guide.speak();  // 'Граф гавкає' – успадкований від Dog
guide.guide();  // 'Граф веде Іван'
```

## 6. Статичні та приватні поля і методи

**Статичні поля та методи** – належать класу, а не екземплярам. Викликаються через ім'я класу:

*файл javascript*

```
class MathHelper {
  static PI = 3.14159;
  static square(x) {
    return x * x;
  }
  static cube(x) {
    return x * x * x;
  }
}
MathHelper.PI;        // 3.14159
MathHelper.square(4);  // 16
MathHelper.cube(3);    // 27

/* Не доступні на екземплярі */
const m = new MathHelper();
m.square(4); /* помилка */
```

**Практичне застосування статичних методів – фабричні методи:**

*файл javascript*

```
class User {
  constructor(name, role) {
    this.name = name;
    this.role = role;
  }
  static createAdmin(name) {
    return new User(name, 'admin');
  }
  static createGuest() {
    return new User('Гість', 'guest');
  }
}
const admin = User.createAdmin('Іван');
const guest = User.createGuest();
```

**Приватні поля та методи** – доступні лише всередині класу. Позначаються символом #:

*файл javascript*

```
class BankAccount {
  #balance = 0;        /* приватне поле */
  #owner;
  constructor(owner, initialBalance) {
    this.#owner = owner;
    this.#balance = initialBalance;
  }
  /* Приватний метод */
  #validate(amount) {
    return amount > 0 && amount <= this.#balance;
  }
  deposit(amount) {
    if (amount > 0) this.#balance += amount;
  }
  withdraw(amount) {
    if (this.#validate(amount)) {
      this.#balance -= amount;
      return amount;
```

```
    }
    throw new Error('Недостатньо коштів');
  }
  get balance() {
    return this.#balance;
  }
}
const account = new BankAccount('Іван', 1000);
account.deposit(500);
account.balance;    // 1500
account.#balance;   /* помилка – приватне поле */
```

**7. Перевірка instanceof**

**instanceof** – перевіряє чи є об'єкт екземпляром певного класу або його нащадка:

*файл javascript*

```
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}
const dog = new Dog();
dog instanceof Dog;     // true
dog instanceof Animal;  // true – Dog успадковує Animal
dog instanceof Cat;     // false
dog instanceof Object;  // true – всі об'єкти є екземплярами Object
```

**Практичне застосування:**

*файл javascript*

```
class Shape {
  area() { return 0; }
}
class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }
  area() { return Math.PI * this.radius ** 2; }
}

class Rectangle extends Shape {
```

```
  constructor(width, height) {
    super();
    this.width = width;
    this.height = height;
  }
  area() { return this.width * this.height; }
}
function printArea(shape) {
  if (shape instanceof Circle) {
    console.log(`Коло, площа: ${shape.area().toFixed(2)}`);
  } else if (shape instanceof Rectangle) {
    console.log(`Прямокутник, площа: ${shape.area()}`);
  } else {
    console.log('Невідома фігура');
  }
}
printArea(new Circle(5));       // Коло, площа: 78.54
printArea(new Rectangle(4, 6)); // Прямокутник, площа: 24
```

**typeof vs instanceof:**

| | **typeof** | **instanceof** |
|---|---|---|
| Призначення | Тип примітиву | Клас об'єкта |
| Повертає | Рядок | true / false |
| Для об'єктів | Завжди 'object' | Конкретний клас |
| Приклад | typeof 'str' → 'string' | dog instanceof Dog → true |

## Контрольні запитання

1. Якими двома способами можна отримати доступ до властивості об'єкта? Коли доцільно використовувати кожен?
2. Як перевірити чи існує властивість в об'єкті? Наведіть два способи.
3. Яка різниця між поверхневим і глибоким копіюванням об'єкта? Наведіть приклад кожного.
4. Що робить оператор spread з об'єктами? Що відбувається якщо два об'єкти мають однакові ключі?

5. Яка різниця між Object.keys(), Object.values() і Object.entries()?
6. Що таке деструктуризація об'єкта? Як перейменувати змінну при деструктуризації?
7. Як задати значення за замовчуванням при деструктуризації об'єкта?
8. Що робить rest-оператор при деструктуризації об'єкта? Наведіть приклад.
9. Що таке this і від чого залежить його значення?
10. Що таке втрата контексту і як її уникнути?
11. Яка різниця між call(), apply() і bind()?
12. Що таке геттер і сеттер? Як вони оголошуються в об'єкті та класі?
13. Що таке клас у JavaScript? Яка роль конструктора?
14. Що відбувається якщо не викликати super() у конструкторі дочірнього класу?
15. Яка різниця між методом екземпляра і статичним методом? Як викликається кожен?
16. Де застосовуються статичні методи? Наведіть практичний приклад.
17. Що таке приватні поля і методи класу? Як вони позначаються?
18. Чим приватні поля відрізняються від звичайних властивостей об'єкта?
19. Що перевіряє оператор instanceof? Чи поверне true перевірка екземпляра дочірнього класу відносно батьківського?
20. Яка різниця між typeof і instanceof? Коли доцільно використовувати кожен?

# Тема 13. DOM, BOM та події.

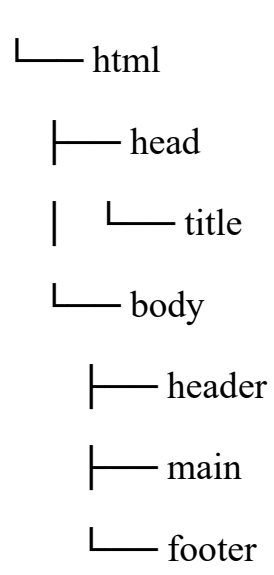
**Мета:** сформувати вміння працювати з DOM-деревом – вибирати, змінювати, створювати та видаляти елементи, керувати подіями та делегуванням, зберігати дані у браузері, обробляти форми та виконувати валідацію засобами JavaScript.

**План**

1. DOM-дерево та вибірка елементів.
2. Властивості елементів: вміст, класи, стилі.
3. Створення, додавання та видалення елементів.
4. localStorage та sessionStorage.
5. Модель подій: addEventListener та поширення.
6. Делегування подій.
7. Робота з формами та валідація засобами JS.

## 1. DOM-дерево та вибірка елементів

**DOM (Document Object Model)** – програмний інтерфейс HTML-документа. Браузер перетворює HTML на дерево об'єктів, кожен з яких можна читати та змінювати через JavaScript.




**Вибірка елементів:**

*файл javascript*

```
/* За id – повертає елемент або null */
document.getElementById('header')
```

```
/* Перший елемент що відповідає CSS-селектору */
document.querySelector('.btn')
document.querySelector('#form input[type="email"]')
document.querySelector('nav > ul > li:first-child')
/* Всі елементи що відповідають селектору – статична NodeList */
document.querySelectorAll('.card')
document.querySelectorAll('ul li')
/* Перетворення NodeList у масив для використання методів масиву */
const cards = [...document.querySelectorAll('.card')];
cards.forEach(card => console.log(card));
```

**Навігація по DOM:**

*файл javascript*

```
const el = document.querySelector('.container');
el.parentElement          /* батьківський елемент */
el.children               /* дочірні елементи (HTMLCollection) */
el.firstElementChild      /* перший дочірній елемент */
el.lastElementChild       /* останній дочірній елемент */
el.nextElementSibling     /* наступний сусідній елемент */
el.previousElementSibling  /* попередній сусідній елемент */
```

**Метод closest()** – шукає найближчого предка що відповідає селектору (включаючи сам елемент):

*файл javascript*

```
/* HTML:
<ul class="list">
  <li class="item">
    <button class="btn-delete">Видалити</button>
  </li>
</ul> */
const btn = document.querySelector('.btn-delete');
btn.closest('.item')  /* знаходить батьківський li.item */
btn.closest('.list')  /* знаходить ul.list */
btn.closest('body')   /* знаходить body */
btn.closest('.popup') /* null – не знайдено */
```

**Метод matches()** – перевіряє чи відповідає елемент селектору:

*файл javascript*

```
const el = document.querySelector('li');
```

```
el.matches('.active')     /* true або false */
el.matches(':first-child') /* перевірка псевдокласу */
```

**2. Властивості елементів: вміст, класи, стилі**

**Вміст елемента:**

*файл javascript*

```
const el = document.querySelector('.title');
/* textContent – текст без HTML-тегів */
el.textContent          /* читання */
el.textContent = 'Новий заголовок' /* запис – безпечно, теги як текст */
/* innerHTML – HTML-вміст */
el.innerHTML            /* читання з тегами */
el.innerHTML = '<strong>Жирний текст</strong>' /* запис – небезпечно з user input */
/* innerText – видимий текст з урахуванням CSS */
el.innerText
```

**Атрибути:**

*файл javascript*

```
const img = document.querySelector('img');
img.getAttribute('src')          /* читання атрибута */
img.setAttribute('alt', 'Фото')   /* встановлення атрибута */
img.removeAttribute('title')       /* видалення атрибута */
img.hasAttribute('src')          /* перевірка наявності */
/* data-атрибути */
/* <div data-user-id="42" data-role="admin"> */
const el = document.querySelector('[data-user-id]');
el.dataset.userId  /* '42' */
el.dataset.role    /* 'admin' */
el.dataset.newProp = 'value' /* додавання data-атрибута */
```

**Класи через classList:**

*файл javascript*

```
const el = document.querySelector('.card');
el.classList.add('active')          /* додати клас */
el.classList.remove('hidden')       /* видалити клас */
el.classList.toggle('expanded')   /* додати якщо немає, видалити якщо є */
el.classList.contains('active')      /* true / false */
el.classList.replace('old', 'new')   /* замінити клас */
```

```
/* Кілька класів одночасно */
el.classList.add('active', 'visible', 'loaded')
el.classList.remove('hidden', 'disabled')
```

**Стилі:**

*файл javascript*

```
const el = document.querySelector('.box');
/* Інлайн-стилі через element.style */
el.style.color = 'red';
el.style.fontSize = '18px';       /* camelCase замість font-size */
el.style.backgroundColor = '#fff';
el.style.display = 'none';
/* Читання обчислених стилів */
const styles = getComputedStyle(el);
styles.color          /* поточний колір з урахуванням всіх CSS */
styles.fontSize       /* поточний розмір шрифту */
```

## 3. Створення, додавання та видалення елементів

**Створення елементів:**

*файл javascript*

```
const div = document.createElement('div');
const p = document.createElement('p');
const img = document.createElement('img');
div.className = 'card';
div.textContent = 'Вміст картки';
p.innerHTML = '<strong>Жирний</strong> текст';
img.src = 'photo.jpg';
img.alt = 'Фото';
```

**Додавання елементів:**

*файл javascript*

```
const container = document.querySelector('.container');
const item = document.createElement('li');
item.textContent = 'Новий пункт';
container.append(item)        /* в кінець, приймає рядки і вузли */
container.prepend(item)       /* на початок */
container.before(item)        /* перед контейнером */
container.after(item)         /* після контейнера */

/* insertAdjacentHTML – вставка HTML у рядковому форматі */
```

```
container.insertAdjacentHTML('beforeend', '<li>Пункт</li>')
container.insertAdjacentHTML('afterbegin', '<li>Перший</li>')
```

**Видалення та переміщення:**

*файл javascript*

```
const el = document.querySelector('.item');
el.remove()                  /* видалення елемента */
el.parentElement.removeChild(el) /* альтернативний спосіб */
/* Переміщення – append видаляє елемент зі старого місця */
const newParent = document.querySelector('.new-container');
newParent.append(el);
```

**Клонування:**

*файл javascript*

```
const original = document.querySelector('.card');
const clone = original.cloneNode(true)  /* true – з дочірніми елементами */
const shallowClone = original.cloneNode(false) /* без дочірніх */
document.querySelector('.wrapper').append(clone);
```

**DocumentFragment – пакетне додавання:**

*файл javascript*

```
const fragment = document.createDocumentFragment();
for (let i = 1; i <= 100; i++) {
  const li = document.createElement('li');
  li.textContent = `Пункт ${i}`;
  fragment.append(li);
}
/* Одна операція додавання замість 100 */
document.querySelector('ul').append(fragment);
```

## 4. localStorage та sessionStorage

**localStorage** – зберігає дані без терміну придатності. Залишаються після закриття браузера:

*файл javascript*

```
/* Запис */
localStorage.setItem('username', 'Іван');
localStorage.setItem('theme', 'dark');
/* Читання */
localStorage.getItem('username')  /* 'Іван' */
localStorage.getItem('missing')   /* null */
```

```
/* Видалення */
localStorage.removeItem('theme')
localStorage.clear()          /* очищення всього сховища */
```

**sessionStorage** – зберігає дані лише на час сесії. Очищується при закритті вкладки. Інтерфейс ідентичний до localStorage.

**Зберігання об'єктів і масивів через JSON:**

*файл javascript*

```
/* Збереження */
const user = { name: 'Іван', age: 25, roles: ['admin', 'user'] };
localStorage.setItem('user', JSON.stringify(user));
/* Читання */
const stored = localStorage.getItem('user');
const parsedUser = stored ? JSON.parse(stored) : null;
```

**Практичний приклад – збереження теми:**

```
javascript
function setTheme(theme) {
  document.body.className = theme;
  localStorage.setItem('theme', theme);
}
function loadTheme() {
  const theme = localStorage.getItem('theme') || 'light';
  document.body.className = theme;
}
loadTheme(); /* викликається при завантаженні сторінки */
```

**Порівняння сховищ:**

| | **localStorage** | **sessionStorage** | **Cookie** |
|---|---|---|---|
| Термін | Без обмежень | Сесія | Задається вручну |
| Обсяг | ~5MB | ~5MB | ~4KB |
| Доступ | JS | JS | JS + сервер |
| Відправка на сервер | Ні | Ні | Так |

## 5. Модель подій: addEventListener та поширення

**addEventListener** – підписка на подію:

*файл javascript*

```
const btn = document.querySelector('.btn');
btn.addEventListener('click', function(event) {
  console.log('Натиснуто');
  console.log(event.target);       /* елемент що викликав подію */
  console.log(event.currentTarget); /* елемент з обробником */
  console.log(event.type);          /* тип події: 'click' */
});
/* Стрілкова функція */
btn.addEventListener('click', (e) => {
  console.log(e.target);
});
/* Видалення обробника – функція має бути іменованою */
function handleClick(e) {
  console.log('Клік');
}
btn.addEventListener('click', handleClick);
btn.removeEventListener('click', handleClick);
```

**Часто вживані події:**

*файл javascript*

```
/* Миша */
el.addEventListener('click', handler)
el.addEventListener('dblclick', handler)
el.addEventListener('mouseover', handler)
el.addEventListener('mouseout', handler)
el.addEventListener('mousemove', handler)
/* Клавіатура */
document.addEventListener('keydown', e => console.log(e.key))
document.addEventListener('keyup', handler)
/* Форми */
input.addEventListener('input', handler)    /* кожне введення символу */
input.addEventListener('change', handler) /* після зміни і втрати фокусу */
form.addEventListener('submit', handler)
/* Документ */
```

document.addEventListener('DOMContentLoaded', handler) /* DOM готовий */

window.addEventListener('load', handler) /* сторінка повністю завантажена */

window.addEventListener('resize', handler) /* зміна розміру вікна */

window.addEventListener('scroll', handler) /* прокручування */

**Поширення подій:**

Подія проходить три фази:

1. **Занурення (capturing)** – від document до цільового елемента
2. **Ціль (target)** – елемент що отримав подію
3. **Спливання (bubbling)** – від цільового елемента до document

*файл javascript*

```
/* За замовчуванням обробники спрацьовують на фазі спливання */
parent.addEventListener('click', () => console.log('parent'))
child.addEventListener('click', () => console.log('child'))
/* При кліку на child: 'child', потім 'parent' */
/* Третій аргумент true – обробник на фазі занурення */
parent.addEventListener('click', handler, true)
```

**Методи об'єкта event:**

```
javascript
el.addEventListener('click', (e) => {
  e.preventDefault()      /* скасування стандартної дії браузера */
  e.stopPropagation()     /* зупинка поширення події */
  e.stopImmediatePropagation() /* зупинка + інших обробників на цьому елементі */
})
/* Практичне застосування preventDefault */
link.addEventListener('click', (e) => {
  e.preventDefault(); /* посилання не перейде по href */
});
form.addEventListener('submit', (e) => {
  e.preventDefault(); /* форма не відправиться на сервер */
});
```

## 6. Делегування подій

**Делегування** – техніка де один обробник встановлюється на батьківський елемент замість множини обробників на дочірні. Використовує механізм спливання подій.

*файл javascript*

```
/* Без делегування – обробник на кожному елементі */
document.querySelectorAll('.btn-delete').forEach(btn => {
  btn.addEventListener('click', handleDelete);
});
/* З делегуванням – один обробник на контейнері */
document.querySelector('.list').addEventListener('click', (e) => {
  if (e.target.matches('.btn-delete')) {
    handleDelete(e);
  }
});
```

**Практичний приклад – динамічний список:**

*файл javascript*

```
const list = document.querySelector('.todo-list');
list.addEventListener('click', (e) => {
  const item = e.target.closest('.todo-item');
  if (!item) return;
  if (e.target.matches('.btn-delete')) {
    item.remove();
  }
  if (e.target.matches('.btn-complete')) {
    item.classList.toggle('completed');
  }
});
/* Нові елементи автоматично підхоплюють обробник */
function addItem(text) {
  list.insertAdjacentHTML('beforeend', `
    <li class="todo-item">
      <span>${text}</span>
      <button class="btn-complete">✓</button>
      <button class="btn-delete">✕</button>
    </li>
  `);
}
```

**Переваги делегування:**

- один обробник замість багатьох;
- автоматично працює для динамічно доданих елементів;

- менше використання пам'яті.

## 7. Робота з формами та валідація засобами JS

**Доступ до елементів форми:**

*файл javascript*

```
const form = document.querySelector('#registration-form');
/* Через elements */
form.elements['username']      /* поле за атрибутом name */
form.elements['email']
/* Через querySelector */
form.querySelector('input[name="username"]')
/* Читання та запис значень */
const input = form.elements['username'];
input.value                    /* поточне значення */
input.value = 'Іван'           /* встановлення значення */
const checkbox = form.elements['agree'];
checkbox.checked               /* true / false */
const select = form.elements['city'];
select.value                   /* вибране значення */
```

**Обробка submit:**

*файл javascript*

```
form.addEventListener('submit', (e) => {
  e.preventDefault(); /* зупинити стандартну відправку */
  const data = {
    username: form.elements['username'].value.trim(),
    email: form.elements['email'].value.trim(),
    password: form.elements['password'].value
  };
  if (validate(data)) {
    sendData(data);
  }
});
```

**Кастомна валідація:**

*файл javascript*

```
function validate(data) {
  const errors = {};
  if (!data.username) {
```

```
  errors.username = "Ім'я обов'язкове";
 } else if (data.username.length < 2) {
  errors.username = "Мінімум 2 символи";
 }
 const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
 if (!data.email) {
  errors.email = 'Email обов\'язковий';
 } else if (!emailRegex.test(data.email)) {
  errors.email = 'Невірний формат email';
 }
 if (!data.password) {
  errors.password = 'Пароль обов\'язковий';
 } else if (data.password.length < 8) {
  errors.password = 'Мінімум 8 символів';
 }
 return errors;
}
```

**Відображення помилок:**

*файл javascript*

```
function showErrors(errors) {
 /* Очищення попередніх помилок */
 document.querySelectorAll('.error-message').forEach(el => el.remove());
 document.querySelectorAll('.input-error').forEach(el => {
  el.classList.remove('input-error');
 });
 /* Відображення нових */
 for (let [field, message] of Object.entries(errors)) {
  const input = form.elements[field];
  input.classList.add('input-error');
  const errorEl = document.createElement('span');
  errorEl.className = 'error-message';
  errorEl.textContent = message;
  input.after(errorEl);
 }
}
```

**Валідація в реальному часі:**

*файл javascript*

```
 form.querySelectorAll('input').forEach(input => {
  input.addEventListener('input', () => {
    const errors = validate({ [input.name]: input.value });
    const errorEl = input.nextElementSibling;
    if (errors[input.name]) {
      input.classList.add('input-error');
      if (errorEl?.classList.contains('error-message')) {
        errorEl.textContent = errors[input.name];
      }
    } else {
      input.classList.remove('input-error');
      if (errorEl?.classList.contains('error-message')) {
        errorEl.remove();
      }
    }
  });
 });
```

**Контрольні запитання**

1. Що таке DOM і яку роль він відіграє у взаємодії JavaScript з HTML-документом?
2. Яка різниця між querySelector() і querySelectorAll()? Що повертає кожен метод?
3. Як перетворити NodeList у масив і навіщо це потрібно?
4. Для чого використовується метод closest()? Наведіть практичний приклад.
5. Яка різниця між textContent і innerHTML? Чому innerHTML потенційно небезпечний?
6. Назвіть методи classList. Чим toggle() відрізняється від add() і remove()?
7. Як читати та змінювати data-атрибути елемента через JavaScript?
8. Яка послідовність дій для створення і додавання нового елемента у DOM?
9. Яка різниця між методами append() і insertAdjacentHTML()?
10. Що таке DocumentFragment і яку проблему він вирішує?
11. Яка різниця між localStorage і sessionStorage?
12. Чому об'єкти і масиви перед збереженням у localStorage потрібно серіалізувати через JSON.stringify()?

13. Яка різниця між event.target і event.currentTarget?
14. Що таке поширення подій? Опишіть три фази поширення.
15. Що робить preventDefault()? Наведіть два практичних приклади.
16. Що таке делегування подій і яку проблему воно вирішує?
17. Чому делегування подій зручне для динамічно доданих елементів?
18. Як у обробнику делегованої події визначити який саме дочірній елемент був натиснутий?
19. Як отримати значення полів форми через JavaScript? Чим відрізняється читання value від checked?
20. Яка різниця між валідацією через атрибути HTML5 і кастомною валідацією засобами JS?

# Модуль IV. Розширені можливості JS та сучасні інструменти розробки

## Тема 14. Взаємодія з сервером та REST API.

**Мета:** сформувати розуміння принципів REST-архітектури та HTTP-протоколу, навчити працювати з форматом JSON, виконувати GET та POST запити через Fetch API, обробляти відповіді сервера та помилки мережі.

**План**

1. Принципи REST архітектури.
2. HTTP-методи та структура запиту.
3. Формат JSON: серіалізація та десеріалізація.
4. Fetch API: базовий синтаксис.
5. GET-запит: отримання даних.
6. POST, PUT, DELETE запити.
7. Обробка відповіді та статусні коди.
8. Обробка помилок.

### 1. Принципи REST архітектури

**REST (Representational State Transfer)** – архітектурний стиль для побудови вебсервісів. Визначає правила взаємодії між клієнтом (браузером) і сервером.

**Основні принципи REST:**

**Клієнт-сервер** – клієнт і сервер розділені та незалежні. Клієнт відповідає за інтерфейс, сервер – за дані та логіку.

**Без стану (Stateless)** – кожен запит містить всю необхідну інформацію. Сервер не зберігає стан між запитами.

**Єдиний інтерфейс** – ресурси адресуються через URL, взаємодія через стандартні HTTP-методи.

**Ресурс** – будь-який об'єкт з яким працює API: користувач, стаття, замовлення. Кожен ресурс має унікальний URL:

```
https://api.example.com/users          /* колекція користувачів */
https://api.example.com/users/42       /* конкретний користувач */
https://api.example.com/users/42/posts /* пости користувача */
```

**Відповідність методів діям:**

| HTTP-метод | Дія | Приклад |
|---|---|---|

| | | |
|---|---|---|
| GET | Отримати | GET /users – список, GET /users/1 – один |
| POST | Створити | POST /users – новий користувач |
| PUT | Замінити | PUT /users/1 – повна заміна |
| PATCH | Оновити | PATCH /users/1 – часткове оновлення |
| DELETE | Видалити | DELETE /users/1 – видалення |

## 2. HTTP-методи та структура запиту

**Структура HTTP-запиту:**

Метод URL HTTP-версія
Заголовки
Тіло запиту (для POST, PUT, PATCH)

```
POST /api/users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer token123
{"name": "Іван", "email": "ivan@ua.com"}
```

**Структура HTTP-відповіді:**

HTTP-версія Статусний-код Повідомлення
Заголовки
Тіло відповіді

```
HTTP/1.1 201 Created
Content-Type: application/json
{"id": 42, "name": "Іван", "email": "ivan@ua.com"}
```

**Статусні коди:**

| **Діапазон** | **Значення** | **Приклади** |
|---|---|---|
| 2xx | Успіх | 200 OK, 201 Created, 204 No Content |
| 3xx | Перенаправлення | 301 Moved, 304 Not Modified |

| | | |
|---|---|---|
| 4xx | Помилка клієнта | 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found |
| 5xx | Помилка сервера | 500 Internal Server Error, 503 Service Unavailable |

**Заголовки запиту:**

*файл javascript*

```
{
 'Content-Type': 'application/json',  /* тип тіла запиту */
 'Authorization': 'Bearer token',     /* токен авторизації */
 'Accept': 'application/json',        /* очікуваний тип відповіді */
}
```

## 3. Формат JSON

**JSON (JavaScript Object Notation)** – текстовий формат обміну даними. Читабельний для людини і легко парситься машиною.

**Синтаксис JSON:**

```
json
{
 "id": 1,
 "name": "Іван Петренко",
 "age": 25,
 "isActive": true,
 "address": {
  "city": "Київ",
  "street": "Хрещатик"
 },
 "roles": ["admin", "user"],
 "phone": null
}
```

**Відмінності від JS-об'єкта:**

- ключі лише у подвійних лапках
- рядки лише у подвійних лапках
- не підтримує функції, undefined, Date
- немає коментарів

**Серіалізація – об'єкт у рядок JSON:**

*файл javascript*

```
const user = {
  name: 'Іван',
  age: 25,
  roles: ['admin', 'user']
};
const json = JSON.stringify(user);
// '{"name":"Іван","age":25,"roles":["admin","user"]}'
/* Форматований вивід */
const pretty = JSON.stringify(user, null, 2);
/*
{
  "name": "Іван",
  "age": 25,
  "roles": ["admin", "user"]
}
*/
/* Вибіркова серіалізація */
JSON.stringify(user, ['name', 'age'])
// '{"name":"Іван","age":25}'
```

**Десеріалізація – рядок JSON у об'єкт:**

*файл javascript*

```
const json = '{"name":"Іван","age":25}';
const user = JSON.parse(json);
user.name  // 'Іван'
user.age   // 25
/* Обробка помилок парсингу */
try {
  const data = JSON.parse(invalidJson);
} catch (error) {
  console.log('Невалідний JSON:', error.message);
}
```

## 4. Fetch API: базовий синтаксис

**Fetch API** – сучасний інтерфейс для виконання HTTP-запитів з браузера. Повертає Promise.

**Базовий синтаксис:**

*файл javascript*

```
fetch(url, options)
```

```
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

**Об'єкт Response – методи читання тіла відповіді:**

*файл javascript*

```
response.json()    /* парсить JSON – повертає Promise */
response.text()    /* повертає текст – повертає Promise */
response.blob()    /* повертає бінарні дані (зображення) */
response.ok        /* true якщо статус 200-299 */
response.status    /* числовий код: 200, 404, 500 */
response.statusText /* 'OK', 'Not Found' */
response.headers   /* заголовки відповіді */
```

**Важлива особливість Fetch** – Promise не відхиляється при HTTP-помилках (404, 500). Він відхиляється лише при мережевих помилках. Тому статус потрібно перевіряти вручну через response.ok.

## 5. GET-запит: отримання даних

*файл javascript*

```
/* Простий GET-запит */
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => response.json())
  .then(users => console.log(users))
  .catch(error => console.log('Помилка:', error));
```

**GET через async/await – рекомендований спосіб:**

*файл javascript*

```
async function getUsers() {
  const response = await fetch('https://jsonplaceholder.typicode.com/users');
  if (!response.ok) {
    throw new Error(`Помилка сервера: ${response.status}`);
  }
  const users = await response.json();
  return users;
}
getUsers()
  .then(users => renderUsers(users))
  .catch(error => console.log(error));
```

**GET з параметрами запиту:**

*файл javascript*

```
async function searchUsers(query, page = 1) {
  const params = new URLSearchParams({
    search: query,
    page: page,
    limit: 10
  });
  const response = await fetch(`/api/users?${params}`);
  /* /api/users?search=Іван&page=1&limit=10 */
  if (!response.ok) throw new Error(`${response.status}`);
  return response.json();
}
```

**Отримання одного ресурсу:**

*файл javascript*

```
async function getUser(id) {
  const response = await fetch(`/api/users/${id}`);
  if (response.status === 404) {
    throw new Error('Користувача не знайдено');
  }
  if (!response.ok) {
    throw new Error(`Помилка: ${response.status}`);
  }

  return response.json();
}
```

## 6. POST, PUT, DELETE запити

**POST – створення ресурсу:**

*файл javascript*

```
async function createUser(userData) {
  const response = await fetch('/api/users', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(userData)
  });
  if (!response.ok) {
    throw new Error(`Помилка створення: ${response.status}`);
```

```
  }
  return response.json(); /* повертає створений об'єкт з id */
}

createUser({ name: 'Іван', email: 'ivan@ua.com' })
  .then(user => console.log('Створено:', user))
  .catch(error => console.log(error));
```

**PUT – повне оновлення ресурсу:**

*файл javascript*

```
async function updateUser(id, userData) {
  const response = await fetch(`/api/users/${id}`, {
    method: 'PUT',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(userData)
  });
  if (!response.ok) throw new Error(`${response.status}`);
  return response.json();
}
```

**PATCH – часткове оновлення:**

*файл javascript*

```
async function patchUser(id, fields) {
  const response = await fetch(`/api/users/${id}`, {
    method: 'PATCH',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(fields) /* лише поля що змінюються */
  });
  if (!response.ok) throw new Error(`${response.status}`);
  return response.json();
}
patchUser(1, { email: 'new@ua.com' }); /* змінюємо лише email */
```

**DELETE – видалення ресурсу:**

*файл javascript*

```
async function deleteUser(id) {
  const response = await fetch(`/api/users/${id}`, {
    method: 'DELETE'
```

```
  });
  if (!response.ok) throw new Error(`${response.status}`);

  /* 204 No Content – тіло відповіді порожнє */
  if (response.status === 204) return true;
  return response.json();
}
```

**Авторизований запит через заголовок:**

*файл javascript*

```
async function getProfile() {
  const token = localStorage.getItem('token');
  const response = await fetch('/api/profile', {
    headers: {
      'Authorization': `Bearer ${token}`,
      'Content-Type': 'application/json'
    }
  });
  if (response.status === 401) {
    throw new Error('Необхідна авторизація');
  }
  if (!response.ok) throw new Error(`${response.status}`);
  return response.json();
}
```

## 7. Обробка відповіді та статусні коди

**Повна обробка статусних кодів:**

*файл javascript*

```
async function request(url, options = {}) {
  const response = await fetch(url, options);
  switch (response.status) {
    case 200:
    case 201:
      return response.json();
    case 204:
      return null; /* No Content */
    case 400:
      const errorData = await response.json();
      throw new Error(`Невірний запит: ${errorData.message}`);
```

```
    case 401:
      throw new Error('Необхідна авторизація');

    case 403:
      throw new Error('Доступ заборонено');
    case 404:
      throw new Error('Ресурс не знайдено');
    case 500:
      throw new Error('Помилка сервера');
    default:
      throw new Error(`Невідома помилка: ${response.status}`);
  }
}
```

**Читання заголовків відповіді:**

*файл javascript*

```
const response = await fetch('/api/users');
response.headers.get('Content-Type')    /* 'application/json' */
response.headers.get('X-Total-Count')   /* кількість записів */
```

## 8. Обробка помилок

**Типи помилок у Fetch:**

*файл javascript*

```
/* Мережева помилка – Promise відхиляється */
/* - відсутній інтернет */
/* - сервер недоступний */
/* - таймаут з'єднання */
/* HTTP-помилка – Promise виконується, але response.ok === false */
/* - 404 Not Found */
/* - 500 Internal Server Error */
/* - 403 Forbidden */
```

**Універсальна функція запиту з обробкою помилок:**

*файл javascript*

```
async function apiFetch(url, options = {}) {
  try {
    const response = await fetch(url, {
      headers: { 'Content-Type': 'application/json' },
      ...options
    });
```

```
    if (!response.ok) {
      const error = await response.json().catch(() => ({}));
      throw new Error(error.message || `HTTP ${response.status}`);
    }
    if (response.status === 204) return null;
    return response.json();
  } catch (error) {
    if (error.name === 'TypeError') {
      throw new Error('Мережева помилка – перевірте з\'єднання');
    }
    throw error;
  }
}
```

**Практичний приклад – завантаження і відображення даних:**

*файл javascript*

```
async function loadAndRenderUsers() {
  const container = document.querySelector('.users-list');
  container.innerHTML = '<p>Завантаження...</p>';
  try {
    const users = await apiFetch('/api/users');
    container.innerHTML = users
      .map(user => `
        <div class="user-card">
          <h3>${user.name}</h3>
          <p>${user.email}</p>
        </div>
      `)
      .join('');
  } catch (error) {
    container.innerHTML = `
      <p class="error">Помилка: ${error.message}</p>
      <button onclick="loadAndRenderUsers()">Спробувати знову</button>
    `;
  }
}
document.addEventListener('DOMContentLoaded', loadAndRenderUsers);
```

**AbortController – скасування запиту:**

*файл javascript*

```
let controller = new AbortController();

async function search(query) {
  controller.abort();            /* скасувати попередній запит */
  controller = new AbortController();
  try {
    const response = await fetch(`/api/search?q=${query}`, {
      signal: controller.signal
    });
    return response.json();
  } catch (error) {
    if (error.name === 'AbortError') return; /* запит скасовано – не помилка
*/
    throw error;
  }
}
/* Пошук під час введення */
input.addEventListener('input', (e) => search(e.target.value));
```

**Контрольні запитання**

1. Що таке REST і які основні принципи лежать в його основі?
2. Як формується URL для REST API? Наведіть приклади URL для колекції та окремого ресурсу.
3. Яке призначення кожного HTTP-методу: GET, POST, PUT, PATCH, DELETE?
4. Яка різниця між PUT і PATCH?
5. Що таке статусний код відповіді? Що означають діапазони 2xx, 4xx і 5xx?
6. Яка різниця між кодами 200, 201 і 204?
7. Яка різниця між кодами 401 і 403?
8. Що таке JSON і чим він відрізняється від JS-об'єкта?
9. Що робить JSON.stringify() і що робить JSON.parse()? Наведіть приклад кожного.
10. Що відбудеться якщо передати у JSON.parse() невалідний рядок?
11. Що таке Fetch API і що він повертає?
12. Чому Fetch не відхиляє Promise при отриманні відповіді зі статусом 404 або 500?

13. Як правильно перевірити чи був запит успішним при використанні Fetch?
14. Які методи об'єкта Response використовуються для читання тіла відповіді і чим вони відрізняються?
15. Які заголовки потрібно передати при POST-запиті з JSON-тілом?
16. Як передати токен авторизації у Fetch-запиті?
17. Яка різниця між мережевою помилкою і HTTP-помилкою у контексті Fetch?
18. Як обробити обидва типи помилок в одній async-функції?
19. Що таке AbortController і яку задачу він вирішує? Наведіть практичний приклад.
20. Як реалізувати відображення стану завантаження та помилки при отриманні даних з сервера?

# Тема 15. Інструменти збірки та середовище розробки.

**Мета:** сформувати розуміння сучасного середовища JavaScript-розробки, навчити працювати з пакетним менеджером npm, читати та налаштовувати package.json, створювати і запускати проєкт через Vite у режимі розробки та збирати його для продакшну.

**План**

1. Поняття пакетного менеджера. npm.
2. Файл package.json.
3. npm-скрипти.
4. Структура сучасного JS-проєкту.
5. Vite: встановлення та налаштування.
6. Режим розробки.
7. Продакшн-збірка.

## 1. Поняття пакетного менеджера. npm

**Пакетний менеджер** – інструмент для встановлення, оновлення та видалення бібліотек і залежностей проєкту. Замість ручного завантаження файлів пакетний менеджер автоматично знаходить потрібну версію бібліотеки та всі її залежності.

**npm (Node Package Manager)** – стандартний пакетний менеджер для JavaScript. Встановлюється разом з Node.js.

**Перевірка встановлення:**

bash

```
node --version   # v20.0.0
npm --version    # 10.0.0
```

Node.js завантажується з офіційного сайту nodejs.org. Рекомендується версія LTS (Long Term Support).

**Основні команди npm:**

bash

```
# Ініціалізація проєкту – створює package.json
npm init
npm init -y      # з відповідями за замовчуванням
# Встановлення пакету
npm install lodash          # встановити пакет
npm install -D vite         # встановити як devDependency
npm install -g nodemon      # глобальне встановлення
```

```
# Скорочений запис
npm i lodash
npm i -D vite
# Видалення пакету
npm uninstall lodash
# Оновлення пакетів
npm update
npm update lodash
# Встановлення всіх залежностей з package.json
npm install
# Перегляд встановлених пакетів
npm list
npm list --depth=0   # лише верхній рівень
```

**Реєстр пакетів** – **npmjs.com** – публічний репозиторій де зберігаються всі пакети. На момент 2024 року містить понад 2 мільйони пакетів.

**2. Файл package.json**

**package.json** – головний конфігураційний файл проєкту. Описує проєкт, його залежності та команди запуску.

**Приклад package.json:**

```
json
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "Мій перший JS-проєкт",
  "author": "Іван Петренко",
  "license": "MIT",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "axios": "^1.6.0"
  },
  "devDependencies": {
    "vite": "^5.0.0"
```

```
  }
}
```

**Ключові поля:**

name – назва проєкту. Лише малі літери, без пробілів.

version – версія за стандартом semver: MAJOR.MINOR.PATCH.

scripts – команди що виконуються через npm run.

dependencies – бібліотеки необхідні для роботи у продакшні.

devDependencies – інструменти потрібні лише під час розробки: збірники, лінтери, тестові фреймворки.

**Версіонування пакетів (semver):**

```
json
"lodash": "4.17.21"    /* точна версія */
"lodash": "^4.17.21"   /* ^ – сумісні оновлення (4.x.x) */
"lodash": "~4.17.21"   /* ~ – лише патч-оновлення (4.17.x) */
"lodash": "*"          /* будь-яка версія – не рекомендується */
```

**package-lock.json** – автоматично генерується npm. Фіксує точні версії всіх встановлених пакетів включно із залежностями залежностей. Забезпечує однаковий результат встановлення на будь-якій машині. Не редагується вручну, має бути у системі контролю версій.

**node_modules** – папка з встановленими пакетами. Не додається до системи контролю версій – додається до .gitignore:

```
# .gitignore
node_modules/
dist/
.env
```

### 3. npm-скрипти

Скрипти у package.json дозволяють визначити та запускати команди через npm run:

```
json
{
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "lint": "eslint src",
    "format": "prettier --write src"
  }
```

}

**Виконання скриптів:**

bash

npm run dev  # запуск сервера розробки

npm run build  # продакшн-збірка

npm run preview  # перегляд зібраного проєкту

npm run lint  # перевірка коду лінтером

Скрипти start і test – виняток, запускаються без run:

bash

npm start

npm test

**Ланцюжок скриптів:**

json

```
{
  "scripts": {
    "lint": "eslint src",
    "format": "prettier --write src",
    "check": "npm run lint && npm run format"
  }
}
```

**4. Структура сучасного JS-проєкту**

Типова структура проєкту на Vite:

```
my-project/
├── node_modules/       ← встановлені пакети (не в git)
├── public/             ← статичні файли (копіюються без змін)
│   └── favicon.ico
├── src/                ← вихідний код
│   ├── assets/         ← зображення, шрифти
│   ├── components/     ← компоненти
│   ├── styles/         ← CSS-файли
│   ├── utils/          ← допоміжні функції
│   ├── main.js         ← точка входу
│   └── style.css       ← глобальні стилі
├── index.html          ← головний HTML-файл
├── package.json        ← конфігурація проєкту
├── package-lock.json   ← зафіксовані версії
├── vite.config.js      ← конфігурація Vite
```

└── .gitignore ← виключення з git

**Папка public** – файли що копіюються у корінь збірки без обробки. Використовується для favicon, robots.txt, статичних зображень що не проходять через збірник.

**Папка src** – весь вихідний код проєкту. Vite обробляє файли з цієї папки: трансформує, об'єднує, мінімізує.

**index.html** – головна точка входу для Vite. На відміну від Webpack – HTML є первинним, а не JS:

*файл html*

```
<!DOCTYPE html>
<html lang="uk">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Мій проєкт</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

**Модульна система ES6+:**

*файл javascript*

```
/* src/utils/math.js */
export function sum(a, b) { return a + b; }
export function multiply(a, b) { return a * b; }
export const PI = 3.14159;
/* src/main.js */
import { sum, multiply } from './utils/math.js';
import './style.css';
console.log(sum(2, 3));
```

### 5. Vite: встановлення та налаштування

**Vite** – сучасний інструмент збірки для JavaScript-проєктів. Розроблений Еваном Ю (автором Vue.js). Ключові переваги:

- миттєвий старт сервера розробки
- гаряче оновлення модулів (HMR) без перезавантаження сторінки

- оптимізована продакшн-збірка через Rollup
- підтримка TypeScript, JSX, CSS-модулів з коробки

**Створення нового проєкту:**

bash

```
# Через npm create
npm create vite@latest my-project
# Вибір шаблону у діалозі:
# > Vanilla (чистий JS)
# > Vue
# > React
# > Preact
# > Lit
# > Svelte
# Для курсу обираємо Vanilla → JavaScript
cd my-project
npm install
npm run dev
```

**Або одразу з параметрами:**

bash

```
npm create vite@latest my-project -- --template vanilla
cd my-project
npm install
```

**Файл vite.config.js:**

*файл javascript*

```
import { defineConfig } from 'vite';
export default defineConfig({
  /* Базовий шлях для розгортання */
  base: '/',
  /* Папка з вихідним кодом */
  root: '.',
  /* Налаштування сервера розробки */
  server: {
    port: 3000,        /* порт за замовчуванням 5173 */
    open: true,        /* автоматично відкрити браузер */
  },
  /* Налаштування збірки */
  build: {
```

```
    outDir: 'dist',    /* папка збірки */
    minify: true,      /* мінімізація */
  }
});
```

**6. Режим розробки**

bash

npm run dev

Після запуску Vite виводить адресу локального сервера:

```
 VITE v5.0.0  ready in 300 ms
  ➜ Local:   http://localhost:5173/
  ➜ Network: http://192.168.1.1:5173/
```

**Можливості режиму розробки:**

**HMR (Hot Module Replacement)** – при зміні файлу браузер оновлює лише змінений модуль без перезавантаження сторінки. Зберігається стан застосунку.

**Native ES Modules** – Vite у режимі розробки не збирає файли у бандл. Браузер отримує модулі напряму через import. Це забезпечує миттєвий старт незалежно від розміру проєкту.

**Обробка ресурсів:**

*файл javascript*

```
/* CSS імпортується напряму у JS */
import './style.css';
import styles from './module.module.css'; /* CSS-модулі */
/* Зображення */
import logo from './assets/logo.svg';
document.querySelector('#logo').src = logo;
/* JSON */
import data from './data.json';
console.log(data);
```

**Змінні середовища:**

bash

```
# Файл .env
VITE_API_URL=https://api.example.com
VITE_APP_TITLE=Мій проєкт
```

javascript

```
/* Доступ у коді */
console.log(import.meta.env.VITE_API_URL);
```

console.log(import.meta.env.MODE); /* 'development' або 'production' */
Лише змінні з префіксом VITE_ доступні у клієнтському коді.

**7. Продакшн-збірка**

bash
npm run build
Vite збирає проєкт у папку dist/:

```
dist/
├── assets/
│   ├── index-Abc123.js    ← зібраний і мінімізований JS
│   └── index-Xyz789.css   ← зібраний і мінімізований CSS
└── index.html             ← HTML з оновленими посиланнями
```

**Що відбувається під час збірки:**

- об'єднання модулів у бандли (через Rollup)
- мінімізація JS і CSS
- хешування імен файлів для кешування браузером
- оптимізація зображень
- tree-shaking – видалення невикористаного коду

**Перегляд збірки локально:**

bash
npm run preview
# Запускає локальний сервер для dist/
# http://localhost:4173/
preview дозволяє перевірити продакшн-збірку до розгортання на сервері.

**Аналіз розміру збірки:**

bash
npm i -D rollup-plugin-visualizer
javascript
/* vite.config.js */
import { visualizer } from 'rollup-plugin-visualizer';

export default defineConfig({
  plugins: [
    visualizer({ open: true }) /* відкриє звіт у браузері після збірки */
  ]
});

**Розгортання (deploy):**

Папка dist готова до розгортання на будь-якому статичному хостингу:

```
bash
# GitHub Pages – через gh-pages
npm i -D gh-pages
# package.json
"scripts": {
  "deploy": "npm run build && gh-pages -d dist"
}
npm run deploy
```

**Порівняння режимів:**

| | **Розробка (dev)** | **Продакшн (build)** |
|---|---|---|
| Швидкість старту | Миттєва | Потребує збірки |
| Бандлінг | Ні (native ESM) | Так (Rollup) |
| Мінімізація | Ні | Так |
| HMR | Так | Ні |
| Source maps | Так | Опційно |
| Оптимізація | Ні | Так |

## Контрольні запитання

1. Що таке пакетний менеджер і яку задачу він вирішує?
2. Яка різниця між локальним та глобальним встановленням пакету? Коли використовується кожне?
3. Яка різниця між dependencies і devDependencies? Наведіть приклади пакетів для кожного.
4. Що таке package.json і які ключові поля він містить?
5. Що таке package-lock.json і навіщо він потрібен?
6. Чому папка node_modules не додається до системи контролю версій?
7. Що означають символи ^ і ~ перед версією пакету у package.json?
8. Як встановити всі залежності проєкту після клонування репозиторію?
9. Що таке npm-скрипти і як їх запустити?
10. Яка різниця між npm run start і npm run dev?
11. Яка різниця між папками public і src у структурі Vite-проєкту?

12. Яку роль відіграє файл index.html у Vite-проєкті?
13. Як створити новий Vite-проєкт з шаблоном Vanilla JavaScript?
14. Що таке HMR і яку перевагу він дає під час розробки?
15. Чому сервер розробки Vite запускається значно швидше ніж Webpack?
16. Що таке змінні середовища у Vite? Чому назви змінних мають починатись з VITE_?
17. Що відбувається під час виконання команди npm run build?
18. Що таке tree-shaking і яку користь він приносить?
19. Навіщо хешуються імена файлів у продакшн-збірці?
20. Яка різниця між командами npm run build і npm run preview?

# Тема 16. Огляд бібліотек та фреймворків.

**Мета:** сформувати розуміння різниці між бібліотекою та фреймворком, ознайомити з екосистемою сучасних JS-інструментів, навчити аналізувати та порівнювати React, Vue і Angular за ключовими характеристиками і обирати інструмент відповідно до задачі.

**План**

1. Поняття бібліотеки та фреймворку.
2. Чому виникли фреймворки.
3. Огляд React.
4. Огляд Vue.
5. Огляд Angular.
6. Порівняння та критерії вибору.

## 1. Поняття CSS Grid та його роль у верстці

## 1. Поняття бібліотеки та фреймворку

**Бібліотека** – набір готових функцій та інструментів що вирішують конкретні задачі. Розробник сам вирішує коли і як використовувати бібліотеку. Бібліотека не диктує структуру проєкту.

*файл javascript*

```
/* Приклад використання бібліотеки lodash */
import _ from 'lodash';
_.groupBy([1, 2, 3, 4, 5], n => n % 2 === 0 ? 'парні' : 'непарні');
/* { непарні: [1, 3, 5], парні: [2, 4] } */
```

**Фреймворк** – комплексна платформа що диктує архітектуру, структуру та підхід до розробки. Фреймворк «керує» кодом розробника через інверсію контролю.

**Ключова різниця – інверсія контролю:**

Бібліотека: Ваш код → викликає → Бібліотеку

Фреймворк: Фреймворк → викликає → Ваш код

| Критерій | Бібліотека | Фреймворк |
|---|---|---|
| Контроль | У розробника | У фреймворку |
| Структура проєкту | Вільна | Визначена |

| | | |
|---|---|---|
| Гнучкість | Висока | Нижча |
| Поріг входу | Нижчий | Вищий |
| Приклади | lodash, axios, dayjs | Angular, NestJS |

**Особливе місце React** – формально є бібліотекою для побудови інтерфейсів, але з екосистемою роутингу, стану та збірки поводиться як фреймворк. Часто називається «бібліотекою з фреймворковим мисленням».

## 2. Чому виникли фреймворки

До появи фреймворків інтерактивні інтерфейси будувались через пряму маніпуляцію DOM. При зростанні складності це призводило до проблем.

**Проблеми чистого JS при побудові інтерфейсів:**

*файл javascript*

```
/* Ручна синхронізація даних і DOM – джерело помилок */
let count = 0;
function increment() {
  count++;
  document.querySelector('#counter').textContent = count;
  document.querySelector('#status').textContent = count > 0 ? 'Активно' : 'Неактивно';
  document.querySelector('#btn').disabled = count >= 10;
  /* ... ще 10 місць де потрібно оновити DOM */
}
```

**Що вирішують фреймворки:**

- **Реактивність** – UI автоматично оновлюється при зміні даних.
- **Компонентний підхід** – інтерфейс розбивається на незалежні частини.
- **Маршрутизація** – навігація без перезавантаження сторінки.
- **Управління станом** – централізоване сховище даних застосунку.
- **Стандартизація** – єдина структура коду у команді.

## 3. Огляд React

**React** – бібліотека для побудови користувацьких інтерфейсів. Розроблена Facebook (Meta) у 2013 році. Найпопулярніший інструмент для фронтенд-розробки станом на 2024 рік.

**Ключові концепції:**

**Компоненти** – основна одиниця React-застосунку. Кожен компонент – це функція що повертає JSX:

```jsx
function UserCard({ name, email, avatar }) {
  return (
    <div className="card">
      <img src={avatar} alt={name} />
      <h2>{name}</h2>
      <p>{email}</p>
    </div>
  );
}
```

**JSX** – синтаксичне розширення JavaScript що дозволяє писати HTML-подібний код у JS. Компілюється у виклики React.createElement():

```jsx
/* JSX */
const element = <h1 className="title">Привіт, {name}!</h1>;

/* Що генерує компілятор */
const element = React.createElement('h1', { className: 'title' }, `Привіт, ${name}!`);
```

**Стан (state)** – внутрішні дані компонента. При зміні стану компонент перерендерюється:

```jsx
import { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Лічильник: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
    </div>
  );
}
```

**Props** – дані що передаються від батьківського компонента до дочірнього. Незмінні всередині дочірнього компонента:

```jsx
/* Батьківський компонент передає props */
function App() {
  return <UserCard name="Іван" email="ivan@ua.com" />;
}
/* Дочірній отримує через параметр */
function UserCard({ name, email }) {
  return <div>{name} – {email}</div>;
}
```

**Virtual DOM** – React підтримує віртуальне представлення DOM у пам'яті. При зміні стану порівнює нове і попереднє дерево та оновлює лише змінені частини реального DOM.

**Екосистема React:**

| Задача | Рішення |
|---|---|
| Маршрутизація | React Router |
| Стан | Redux, Zustand, Context API |
| Запити до API | TanStack Query |
| Форми | React Hook Form |
| UI-компоненти | shadcn/ui, MUI, Ant Design |
| Фреймворк на основі React | Next.js |

**Характеристики:**

- Розробник: Meta (Facebook)
- Перший реліз: 2013
- Підхід: компонентний, однонаправлений потік даних
- Мова: JavaScript / TypeScript
- Розмір: ~45KB (мінімізований)

## 4. Огляд Vue

**Vue** – прогресивний фреймворк для побудови інтерфейсів. Розроблений Еваном Ю у 2014 році. Позиціонується як найпростіший у вивченні серед трьох.

**Ключові концепції:**

**Однофайлові компоненти (SFC)** – весь код компонента в одному .vue файлі:

```
vue
<template>
  <div class="card">
    <h2>{{ user.name }}</h2>
    <p>{{ user.email }}</p>
    <button @click="greet">Привітати</button>
  </div>
</template>
<script setup>
import { ref } from 'vue';
const props = defineProps(['user']);
function greet() {
  alert(`Привіт, ${props.user.name}!`);
}
</script>
<style scoped>
.card { border: 1px solid #ddd; padding: 16px; }
</style>
```

**Реактивність** – Vue автоматично відстежує залежності та оновлює DOM:

```
javascript
import { ref, computed } from 'vue';
const count = ref(0);
const doubled = computed(() => count.value * 2);
count.value++; /* Vue автоматично оновить все що залежить від count */
```

**Директиви** – спеціальні атрибути з префіксом v-:

```
vue
<template>
  <!-- Умовний рендеринг -->
  <p v-if="isLoggedIn">Вітаємо!</p>
  <p v-else>Увійдіть будь ласка</p>
  <!-- Рендеринг списку -->
  <ul>
    <li v-for="item in items" :key="item.id">{{ item.name }}</li>
```

```
  </ul>
  <!-- Двостороннє прив'язування -->
  <input v-model="username" placeholder="Ім'я">
  <p>{{ username }}</p>
</template>
```

**Екосистема Vue:**

| Задача | Рішення |
|---|---|
| Маршрутизація | Vue Router |
| Стан | Pinia |
| Запити до API | TanStack Query для Vue |
| UI-компоненти | Vuetify, PrimeVue, Quasar |
| Фреймворк на основі Vue | Nuxt.js |

**Характеристики:**

- Розробник: Еван Ю та спільнота.
- Перший реліз: 2014.
- Підхід: компонентний, двостороннє прив'язування даних.
- Мова: JavaScript / TypeScript.
- Розмір: ~33KB (мінімізований).

## 5. Огляд Angular

**Angular** – повноцінний фреймворк для розробки великих корпоративних застосунків. Розроблений Google у 2016 році (переписаний з AngularJS). Використовує TypeScript як основну мову.

**Ключові концепції:**

**Модулі та компоненти:**

typescript

```
/* app.component.ts */
import { Component } from '@angular/core';
ponent({
  selector: 'app-user-card',
  template: `
    <div class="card">
```

```
      <h2>{{ user.name }}</h2>
      <p>{{ user.email }}</p>
      <button (click)="greet()">Привітати</button>
    </div>
  `
})
export class UserCardComponent {
  user = { name: 'Іван', email: 'ivan@ua.com' };
  greet() {
    alert(`Привіт, ${this.user.name}!`);
  }
}
```

**Dependency Injection** – вбудований механізм впровадження залежностей:

```
typescript
/* Сервіс */
@Injectable({ providedIn: 'root' })
export class UserService {
  getUsers() {
    return this.http.get('/api/users');
  }
}
/* Компонент отримує сервіс автоматично */
export class UserListComponent {
  constructor(private userService: UserService) {}
}
```

**Вбудовані інструменти Angular:**

| Задача | Рішення |
|---|---|
| Маршрутизація | Angular Router (вбудований) |
| HTTP | HttpClient (вбудований) |
| Форми | Reactive Forms (вбудований) |
| Стан | NgRx |

| | |
|---|---|
| UI-компоненти | Angular Material |
| Тестування | Jasmine + Karma (вбудовані) |

**Характеристики:**

- Розробник: Google.
- Перший реліз: 2016.
- Підхід: компонентний, MVC, двостороннє прив'язування.
- Мова: TypeScript (обов'язково).
- Розмір: ~180KB (мінімізований).

## 6. Порівняння та критерії вибору

**Загальне порівняння:**

| Критерій | React | Vue | Angular |
|---|---|---|---|
| Тип | Бібліотека | Фреймворк | Фреймворк |
| Розробник | Meta | Спільнота | Google |
| Мова | JS / TS | JS / TS | TypeScript |
| Поріг входу | Середній | Низький | Високий |
| Гнучкість | Висока | Середня | Низька |
| Розмір | Малий | Малий | Великий |
| Вбудовані інструменти | Мінімум | Середньо | Максимум |
| Популярність (2024) | ★★★★★ | ★★★★ | ★★★ |
| Ринок праці | Найвищий | Середній | Середній |

**Коли обирати React:**

- потрібна максимальна гнучкість у виборі інструментів;
- проєкт різного масштабу – від лендінгу до великого застосунку;
- важливий найбільший ринок праці;
- команда знає JavaScript.

**Коли обирати Vue:**

- для початківців або швидкого старту;
- невелика команда або соло-розробник;
- потрібна чітка структура без зайвої складності;
- проєкт малого або середнього розміру.

**Коли обирати Angular:**

- великий корпоративний проєкт з великою командою;
- потрібна жорстка стандартизація підходів;
- команда знає TypeScript і ООП;
- важлива вбудована підтримка тестування і DI.

**Тенденції та реальність:**

Популярність за даними Stack Overflow Developer Survey 2024:
React: 40.6% (використовують)
Angular: 17.1%
Vue: 15.4%
Популярність вакансій (Україна та ЄС):
React: ~60%
Vue: ~20%
Angular: ~20%

**Спільні риси всіх трьох:**

- компонентний підхід;
- реактивне оновлення UI;
- підтримка TypeScript;
- CLI для генерації проєкту;
- велика спільнота та екосистема;
- підтримка SSR (серверного рендерингу).

## Контрольні запитання

1. Що таке бібліотека і що таке фреймворк? У чому принципова різниця між ними?
2. Що таке інверсія контролю і як вона проявляється у фреймворках?
3. Чому при зростанні складності застосунку пряма маніпуляція DOM стає проблемою?
4. Які чотири основні проблеми вирішують сучасні JS-фреймворки?
5. Що таке компонентний підхід і яку перевагу він дає при розробці інтерфейсів?
6. Що таке JSX у React і у що він компілюється?
7. Що таке стан (state) у React і що відбувається при його зміні?

8. Яка різниця між state і props у React?
9. Що таке Virtual DOM і яку перевагу він дає?
10. Що таке однофайловий компонент у Vue і з яких трьох секцій він складається?
11. Що таке директиви у Vue? Назвіть три директиви і поясніть призначення кожної.
12. Що таке двостороннє прив'язування даних і яка директива Vue його реалізує?
13. Чим підхід Angular до розробки відрізняється від React і Vue?
14. Що таке Dependency Injection у Angular і яку задачу він вирішує?
15. Чому Angular використовує TypeScript як обов'язкову мову?
16. Порівняйте React, Vue і Angular за порогом входу та гнучкістю.
17. Які вбудовані інструменти має Angular на відміну від React?
18. У якому випадку доцільно обрати Vue замість React?
19. У якому випадку доцільно обрати Angular?
20. Який фреймворк має найбільший попит на ринку праці і чому це важливо при виборі технології для вивчення?

# Тема 17. Основи React.

**Мета:** сформувати практичні навички розробки React-застосунків – створювати функційні компоненти, передавати дані через props, керувати станом через useState, обробляти події, використовувати useEffect для побічних ефектів та отримувати дані з сервера.

**План**

1. Компонентний підхід. Створення React-проєкту.
2. JSX: синтаксис та особливості.
3. Умовний рендеринг та рендеринг списків.
4. Функційні компоненти та props.
5. Стан компонента: хук useState.
6. Обробка подій у React.
7. Хук useEffect.
8. Отримання даних із сервера.

## 1. Компонентний підхід. Створення React-проєкту

**Компонентний підхід** – інтерфейс розбивається на незалежні частини що називаються компонентами. Кожен компонент відповідає за власну частину UI, має власні дані та логіку.

App

├── Header

│ ├── Logo

│ └── Navigation

├── Main

│ ├── SearchBar

│ └── ProductList

│ ├── ProductCard

│ ├── ProductCard

│ └── ProductCard

└── Footer

Переваги компонентного підходу:

- повторне використання компонентів;
- незалежна розробка та тестування;
- чіткий розподіл відповідальності;
- легке масштабування.

**Створення проєкту:**

bash

```
npm create vite@latest my-react-app -- --template react
cd my-react-app
npm install
npm run dev
```

**Структура React-проєкту на Vite:**

```
my-react-app/
├── src/
│   ├── components/      ← компоненти
│   ├── assets/          ← зображення та інші ресурси
│   ├── App.jsx          ← кореневий компонент
│   ├── App.css
│   ├── main.jsx         ← точка входу
│   └── index.css
├── index.html
├── package.json
└── vite.config.js
```

**Точка входу – main.jsx:**

jsx

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import './index.css';
import App from './App.jsx';
```

```
createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

**Кореневий компонент – App.jsx:**

```
jsx
import './App.css';

function App() {
  return (
    <div className="app">
      <h1>Мій React-застосунок</h1>
    </div>
  );
}
export default App;
```

## 2. JSX: синтаксис та особливості

**JSX** – розширення синтаксису JavaScript що дозволяє писати HTML-подібний код у JS-файлах.

**Основні правила JSX:**

**Один кореневий елемент** – компонент повертає один кореневий елемент:

```
jsx
/* Помилка – два кореневих елементи */
return (
  <h1>Заголовок</h1>
  <p>Текст</p>
);
/* Правильно – обгортка div */
return (
  <div>
    <h1>Заголовок</h1>
    <p>Текст</p>
  </div>
);
```

```
/* Правильно – Fragment (не додає зайвого вузла до DOM) */
return (
  <>
    <h1>Заголовок</h1>
    <p>Текст</p>
  </>
);
```

**className замість class:**

```
jsx
<div className="container">
  <p className="text-primary">Текст</p>
</div>
```

**Вирази у фігурних дужках:**

```
jsx
const name = 'Іван';
const price = 299;
return (
  <div>
    <h2>{name}</h2>
    <p>Ціна: {price} грн</p>
    <p>Знижка: {price * 0.9} грн</p>
    <p>{new Date().getFullYear()}</p>
  </div>
);
```

**Атрибути у JSX:**

```
jsx
/* Рядкові атрибути */
<img src="photo.jpg" alt="Фото" />
/* Динамічні атрибути через {} */
<img src={user.avatar} alt={user.name} />
<button disabled={isLoading}>Надіслати</button>
<input type="text" value={inputValue} />
/* Стилі як об'єкт */
<div style={{ color: 'red', fontSize: '18px', marginTop: '16px' }}>
  Текст
</div>
```

**Самозакривні теги – обов'язкові:**

jsx

```
<img src="photo.jpg" alt="Фото" />
<input type="text" />
<br />
<hr />
```

### 3. Умовний рендеринг та рендеринг списків

**Умовний рендеринг через тернарний оператор:**

jsx

```
function UserGreeting({ isLoggedIn, username }) {
  return (
    <div>
      {isLoggedIn
        ? <h2>Привіт, {username}!</h2>
        : <h2>Будь ласка, увійдіть</h2>
      }
    </div>
  );
}
```

**Умовний рендеринг через &&:**

jsx

```
function Notification({ hasMessages, count }) {
  return (
    <div>
      <h1>Пошта</h1>
      {hasMessages && <p>У вас {count} нових повідомлень</p>}
    </div>
  );
}
```

**Умовне повернення з функції:**

jsx

```
function LoadingSpinner({ isLoading, data }) {
  if (isLoading) return <p>Завантаження...</p>;
  if (!data) return <p>Дані відсутні</p>;
  return (
    <ul>
      {data.map(item => <li key={item.id}>{item.name}</li>)}
```

```
    </ul>
  );
}
```

**Рендеринг списків через map:**

```
jsx
function ProductList({ products }) {
  return (
    <ul className="product-list">
      {products.map(product => (
        <li key={product.id} className="product-item">
          <h3>{product.name}</h3>
          <p>{product.price} грн</p>
        </li>
      ))}
    </ul>
  );
}
```

**Атрибут key – обов'язковий при рендерингу списків.** Допомагає React визначити які елементи змінились. Має бути унікальним серед сусідів. Не використовувати індекс масиву якщо список змінюється:

```
jsx
/* Погано – індекс як key */
{items.map((item, index) => <li key={index}>{item.name}</li>)}
/* Добре – унікальний id */
{items.map(item => <li key={item.id}>{item.name}</li>)}
```

**Рендеринг з фільтрацією:**

```
jsx
function ActiveUsers({ users }) {
  return (
    <ul>
      {users
        .filter(user => user.isActive)
        .map(user => (
          <li key={user.id}>{user.name}</li>
        ))
      }
    </ul>
```

```
  );
}
```

**4. Функційні компоненти та props**

**Функційний компонент** – звичайна JS-функція що приймає props і повертає JSX. Назва починається з великої літери.

```
jsx
/* Базовий компонент */
function Button({ label, onClick, variant = 'primary' }) {
  return (
    <button
      className={`btn btn-${variant}`}
      onClick={onClick}
    >
      {label}
    </button>
  );
}
```

**Props** – об'єкт з даними що передаються від батька до дитини. Незмінні всередині компонента:

```
jsx
/* Батьківський компонент */
function App() {
  return (
    <div>
      <UserCard
        name="Іван Петренко"
        email="ivan@ua.com"
        avatar="https://example.com/avatar.jpg"
        isAdmin={true}
      />
    </div>
  );
}
/* Дочірній компонент */
function UserCard({ name, email, avatar, isAdmin = false }) {
  return (
    <div className="user-card">
```

```
      <img src={avatar} alt={name} />
      <h2>{name}</h2>
      <p>{email}</p>
      {isAdmin && <span className="badge">Адмін</span>}
    </div>
  );
}
```

**Передача функцій через props:**

```jsx
function App() {
  function handleDelete(id) {
    console.log('Видалити:', id);
  }
  return <ItemList onDelete={handleDelete} />;
}
function ItemList({ onDelete }) {
  return (
    <ul>
      <li>
        Елемент 1
        <button onClick={() => onDelete(1)}>Видалити</button>
      </li>
    </ul>
  );
}
```

**Prop children – вміст між тегами компонента:**

```jsx
function Card({ title, children }) {
  return (
    <div className="card">
      <h2 className="card__title">{title}</h2>
      <div className="card__content">
        {children}
      </div>
    </div>
  );
}
```

```
/* Використання */
<Card title="Заголовок">
  <p>Будь-який вміст</p>
  <button>Кнопка</button>
</Card>
```

## 5. Стан компонента: хук useState

**Хук** – спеціальна функція React що додає можливості до функційних компонентів. Починається з use.

**useState** – хук для оголошення стану компонента:

```
jsx
import { useState } from 'react';
const [state, setState] = useState(initialValue);
/* state – поточне значення */
/* setState – функція для зміни стану */
/* initialValue – початкове значення */
```

**Простий лічильник:**

```
jsx
import { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Значення: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
      <button onClick={() => setCount(0)}>Скинути</button>
    </div>
  );
}
```

**Стан з рядком:**

```
jsx
function SearchInput() {
  const [query, setQuery] = useState('');
  return (
    <div>
      <input
```

```
        type="text"
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder="Пошук..."
      />
      <p>Запит: {query}</p>
    </div>
  );
}
```

**Стан з об'єктом:**

```
jsx
function UserForm() {
  const [user, setUser] = useState({ name: '', email: '' });
  function handleChange(e) {
    const { name, value } = e.target;
    setUser(prev => ({ ...prev, [name]: value }));
    /* spread зберігає інші поля об'єкта */
  }
  return (
    <form>
      <input name="name" value={user.name} onChange={handleChange} />
      <input name="email" value={user.email} onChange={handleChange} />
    </form>
  );
}
```

**Стан з масивом:**

```
jsx
function TodoList() {
  const [todos, setTodos] = useState([]);
  const [input, setInput] = useState('');
  function addTodo() {
    if (!input.trim()) return;
    setTodos(prev => [...prev, { id: Date.now(), text: input, done: false }]);
    setInput('');
  }
  function removeTodo(id) {
    setTodos(prev => prev.filter(todo => todo.id !== id));
```

```
  }

  function toggleTodo(id) {
   setTodos(prev =>
    prev.map(todo =>
     todo.id === id ? { ...todo, done: !todo.done } : todo
    )
   );
  }
  return (
   <div>
    <div>
     <input value={input} onChange={e => setInput(e.target.value)} />
     <button onClick={addTodo}>Додати</button>
    </div>
    <ul>
     {todos.map(todo => (
      <li key={todo.id}>
       <span
        style={{ textDecoration: todo.done ? 'line-through' : 'none' }}
        onClick={() => toggleTodo(todo.id)}
       >
        {todo.text}
       </span>
       <button onClick={() => removeTodo(todo.id)}>✕</button>
      </li>
     ))}
    </ul>
   </div>
  );
}
```

**Важливі правила useState:**

- не змінювати стан напряму – лише через setState;
- setState асинхронний – новий стан доступний на наступному рендері;
- для об'єктів і масивів – завжди створювати нову копію.

## 6. Обробка подій у React

**Синтаксис обробників подій:**

jsx

```
/* Інлайн стрілкова функція */
<button onClick={() => console.log('Клік')}>Натисни</button>
/* Посилання на функцію */
function handleClick() {
  console.log('Клік');
}
<button onClick={handleClick}>Натисни</button>
/* З аргументом */
<button onClick={() => handleDelete(item.id)}>Видалити</button>
```

**Об'єкт події:**

jsx

```
function InputField() {
  function handleChange(event) {
    console.log(event.target.value);    /* поточне значення */
    console.log(event.target.name);     /* атрибут name */
    console.log(event.target.checked);  /* для checkbox */
  }
  function handleSubmit(event) {
    event.preventDefault(); /* запобігає перезавантаженню сторінки */
    console.log('Форма відправлена');
  }
  return (
    <form onSubmit={handleSubmit}>
      <input onChange={handleChange} />
      <button type="submit">Відправити</button>
    </form>
  );
}
```

**Часто вживані події React:**

jsx

```
onClick         /* клік */
onDoubleClick    /* подвійний клік */
onChange         /* зміна значення input, select, textarea */
onInput         /* введення символу */
onSubmit         /* відправка форми */
```

```
onFocus         /* фокус на елементі */
onBlur          /* втрата фокусу */
onKeyDown       /* натискання клавіші */
onMouseOver     /* наведення миші */
```

**Форма з контрольованими полями:**

```
jsx
function LoginForm() {
  const [form, setForm] = useState({ email: '', password: '' });
  const [error, setError] = useState('');
  function handleChange(e) {
    setForm(prev => ({ ...prev, [e.target.name]: e.target.value }));
    setError('');
  }
  function handleSubmit(e) {
    e.preventDefault();
    if (!form.email || !form.password) {
      setError('Заповніть всі поля');
      return;
    }
    console.log('Відправка:', form);
  }
  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        name="email"
        value={form.email}
        onChange={handleChange}
        placeholder="Email"
      />
      <input
        type="password"
        name="password"
        value={form.password}
        onChange={handleChange}
        placeholder="Пароль"
      />
```

```
      {error && <p className="error">{error}</p>}
      <button type="submit">Увійти</button>
    </form>
  );
}
```

### 7. Хук useEffect

**useEffect** – хук для виконання побічних ефектів: запити до API, підписки на події, робота з DOM, таймери.

**Синтаксис:**

```
jsx
import { useEffect } from 'react';
useEffect(() => {
  /* побічний ефект */
  return () => {
    /* функція очищення (опційно) */
  };
}, [залежності]);
```

**Масив залежностей визначає коли виконується ефект:**

```
jsx
/* Без масиву – після кожного рендеру */
useEffect(() => {
  console.log('Рендер');
});
/* Порожній масив – лише після першого рендеру */
useEffect(() => {
  console.log('Компонент змонтовано');
}, []);

/* З залежностями – при зміні вказаних значень */
useEffect(() => {
  console.log('count змінився:', count);
}, [count]);
```

**Функція очищення:**

```
jsx
function Timer() {
  const [time, setTime] = useState(0);
  useEffect(() => {
```

```
    const interval = setInterval(() => {
      setTime(prev => prev + 1);
    }, 1000);
    /* Очищення – викликається при розмонтуванні компонента */
    return () => clearInterval(interval);
  }, []);
  return <p>Час: {time} сек</p>;
}
```

**Підписка та відписка від подій:**

```
jsx
function WindowSize() {
  const [size, setSize] = useState({
    width: window.innerWidth,
    height: window.innerHeight
  });
  useEffect(() => {
    function handleResize() {
      setSize({ width: window.innerWidth, height: window.innerHeight });
    }
    window.addEventListener('resize', handleResize);
    return () => window.removeEventListener('resize', handleResize);
  }, []);
  return <p>{size.width} × {size.height}</p>;
}
```

## 8. Отримання даних із сервера

**Базовий патерн отримання даних:**

```
jsx
import { useState, useEffect } from 'react';
function UserList() {
  const [users, setUsers] = useState([]);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);
  useEffect(() => {
    async function fetchUsers() {
      try {
        setIsLoading(true);
```

```
        const response = await fetch('https://jsonplaceholder.typicode.com/users');

        if (!response.ok) throw new Error(`Помилка: ${response.status}`);
        const data = await response.json();
        setUsers(data);
      } catch (err) {
        setError(err.message);
      } finally {
        setIsLoading(false);
      }
    }
    fetchUsers();
  }, []);

  if (isLoading) return <p>Завантаження...</p>;
  if (error) return <p>Помилка: {error}</p>;

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name} – {user.email}</li>
      ))}
    </ul>
  );
}
```

**Отримання даних з параметром:**

```
jsx
function UserPosts({ userId }) {
  const [posts, setPosts] = useState([]);
  const [isLoading, setIsLoading] = useState(false);
  useEffect(() => {
    if (!userId) return;
    async function fetchPosts() {
      setIsLoading(true);
      try {
        const response = await fetch(
```

```
        `https://jsonplaceholder.typicode.com/posts?userId=${userId}`
      );
      const data = await response.json();
      setPosts(data);
    } catch (err) {
      console.error(err);
    } finally {
      setIsLoading(false);
    }
  }
  fetchPosts();
}, [userId]); /* викликається при зміні userId */
if (isLoading) return <p>Завантаження постів...</p>;
return (
  <ul>
    {posts.map(post => (
      <li key={post.id}>
        <h3>{post.title}</h3>
        <p>{post.body}</p>
      </li>
    ))}
  </ul>
);
}
```

**Скасування запиту при розмонтуванні:**

```jsx
useEffect(() => {
  const controller = new AbortController();
  async function fetchData() {
    try {
      const response = await fetch('/api/data', {
        signal: controller.signal
      });
      const data = await response.json();
      setData(data);
    } catch (err) {
      if (err.name === 'AbortError') return; /* запит скасовано */
```

```
      setError(err.message);
    }
  }
  fetchData();
  return () => controller.abort(); /* скасування при розмонтуванні */
}, []);
```

**Повний приклад – пошук з API:**

```
jsx
function Search() {
  const [query, setQuery] = useState('');
  const [results, setResults] = useState([]);
  const [isLoading, setIsLoading] = useState(false);
  useEffect(() => {
    if (!query.trim()) {
      setResults([]);
      return;
    }
    const controller = new AbortController();
    async function search() {
      setIsLoading(true);
      try {
        const response = await fetch(
          `https://jsonplaceholder.typicode.com/users?q=${query}`,
          { signal: controller.signal }
        );
        const data = await response.json();
        setResults(data);
      } catch (err) {
        if (err.name !== 'AbortError') console.error(err);
      } finally {
        setIsLoading(false);
      }
    }
    const timer = setTimeout(search, 500); /* debounce */
    return () => {
      clearTimeout(timer);
      controller.abort();
```

```
    };
  }, [query]);

  return (
    <div>
      <input
        value={query}
        onChange={e => setQuery(e.target.value)}
        placeholder="Пошук..."
      />
      {isLoading && <p>Пошук...</p>}
      <ul>
        {results.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

**Контрольні запитання**

1. Що таке компонентний підхід і які переваги він дає при розробці інтерфейсів?
2. Яку роль відіграє файл main.jsx у React-застосунку?
3. Які основні правила синтаксису JSX? Назвіть не менше чотирьох.
4. Чому у JSX використовується className замість class?
5. Як у JSX передати динамічне значення атрибута? Наведіть приклад.
6. Які три способи умовного рендерингу існують у React? Наведіть приклад кожного.
7. Як виконується рендеринг списку у React? Який метод масиву використовується?
8. Навіщо потрібен атрибут key при рендерингу списків? Чому не рекомендується використовувати індекс масиву як key?
9. Що таке props і яке головне обмеження на їх використання всередині компонента?
10. Що таке prop children і як він використовується?
11. Як передати функцію від батьківського компонента до дочірнього через props? Навіщо це потрібно?

12. Що таке хук у React? Яке правило щодо місця виклику хуків?
13. Що повертає useState? Що відбувається при виклику функції оновлення стану?
14. Чому не можна змінювати стан напряму – наприклад state.count = 1?
15. Як правильно оновити об'єкт у стані не втративши інші поля?
16. Як правильно додати елемент до масиву у стані?
17. Яка різниця між onClick={handleClick} і onClick={() => handleClick()}? Коли використовується кожен запис?
18. За що відповідає масив залежностей у useEffect? Що відбувається якщо він порожній?
19. Навіщо потрібна функція очищення у useEffect? Наведіть приклад.
20. Який патерн використовується для отримання даних із сервера у React-компоненті? Які три стани зазвичай відстежуються?

## Рекомендована література

### *Основна:*


1. Бородкіна І.Л., Бородкін Г.О. Web-технології та web-дизайн: застосування мови HTML для створення електронних ресурсів. К.: Ліра-К, 2022. 212 с.
2. Елізабет Робсон, Ерік Фрімен. Head First. Програмування на JavaScript. Х.: Фабула, 2022. 672 с.
3. Мосіюк О.О. WEB-технології. Частина 1. Верстка. Житомир: Вид-во ЖДУ ім. Івана Франка, 2020. 56 с.
4. Пасічник В. В., Пасічник О. В., Угрин Д.І. Веб-технології: підручник. Львів : Магнолія, 2024. 336 с.
5. Пецько В.І., Беца А.С., Мазютинець Г.В., Міца О.В. Вступ до вебтехнологій: навчальний посібник для студентів спеціальності 122 - "Комп'ютерні науки". Ужгород: РІК-У, 2024. 252 с.
6. Юрчак І.Ю., Гузинець Н.В. Базові засади веб-розробки. Львів : Магнолія, 2023. 180 с.


### *Додаткова:*


1. Frain B. Responsive Web Design with HTML5 and CSS. 4th ed. Packt Publishing, 2022. 504 p.
2. Haverbeke M. Eloquent JavaScript. 4th ed. No Starch Press, 2024. 472 p.
3. McGrath M. HTML, CSS & JavaScript in easy steps. In Easy Steps Limited, 2020. 480 p.
4. Wieruch R. The Road to React. 2024 Edition. Self-published, 2024. 350 p.
5. Пасічник В. В., Пасічник О. В., Угрин Д. І. Веб-технології та веб-дизайн: підручник. Львів : Магнолія, 2021. Ч. 1. 336 с.


### *Інтернет- ресурси:*


1. CSS-Tricks. Повний посібник з Flexbox. URL: https://css-tricks.com/snippets/css/a-guide-to-flexbox.
2. CSS-Tricks. Повний посібник з Grid. URL: https://css-tricks.com/snippets/css/complete-guide-grid.
3. HTML. URL: html.spec.whatwg.org.
4. MDN Web Docs. URL: developer.mozilla.org.
5. React. Офіційна документація. URL: https://react.dev/.
6. Основи Web UI розробки 2023. URL:https://prometheus.org.ua/prometheus-free/web-ui-development-basics/.
7. Офіційна документація Vite. URL: https://vitejs.dev
8. Офіційна сторінка Codecademy URL: https://www.codecademy.com/
9. Офіційна сторінка W3C URL: https://www.w3.org.
10. Підручник з JavaScript. URL: https://javascript.info/.

Матеріали лекційного курсу

Укладачі:

Федорчук Анна Леонідівна

# «Вебтехнології та вебдизайн»

Конспект лекцій

Видавництво Житомирського державного університету імені Івана Франка
Свідоцтво суб'єкта видавничої справи:
серія ЖТ № 10 від 07.12.2004 р.
10008, м. Житомир, вул. Велика Бердичівська, 40