Міністерство освіти і науки України

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Основи клієнтської розробки

Лабораторний практикум

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для здобувачів ступеня бакалавра
спеціальності F2 Інженерія програмного забезпечення
спеціальності F6 Інформаційні системи та технології
спеціальності F7 Комп'ютерна інженерія

Укладачі: М. С. Хмелюк

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2025

УДК 004.43 (075.8)

Укладачі: *Хмелюк Марина Сергіївна*

Рецензент *Лісовиченко О.І.,* к.т.н., доцент кафедри інформатики та програмної інженерії КПІ ім. Ігоря Сікорського

Відповідальний редактор *Амонс О.А.,* к.т.н., доцент кафедри інформаційних систем та технологій КПІ ім. Ігоря Сікорського

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 5 від 05.03.2026р.)
за поданням вченої ради факультету Інформатики та обчислювальної техніки
(протокол № 7 від 23.02.2026 р.)

Основи клієнтської розробки. [Електронний ресурс] : лаб. практикум : навч. посіб. для здобувачів ступеня бакалавра за спец. F2 Інженерія програмного забезпечення, F6 Інформаційні системи та технології, F7 Комп'ютерна інженерія / КПІ ім. Ігоря Сікорського ; уклад.: М. С. Хмелюк. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2025. – 234 с.

Навчальний посібник охоплює теоретичний матеріал та практичні завдання, які необхідні для виконання лабораторного практикуму з дисципліни «Основи клієнтської розробки». Посібник містить роз'яснення щодо виконання лабораторного практикуму, передбаченого робочою програмою дисципліни «Основи клієнтської розробки». В посібнику викладено інформаційні матеріали та приклади виконання завдань: налаштування управління проєктом за допомогою системи контролю версій Git, створенням та наповненням вебсторінок, оформленням зовнішнього вигляду сторінок, написанням скриптів для управлінням елементами сайту. Посібник може бути корисним студентам відповідних спеціальностей при вивченні дисциплін, пов'язаних з розробкою програмного забезпечення.

УДК 004.43 (075.8)

Реєстр. № НП 25/26-231. Обсяг 11,7 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

ВСТУП

Клієнтська розробка (фронтенд) - це область розробки, яка відповідає за створення клієнтської частини вебзастосунку або програми, яку бачить користувач і з якою він взаємодіє: інтерфейс, кнопки, меню, форми, анімації та відображення даних на різних пристроях, забезпечуючи привабливість, зручність та коректне відображення.

Клієнтську розробку будь-якого вебзастосунку, програми можна розділити на три частини:

HTML (HyperText Markup Language) - мова гіпертекстової розмітки, яка використовується для структурування вебсторінок та їх вмісту, тобто, це код, який повідомляє браузеру, як відображати сторінку. За допомогою HTML можна прописати заголовки, абзаци, переліки, форми, таблиці, посилання та інші елементи сторінки.

CSS (Cascading Style Sheets) відповідає за візуальне оформлення сторінки: розташування блоків на сторінці, фон сторінки, шрифти, кольори та ін. Без використання стилів html-сторінка виглядає «бідно», тому HTML і CSS застосовуються разом. CSS-код -це код, який повідомляє браузеру, як саме відображати елементи сторінки.

JavaScript - мова програмування, за допомогою якої можна додати інтерактивності на сторінці, а JavaScript-код відповідає за логіку і функціонал. Спливаючі вікна, форми зворотного зв’язку, анімації, обробку подій (кліків мишки, переміщення курсору та інших дій користувача) на багато іншого, можна реалізувати до допомогою мови JavaScript.

При розробці програмного забезпечення, в тому числі і клієнтській, використовуються системи контролю версій, які дозволяють відстежувати, керувати змінами в коді, документувати, зберігати різні версії програми, перемикатись між ними, виявляти помилки та виправляти їх, виконувати

паралельну розробку за допомогою гілкування, забезпечувати роботу над проєктами в команді. Однією з таких систем є Git - розподілена система контролю версій та хмарна платформа GitHub для хостингу репозиторіїв.

Дисципліна «Основи клієнтської розробки» належить Вибіркових освітніх компонент спеціальностей: F2 Інженерія програмного забезпечення, F6 Інформаційні системи та технології, F7 Комп'ютерна інженерія. Вивчення дисципліни базується на знаннях, навичках та уміннях, отриманих під час вивчення студентами дисципліни «Програмування».

Основна мета вивчення дисципліни - формування та закріплення у студентів здатності до розуміння предметної області та професійної діяльності, здатності вчитися і оволодівати сучасними знаннями, обробленням та узагальненням інформації з різних джерел, здатності застосування системи контролю версій для управління проєктами, знань щодо розробки клієнтської частини вебзастосунків, створення їх за допомогою мови розмітки HTML, мови опису зовнішнього вигляду документу CSS та мови JavaScript – для управління вебсторінками.

ЗАГАЛЬНІ МЕТОДИЧНІ ВКАЗІВКИ

Посібник призначений для виконання комп1ютерних практикумів з дисципліни «Основи клієнтської розробки». Він складаються з 8 тем комп'ютерного практикуму, де для кожної із тем систематизовані теоретичні матеріали, приклади практичного засвоєння теоретичного матеріалу, завдання, та рекомендована література. В процесі вивчення даної навчальної дисципліни студенти засвоюють основні принципи розроблення клієнтської частини. В комп'ютерному практикумі основна увага приділяється створенню та наповненню вебсторінок, їх оформленню, та управлінню елементами на сторінці.

Для виконання комп'ютерних практикумів використовується будь-який текстовий редактор або редактор Microsoft Visual Studio Code, мова розмітки HTML, мова стилів CSS, мова програмування JavaScript та система контролю версій Git.

Студенти обирають тему із запропонованого переліку і всі практикуми виконуються в рамках обраної теми.

При вивченні конкретної теми практикуму слід уважно ознайомитися з поставленим завданням, пропрацювати матеріали комп'ютерного практикуму по цій темі. Лабораторний практикум відпрацьовується студентами групи індивідуально відповідно до обраної теми, протягом практикуму студенти виконують всі заплановані практикуми один за одним. Кожна практична робота комп'ютерного практикуму оформляється у вигляді звітного документу. Звіт повинен містити наступні розділи:

- стандартний титульний лист (вказати: ВНЗ, факультет, кафедру; тему та номеру поточної роботи; код групи та П.І.Б. виконавця; П.І.Б. викладача що перевірятиме роботу);
- назва практикуму та мета роботи;
- завдання до практикуму;

– тези теоретичних відомостей;

– результати виконання;

– висновки по роботі.

Visual Studio Code (VS Code) - це безкоштовний, відкритий і крос-платформний редактор коду, розроблений компанією Microsoft [1]. VS Code підтримує майже всі основні мови програмування; JavaScript, TypeScript, HTML, CSS (вбудовані), Python, C++, C#, Java, PHP та інші (через розширення) Встановити його можна, завантаживши інсталятор для вашої операційної системи з офіційного сайту https://code.visualstudio.com/download. VS Code використовується для веброзробки, розробки мобільних застосунків, роботи з базами даних. В ньому вбудована підтримка системи контролю версій Git для керування кодом, фіксації змін та злиття потоків роботи.

Головне вікно програми (рисунок 0.1):

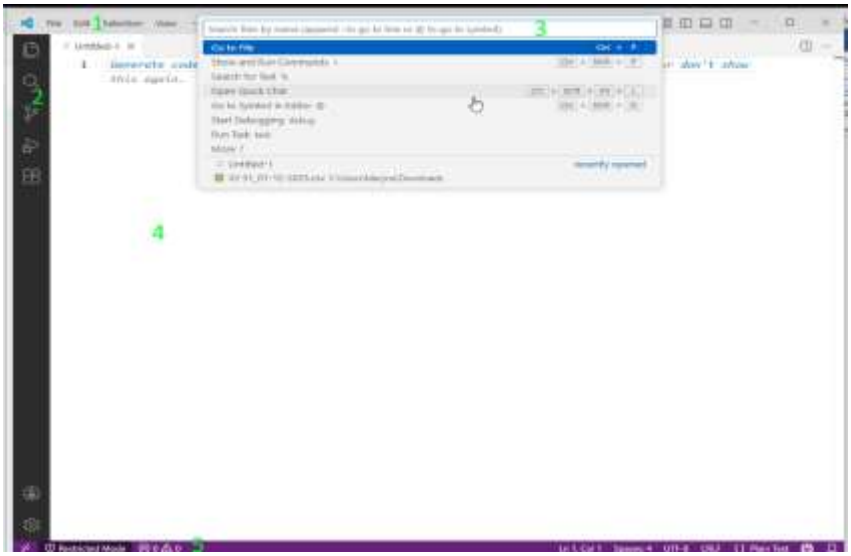


Рисунок 0.1 – Головне вікно VS Code

1 – меню програми;

2 – панель дій (менеджер файлів, пошук по файлам, управління вихідним кодом, запуск та налагодження, управління розширеннями);

3 – інтерфейс для виконання будь-якої команди, яку можна викликати;

4 – вікно редагування коду;

5 – рядок стану (попередження, перегляд помилок).

Комбінацією клавіш Ctrl+Shift+M відкривається панель, де будуть виводитись повідомлення про проблеми, консоль для налагодження, інтегрований термінал (рисунок 0.2):

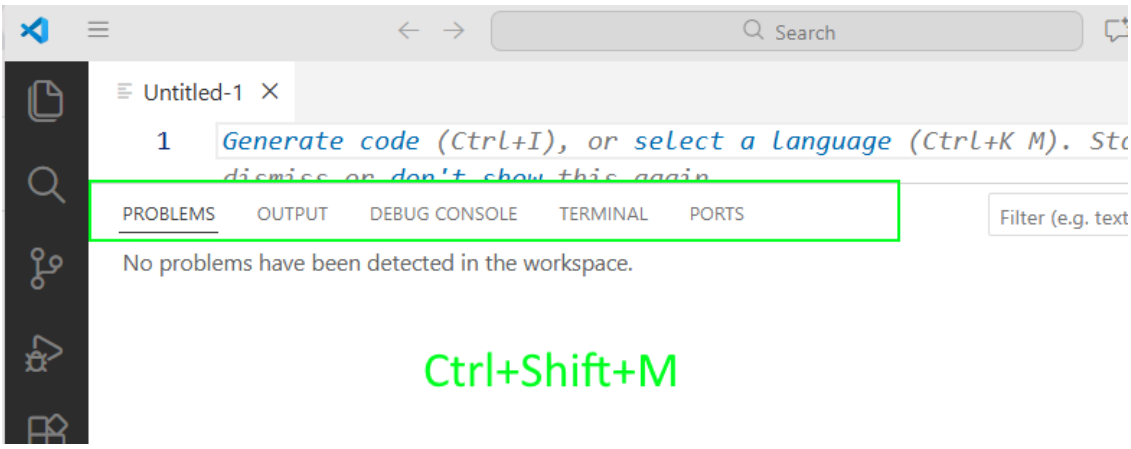


Рисунок 0.2 – Панель помилок, повідомлень

Менеджер файлів показує перелік відкритих файлів і каталогів, так можна відкрити каталог з файлами або клонувати репозиторій з GitHub - сервісу для зберігання коду, командної роботи та контролю версій (рисунок 0.3).

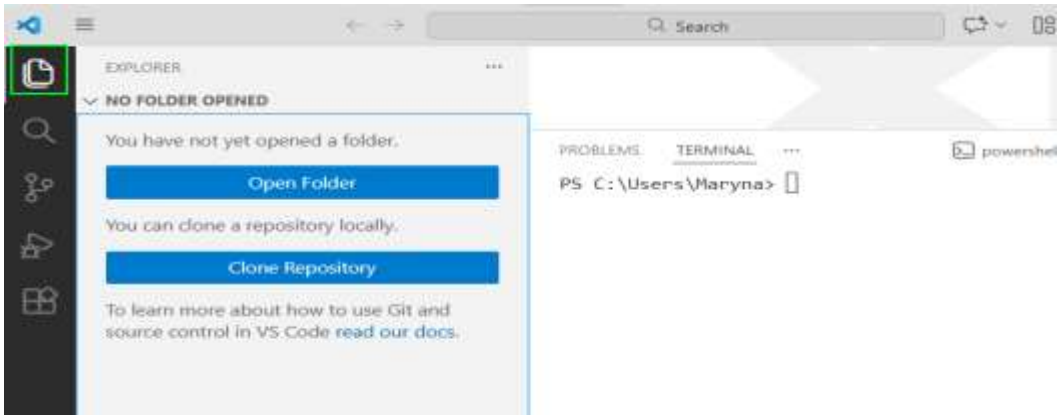


Рисунок 0.3 – Менеджер файлів

Можна відкрити каталог і в ньому вже створювати файли (рисунок 0.4):

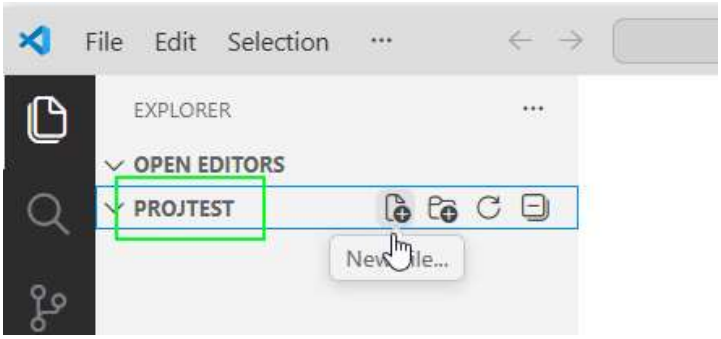


Рисунок 0.4 – Каталог для проєкту

При створенні файлу потрібно вказати його ім'я, розширення, наприклад, index.html, обрати мову (рисунок 0.5). Створиться файл ы ми можемо написати текст (рисунок 0.6).

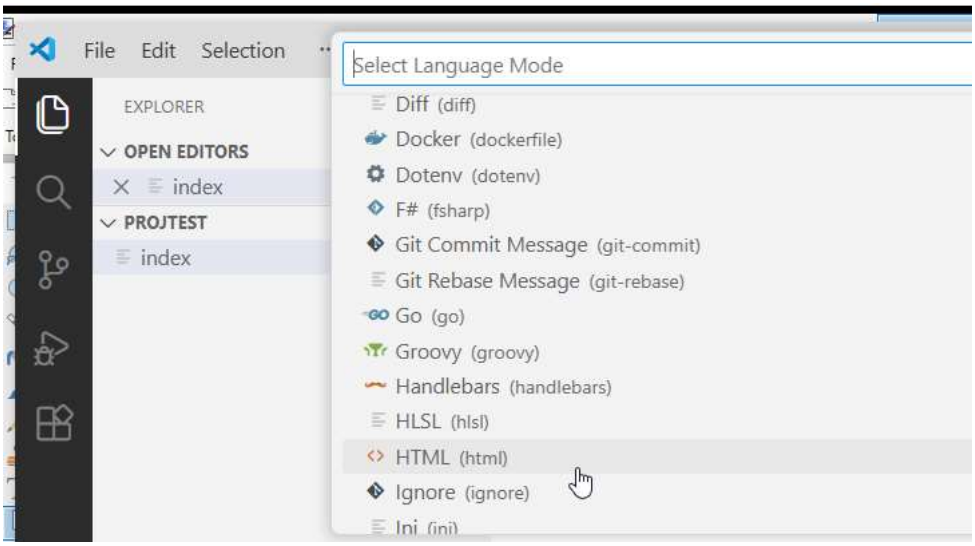


Рисунок 0.5 – Створення файлу

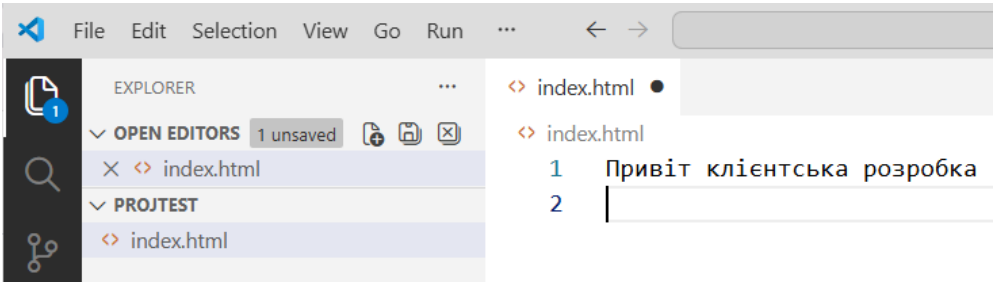


Рисунок 0.6 – Наповнення файлу

Для запуску і налагодження обираємо відповідний інструмент, вказуємо куди виводити результат (рисунок 0.7):

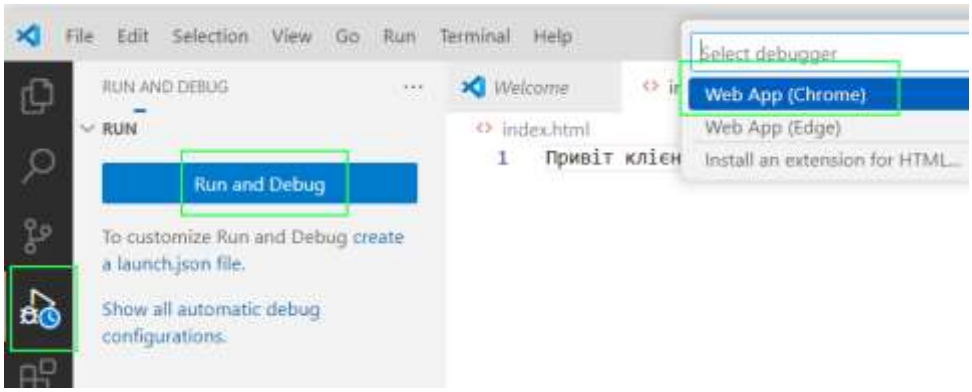


Рисунок 0.7 – Запуск і налагодження програми

Було обрано браузер Chrome, тому відкривається даний браузер і виводить результат (рисунок 0.8):

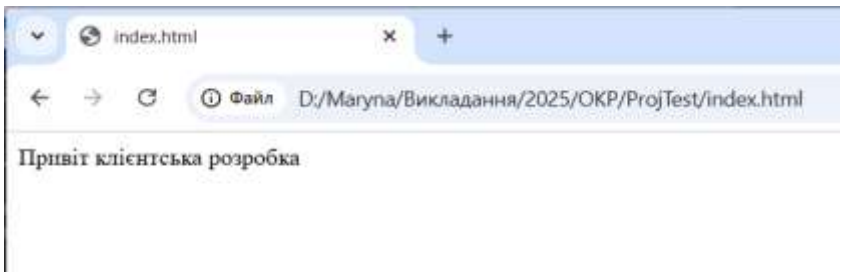


Рисунок 0.8 – Результат у браузері

Результат дебагінгу буде в нижній панелі. Також, можна ставити точки зупинки коду і переміщатись між ними (рисунок 0.9).

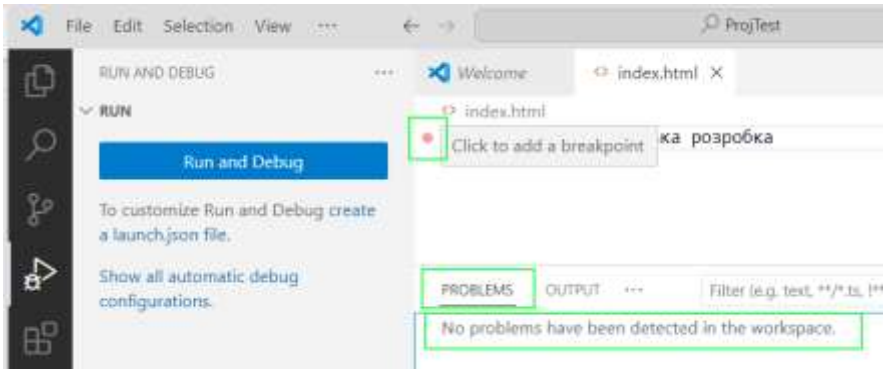


Рисунок 0.9 – Debbug

Теми:

1) Комп’ютерні ігри
2) Автомобілі
3) Музика
4) Відео
5) Картинна галерея
6) Продукти
7) Промислові товари
8) Комп’ютери
9) Подарунки
10) Оголошення
11) Програмне забезпечення
12) Мобільні телефони
13) Зірки кіно
14) Інтер’єр
15) Побутова техніка
16) Меблі
17) Готелі
18) Екскурсії
19) Книги
20) Бібліотека
21) Довідники
22) Фармацевтика
23) Будівництво
24) Туризм
25) Польоти
26) Музеї
27) Своя тема

ЛАБОРАТОРНИЙ ПРАКТИКУМ №1 Системи контролю версій. Git. Проєкт. Структура проєкту

Мета: навчитись працювати з системою контролю версій Git. Створювати, запускати html-сторінки.

Завдання

1) Встановити Git на комп'ютер. Створити каталог для проєкту. Ініціалізувати Git директорію у своєму проєкті (git init).

2) Створити 4 сторінки відповідно до обраної тематики з розширенням .html. Перша сторінка має назву - index.html. В вказати автора документа. Кожна сторінка повинна мати назву <title></title>. Кожна сторінка повинна мати заголовок <head></head>.

3) Додати створені сторінки під контроль Git на комп'ютері.

4) Зареєструватись на GitHub. Створити репозиторій. Завантажити проєкт на віддалений репозиторій (на GitHub).

Теоретичні відомості

Системи контролю версіями (Version Control Systems, VCS) важливий аспект розробки сучасного ПЗ [2].

VCS надає такі можливості:

– підтримку зберігання файлів у репозиторії;

– підтримку історії версій файлів у репозиторії;

– знаходження конфліктів при зміні вихідного коду та забезпечення синхронізації при роботі в багатокористувацькому середовищі розробки;

– відстеження змін авторів.

Звичайний цикл роботи розробника з СКВ виглядає так:

– оновлення робочої копії. Розробник виконує операцію оновлення робочої копії з сервера на скільки можливо;

– модифікація проєкту. Розробник локально модифікує проєкт, змінюючи файли, що входять до нього, в робочій копії;

– фіксація змін. Завершивши черговий етап роботи, розробник фіксує свої зміни, передаючи їх на сервер. Перед використанням VCS може вимагати від розробника оновлення.

Типи систем контролю версій:

– локальні системи контролю версій (rcs);

– централізовані системи контролю версій (CVS, Subversion);

– децентралізовані системи контролю версій (Git).

Локальні базуються на простій базі даних, в якій зберігаються зміни потрібних файлів (рисунок 1.1).

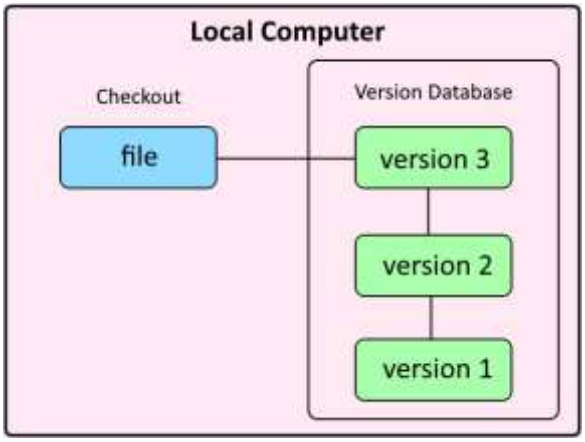


Рисунок 1.1 – Локальна система контролю версій

Централізовані - дозволяють співпрацювати розробникам. Є центральний сервер, на якому зберігаються усі файли під версійним контролем та клієнти, які взаємодіють з цим сервером (рисунок 1.2).

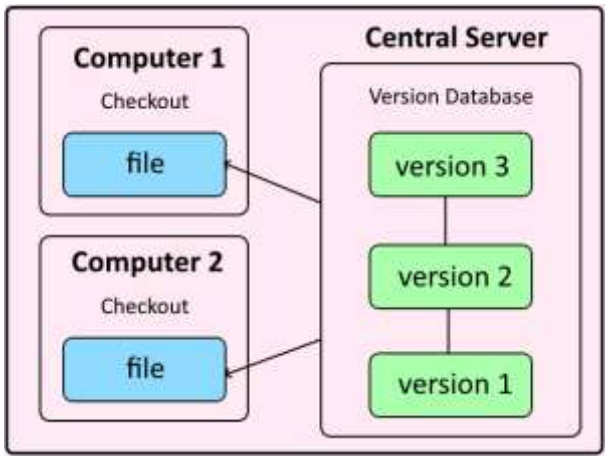


Рисунок 1.2 – Централізована система контролю версій

Переваги централізованих систем контролю версій:

– всі розуміють, хто і чим займається у проєкті;

– у адміністраторів є чіткий контроль над тим, хто і що може робити.

– Недоліки централізованих систем контролю версій:

– якщо сервер не працює, то розробники не можуть взаємодіяти і ніхто не може зберегти нову версію своєї роботи;

– якщо ж ушкоджується диск з центральною базою даних і немає резервної копії, ви втрачаєте абсолютно всю історію проєкту.

Розподілені системи контролю версій на відміну від централізованих систем, клієнти не просто вивантажують останні версії файлів, а повністю копіюють весь репозиторій (рисунок 1.3).

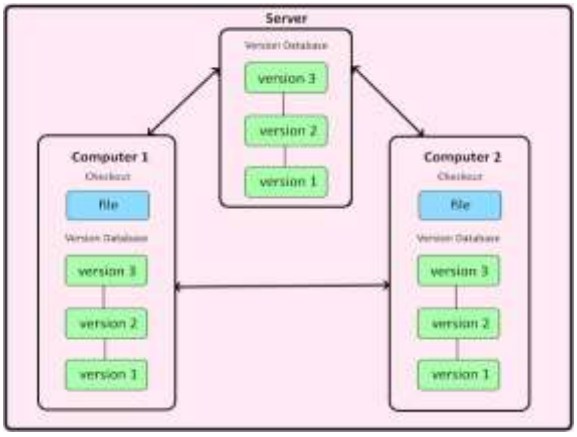


Рисунок 1.3 – Розподілена система контролю версій

Переваги розподілених систем контролю версій:

– так як кожного разу, коли клієнт забирає свіжу версію файлів, він створює собі повну копію всіх даних, то у разі збоїв на сервері, через який йшла робота, будь-який клієнтський репозиторій може бути скопійовано назад на сервер, щоб відновити базу даних;

– можливість працювати з кількома віддаленими репозиторіями. Таким чином, можна одночасно працювати по-різному з різними групами людей у рамках одного проєкту.

Git [3] - це розподілена система контролю версій, яка виникла внаслідок розробки ядра Linux. Git моделює свої дані як набір знімків, зберігаючи посилання на них. Git має цілісність, все перевіряється за контрольною сумою перед збереженням і з цього моменту ідентифікується за цією контрольною сумою. Git має три основні стани, в яких можуть знаходитися ваші файли:

– зафіксовано (підтверджено): дані безпечно зберігаються локально;

– змінено: ще не підтверджено;

– підготовлено: позначений для переходу до наступного коміту.

Ці стани приводять до трьох розділів:

– робочий каталог: це копія версії проєкту;

– Staging Area: простий файл, в якому зберігається інформація про те, що буде у наступному коміті;

– каталог Git: зберігає метадані та об'єкти бази даних для вашого проєкту.

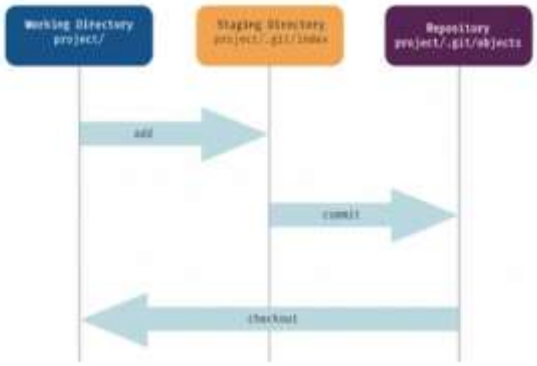


Рисунок 1.4 – Git

Установка Git

Необхідно встановити git. можна перейти на офіційний сайт за посиланням https://git-scm.com/downloads, вашу операційну систему, завантажити інсталер і встановити (рисунок 1.5).

Рисунок 1.5 – Завантаження, установка git

Робота з git буде починається з того, що потрібно ініціалізувати Git директорію у своєму проєкті. Це робиться за допомогою команди:

git init

Її необхідно ввести в корені вашого проєкту (рисунок 1.6). Це створить у поточному каталозі новий підкаталог .git (рисунок 1.7):

```
e:\Викладання\2025\ОКР_2025\TestGit>git init
Initialized empty Git repository in E:/Викладання/2025/ОКР_2025/TestGit/.git/

e:\Викладання\2025\ОКР_2025\TestGit>
```

Рисунок 1.6 – git init

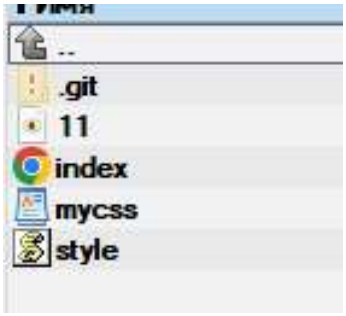


Рисунок 1.7 – Директорія git

У цій директорії буде вся конфігурація git та історія проєкту (рисунок 1.8). За бажанням можна редагувати ці файли вручну, вносячи необхідні зміни в історію проєкту.

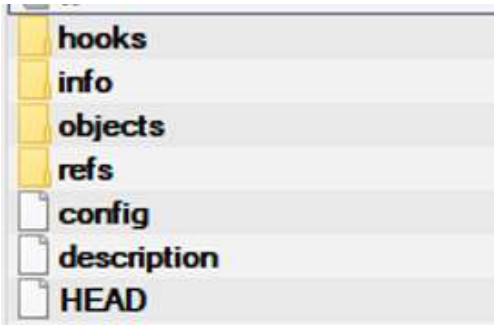


Рисунок 1.8 – Вміст директорії git

Опис вмісту каталогу:

config - файл містить налаштування git репозиторію. тут можна зберігати email та ім'я користувача;

description - файл призначений для gitweb - це вебінтерфейс, написаний для перегляду Git-репозиторія використовуючи веббраузер і містить інформацію про проєкт (назва проєкту та його опис);

hooks - у цьому каталозі git надає набір скриптів, які можуть автоматично запускатись під час виконання git команд. Наприклад, можна написати скрипт, який редагуватиме повідомлення коміту відповідно до ваших вимог;

info - каталог містить файл exclude, в якому можна вказувати будь-які файли, і піt не додаватиме їх у свою історію;

refs - каталог зберігає копію посилань на об'єкти комітів (фіксацій) у локальних і віддалених гілках;

logs - каталог зберігає історію проєкту для всіх гілок у вашому проєкті.

objects - каталог objects зберігає у собі BLOB об'єкти, кожен із яких проіндексований унікальним SHA (Secure Hash Algorithm);

index - проміжна область з метаданими, такими як тимчасові мітки, імена файлів, а також файли SHA, які вже упаковані git. У цю область потрапляють файли, над якими ви працювали, під час виконання команди git add (додавання файлів);

head -файл містить посилання на поточну гілку, в якій ви працюєте;

orig_head - щоразу під час злиття до цього файлу потрапляє SHA гілка, з якою проводилося злиття.

fetch_head - файл зберігає посилання у вигляді SHA на гілки, які брали участь у git fetch (завантаження комітів, файлів, посилань з віддаленого репозиторія в локальний);

merge_head - файл зберігає посилання у вигляді SHA на гілки, які брали участь у git merge (злитті);

commit_editmsg - файл містить останнє введене вами повідомлення коміту.

Робота над проєктом з використанням СКВ обмежена певними діями:

1) внести зміни до проєкту;

2) додати зміни до індексу (staging area) - git add (ви повідомляєте git, які саме зміни повинні бути занесені в історію);

3) закомітити зміни - git commit (зберегти зміни до історії проєкту);

4) запушити - git push (надіслати результати роботи на віддалений сервер, щоб інші розробники теж мали до них доступ).

git help - найголовніша команда, яка виводить перелік усіх команд із коротким описом (рисунок 1.9).

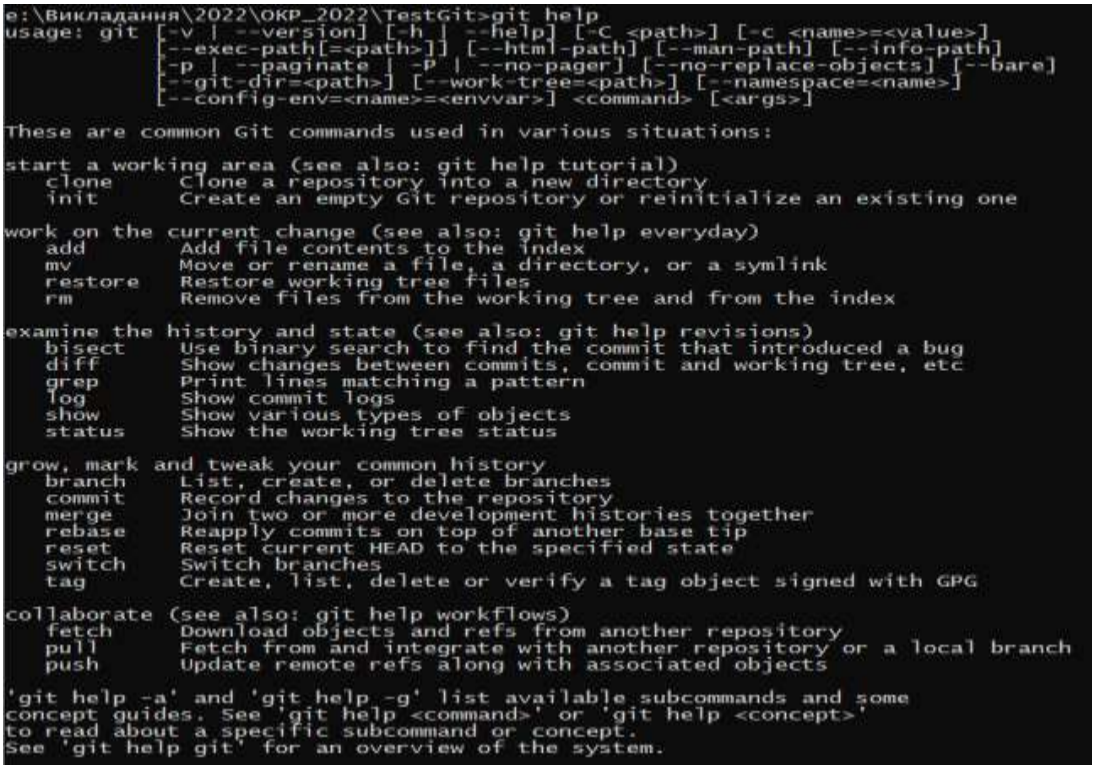


Рисунок 1.9 – Команда git help

Створимо новий файл 111.txt і перевіримо стан директорії.

Команда **git status** відображає стан директорії та індексу (staging area) (). Це дозволяє визначити, які файли в про’кті відстежуються git, а також які зміни будуть включені до наступного коміту (рисунок 1.10):

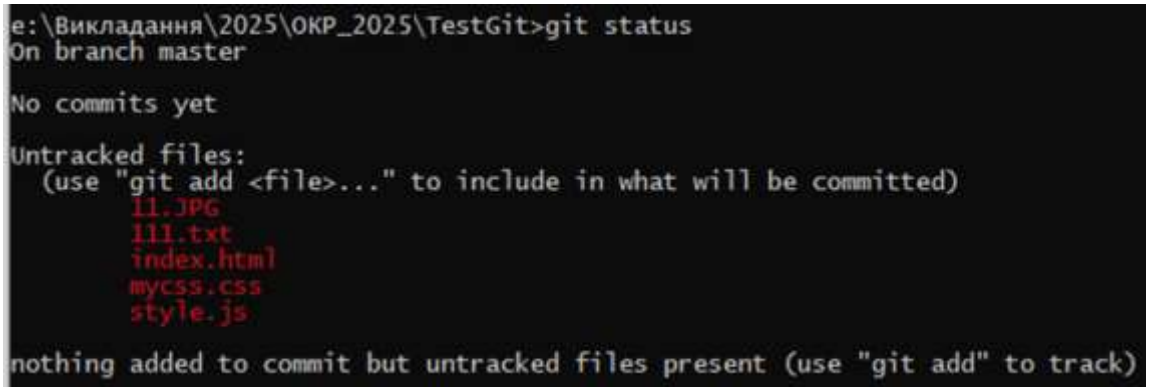


Рисунок 1.10 – Команда git status

Команда **git add** - додає файл під контроль. Додамо створений файл під контроль і перевіримо статус git (рисунок 1.11).

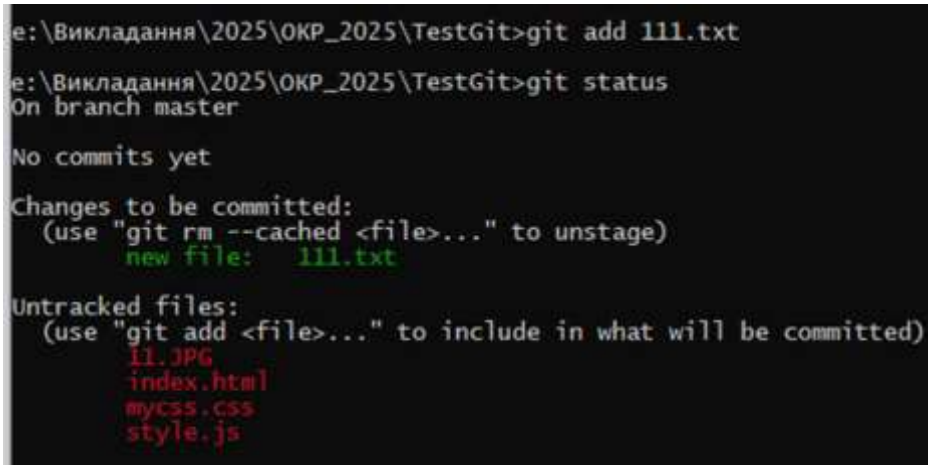


Рисунок 1.11 – Команда git add

Файл додано до індексу. Тепер можна закомітити внесені зміни та додати коментар. Для цього є команда git commit -m "first commit"

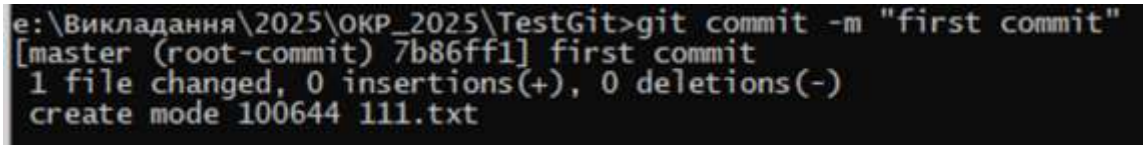


Рисунок 1.12 – Команда git commit

Відредагуємо файл і перевіримо статус (рисунок 1.13):

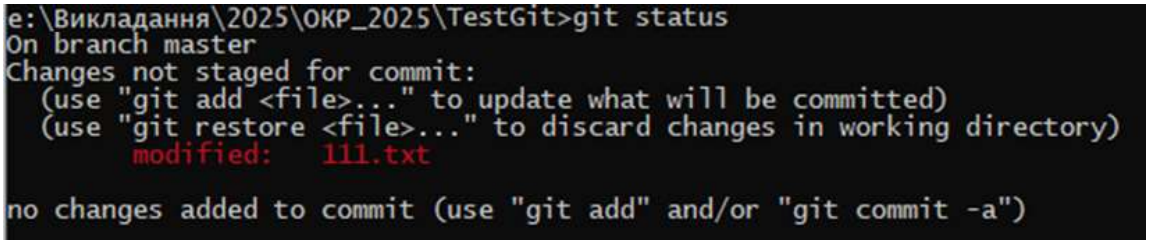


Рисунок 1.13 – Зміни після редагування

Git повідомляє, що ми маємо змінений файл 111.txt. І тепер нам потрібно додати його в індекс і потім закомітити.

Ми все зберігаємо на локальній машині, тепер нам потрібно відправити версію нашої історії на віддалений сервер.

Можна скористатись репозиторієм на GitHub.

Реєстрація на https://github.com/ (рисунок 1.14):

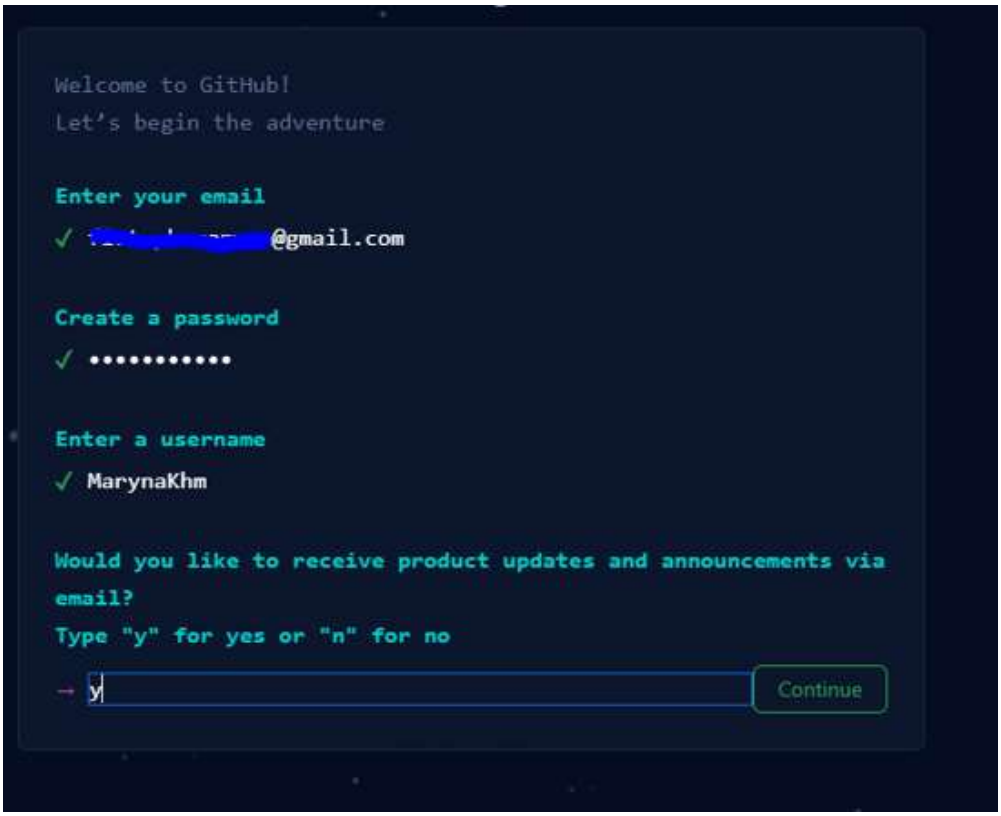


Рисунок 1.14 – Реєстрація на GitHub

Створення репозиторія (рисунок 1.15):

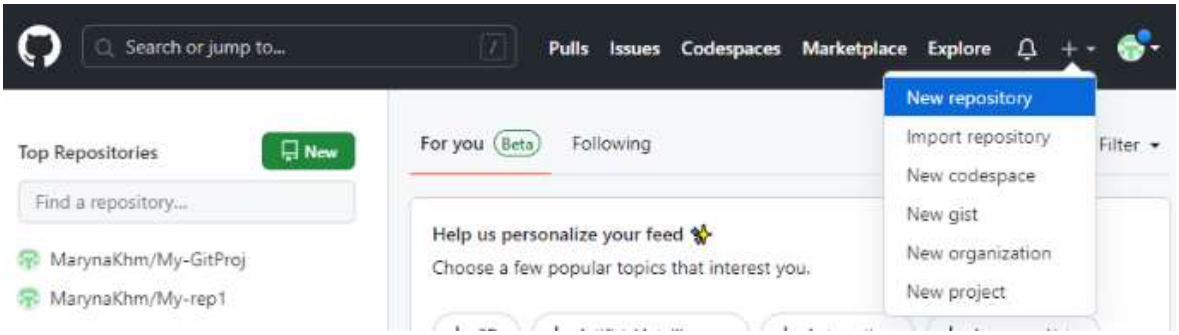


Рисунок 1.15 – Створення репозиторія

Потрібно ввести коротку назву, видимість сховища, опис (опціонально), натиснути «Create repository» (рисунок 1.16, 1.17):

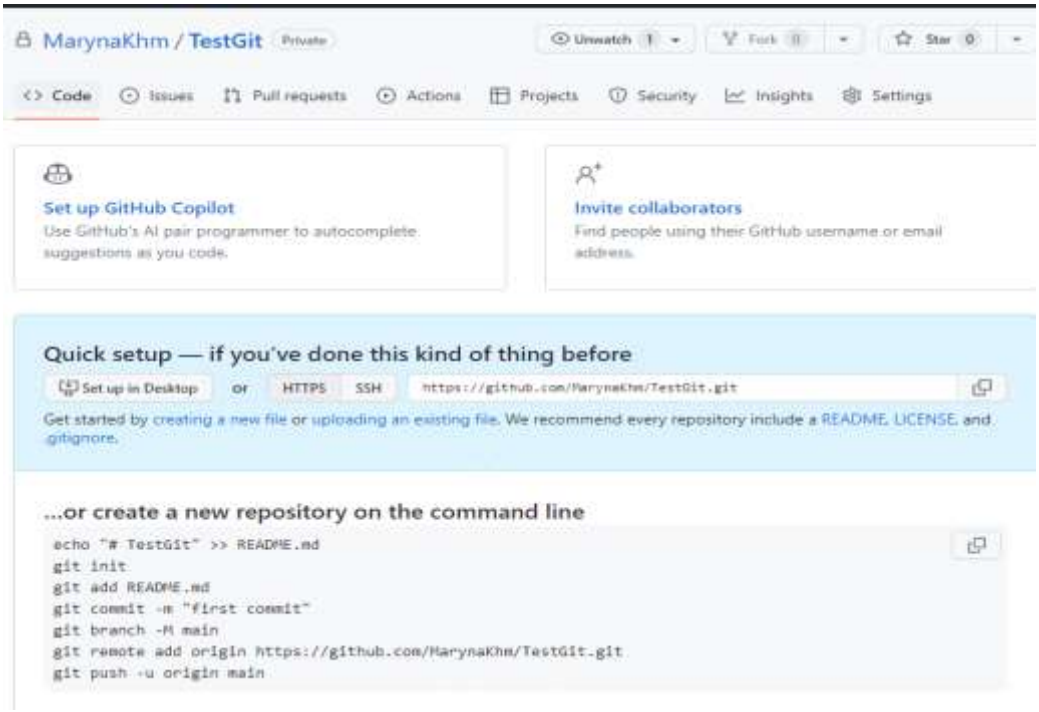


Рисунок 1.16 – Створення репозиторія

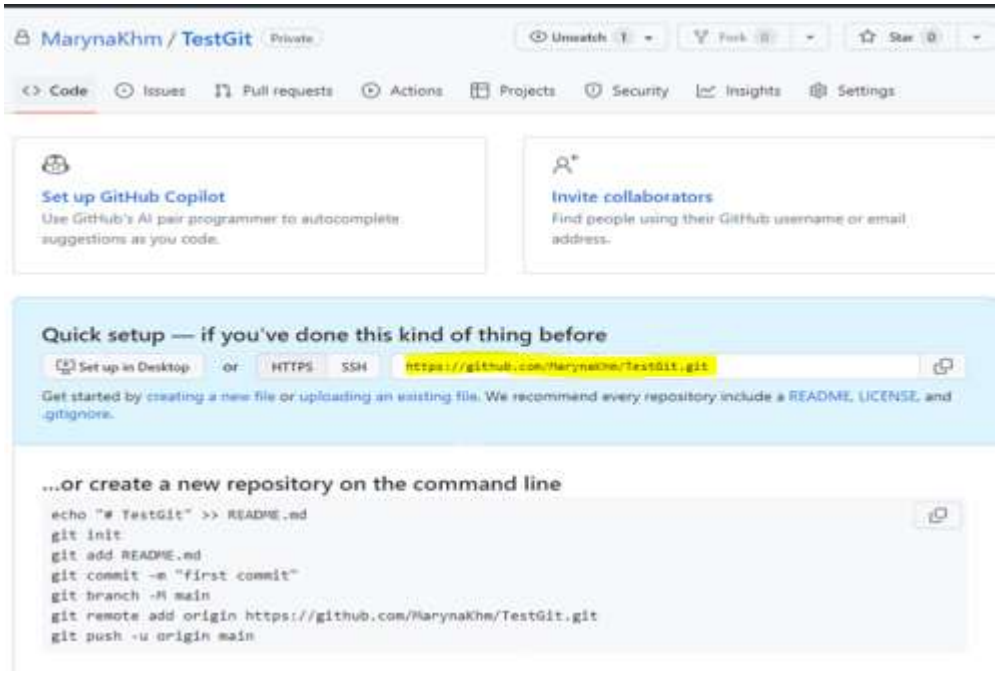


Рисунок 1.17 – Посилання на репозиторій

Далі необхідно додати віддалений репозиторій у ваш Git:

git remote add <remote_name> <remote_repo_url> - команда прив’язує віддалене сховище з локальним репозиторієм. З цього моменту можна звернутися до віддаленого репозиторію через посилання (рисунок 1.18):

Рисунок 1.18 – git remote add

«origin» є коротким іменем для віддаленого репозиторію, на який він буде посилатись (може бути будь-яке ім’я).

Відправимо результат нашої роботи в репозиторій:

git push origin master

Тепер історія змін вашого проєкту буде зберігатися у віддаленому репозиторії (рисунок 1.19, 1.20).

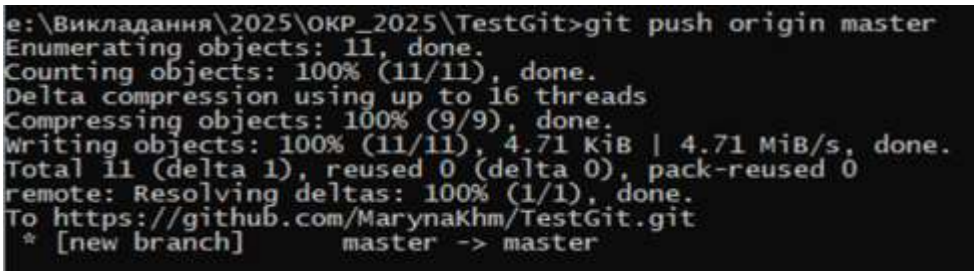


Рисунок 1.19 – git push origin master

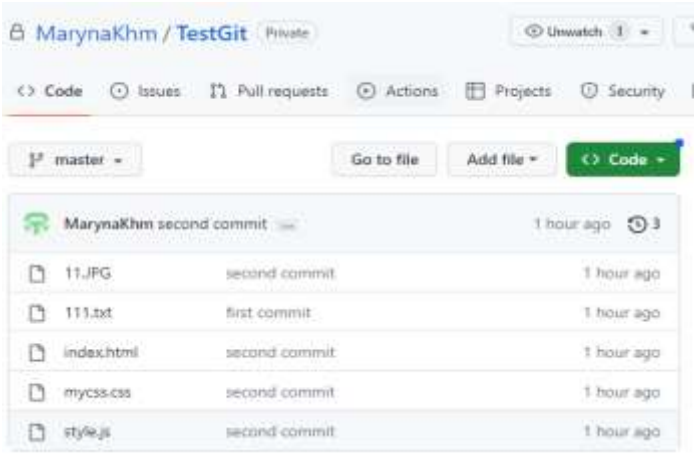


Рисунок 1.20 – git push origin master

Гілка в Git - це простий переміщуваний покажчик на один з комітів. За замовчуванням, ім'я основної гілки в Git — master. Як тільки ви почнете створювати коміти, гілка master завжди вказуватиме на останній коміт.

Для перегляду історії в Git є ряд команд:

git log - призначена для відображення всієї історії вашого проєкту, Дозволяє дізнатися, які зміни ви внесли раніше (рисунок 1.21):

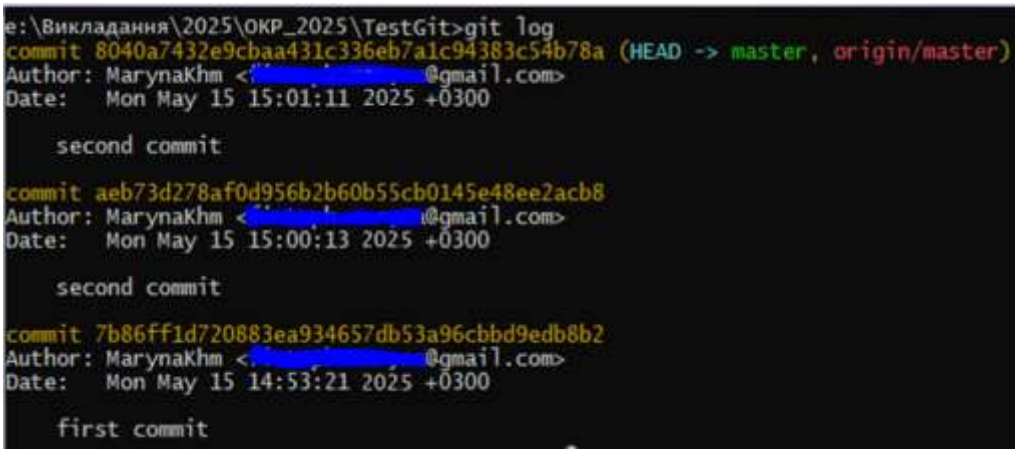


Рисунок 1.21 – git log

git show - використовується для відображення повної інформації про будь-який об’єкт у Git, коміт або гілку (рисунок 1.22). За замовчуванням git show відображає інформацію коміту, на який у даний момент часу вказує HEAD.

Рисунок 1.22 – git show

git reflog - виводить упорядкований список комітів, на який вказує HEAD (відображає історію всіх ваших переміщень по проєкту) (рисунок 1.23). Основна перевага цієї команди полягає в тому, що якщо ви випадково видалили частину

історії або відкотилися назад, ви зможете переглянути момент втрати потрібної вам інформації та відкотитись назад. git reflog зберігає свою інформацію на вашій машині окремо від комітів, тому при видаленні будь-чого в історії можна знайти її в git reflog.

```
e:\Викладання\2025\ОКР_2025\TestGit>git reflog
8040a74 (HEAD -> master, origin/master) HEAD@{0}: commit: second commit
aeb73d2 HEAD@{1}: commit: second commit
7b86ff1 HEAD@{2}: commit (initial): first commit
```

Рисунок 1.23 – git reflog

git reset - дозволяє відкотити проєкт до визначеної точки. Цю команду можна використовувати з трьома параметрами:

git reset --soft <commit> - вміст вашого індексу, а також робочої директорії, залишаються незмінними. якщо ми відкотимося назад на пару комітів, ми змінимо посилання вказівника HEAD на вказаний коміт і всі зміни, які були до цього внесені, покажуться в індексі.

git reset --mixed <commit> - змінимо посилання вказівника HEAD, але всі попередні зміни в індекс не попадуть, а будуть відображатися як не занесені в індекс. Це дає можливість внести в індекс тільки ті зміни, які нам необхідні.

git reset --hard <commit> - знову змінимо посилання вказівника HEAD, але всі попередні зміни не попадуть ні в індекс, ні в зону шуканих файлів, ми повністю зітремо всі зміни, які внесли раніше.

Переглянемо історію (рисунок 1.24). Зроблено 4 коміти.

```
e:\Викладання\2025\ОКР_20252\TestGit>git log --oneline
434c04f (HEAD -> master, origin/master) 3 commit
8040a74 second commit
aeb73d2 second commit
7b86ff1 first commit
```

Рисунок 1.24 – Історія проєкту

Відкотимось до 2 коміта з git reset –soft (рисунок 1.25):

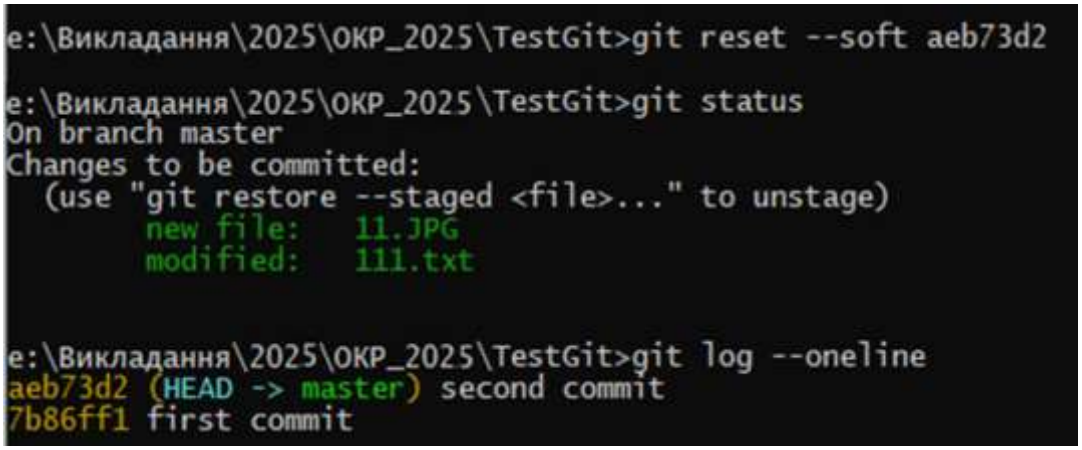


Рисунок 1.25 – git reset –soft

Вказівник HEAD перемістився на другий коміт, а стан індексу залишився незмінним.

Відкотимось до 2 коміта з git reset --mixed (рисунок 1.26):

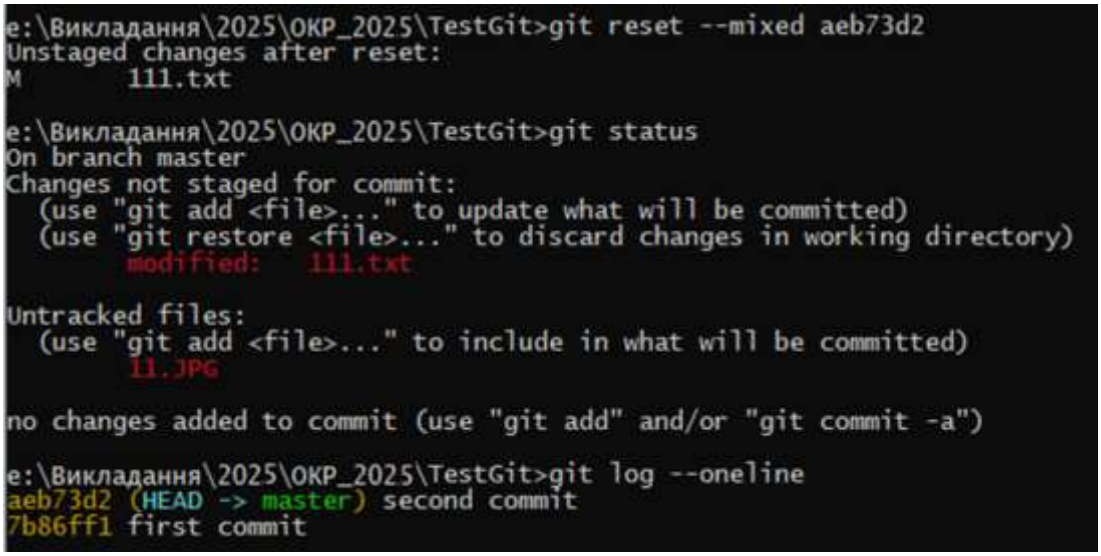


Рисунок 1.26 – git reset – mixed

У цьому випадку вказівник знову перемістився на другий коміт, але попередні зміни потрапили в зону відстежуваних файлів. Це означає, що тепер ми можемо вирішити - залишити ці зміни, додавши їх в індекс, або позбутися них.

Відкотимось до 2 коміта з git reset --hard (рисунок 1.27):

```
e:\Викладання\2025\ОКР_2025\TestGit>git reset --hard aeb73d2
HEAD is now at aeb73d2 second commit

e:\Викладання\2025\ОКР_2025\TestGit>git status
On branch master
nothing to commit, working tree clean

e:\Викладання\2025\ОКР_2025\TestGit>git log --oneline
aeb73d2 (HEAD -> master) second commit
7b86ff1 first commit
```

Рисунок 1.27 – git reset – hard

Вказівник HEAD перемістився на другий коміт, а всі попередні зміни були стерті, що видно по порожньому індексу та зоні відстежуваних файлів.

Гілкування в Git

Гілкування означає, що у вас є можливість працювати над різними версіями проєкту. Тобто, якщо раніше історія вашої розробки була прямою послідовністю комітів, то тепер вона може розійтися в певних точках. Це дуже корисна функція з багатьох причин, наприклад для взаємодії кількох розробників.

Стан робочої директорії (локальний репозиторій) (рисунок 1.28):

```
e:\Викладання\2025\ОКР_2025\TestGit>git log --oneline
aeb73d2 (HEAD -> master) second commit
7b86ff1 first commit
```

Рисунок 1.28 – Стан робочої директорії локальному репозиторії

У віддаленому репозиторії (рисунок 1.29):

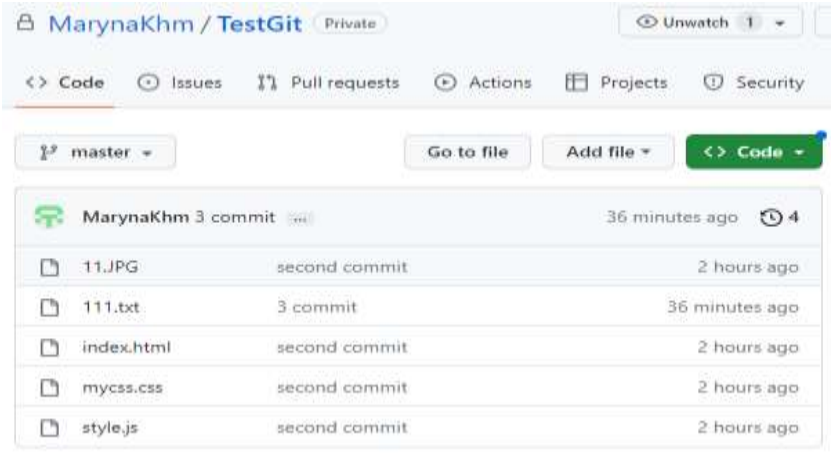


Рисунок 1.29 – Стан робочої у віддаленому репозиторії

Команда **git pull** використовується для вилучення та завантаження вмісту з віддаленого репозиторію та негайного оновлення локального репозиторію цим вмістом (рисунок 1.30).

```
e:\Викладання\2025\ОКР_2025\TestGit>git pull origin master
From https://github.com/MarynaKhm/TestGit
 * branch            master     -> FETCH_HEAD
Updating aeb73d2..434c04f
Fast-forward
 11.JPG  | Bin 0 -> 10643 bytes
 111.txt |   1 +
 2 files changed, 1 insertion(+)
 create mode 100644 11.JPG

e:\Викладання\2025\ОКР_2025\TestGit>git log --onelin
fatal: unrecognized argument: --onelin

e:\Викладання\2025\ОКР_2025\TestGit>git log --oneline
434c04f (HEAD -> master, origin/master) 3 commit
8040a74 second commit
aeb73d2 second commit
7b86ff1 first commit
```

Рисунок 1.30 – git pull

Git за замовчуванням під час ініціалізації створює гілку master і вже веде свою роботу в ній. Перевірити можна **git branch** (рисунок 1.31):

```
e:\Викладання\2025\ОКР_2025\TestGit>git branch
* master
```

Рисунок 1.31 – git branch

Для створення нової гілки, її потрібно створити **git branch <branch_name>** (рисунок 1.32):

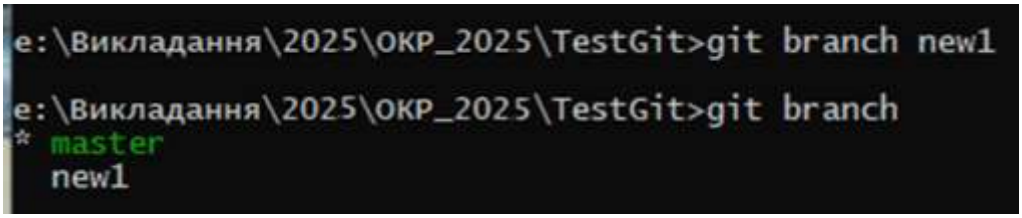


Рисунок 1.32 – git branch <branch_name>

* вказує на поточну гілку, в якій ми працюємо.

Для того, щоб перейти на іншу гілку, є команда **git checkout <branch_name>** (рисунок 1.33)

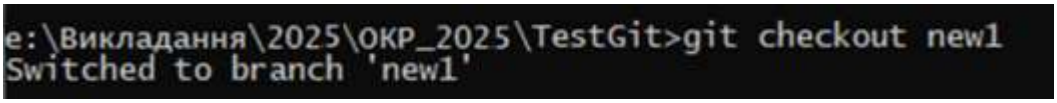


Рисунок 1.33 – git checkout <branch_name>

Зробимо зміни в файлі 111.txt, зробимо коміт (рисунок 1.34)

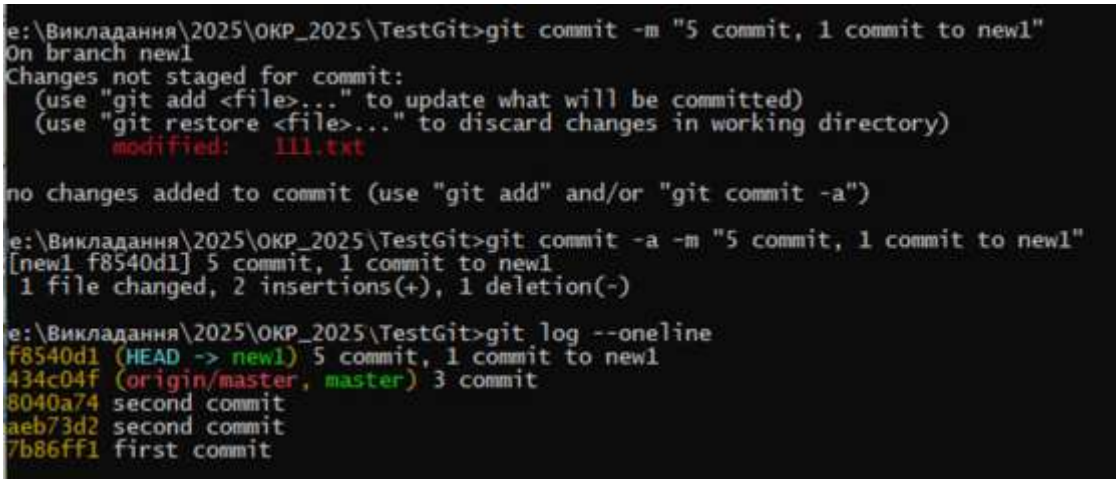


Рисунок 1.34 – Коміт зміненого файлу

Перейдемо в гілку **master - git checkout master** і переглянемо історію git log --oneline, і переконаємось, що все залишилося без змін (рисунок 1.35):

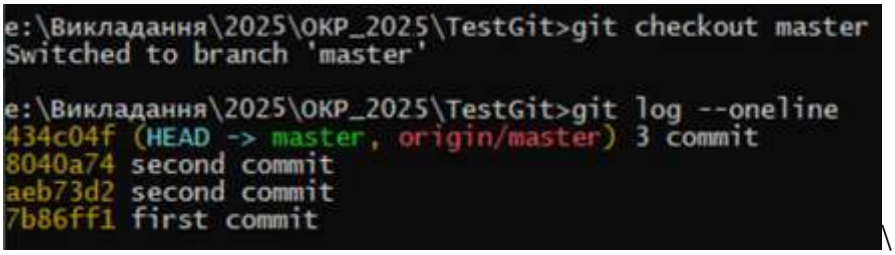


Рисунок 1.35 – master - git checkout master

Окрім розділеної історії в Git (гілкування), ми також можемо об’єднати одночасно два потоки розробки. Це означає, що нашу роботу в новій гілці ми можемо злити в master. Такий процес злиття можна виконати за допомогою команди git merge <branch_name>.

Якщо ми хочемо злити зміни з гілки «new1» у гілку «master», нам необхідно перейти на гілку «master», і в цій гілці виконати **git merge new1** (рисунок 1.36):

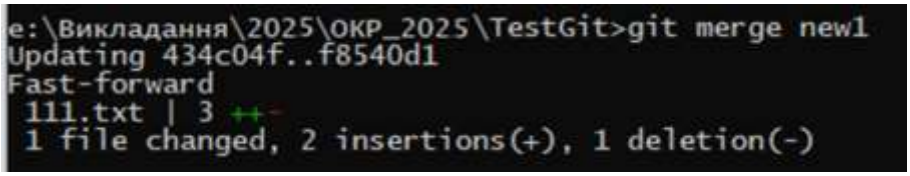


Рисунок 1.36 – git merge new1

Тепер непотрібної гілки можна позбутися і видалити її за допомогою команди **git branch -d <branch_name>** (рисунок 1.37):

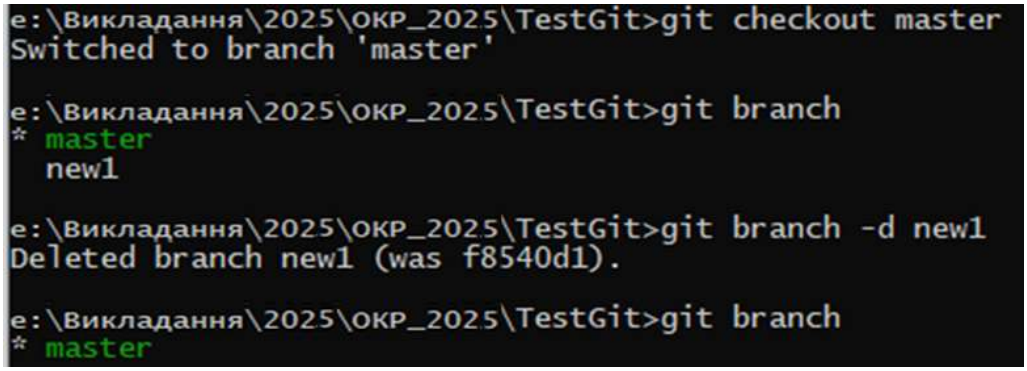


Рисунок 1.37 – git branch -d <branch_name>

Завантажимо все на віддалений репозиторій (рисунок 1.38):

```
e:\Викладання\2025\ОКР_2025\TestGit>git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 279 bytes | 279.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/MarynaKhm/TestGit.git
   434c04f..f8540d1  master -> master
```

Рисунок 1.38 – Завантаження змін на віддалений репозиторій

Переглянемо локальний репозиторій (рисунок 1.39):

```
e:\Викладання\2025\ОКР_2025\TestGit>git log --oneline
f8540d1 (HEAD -> master, new1) 5 commit, 1 commit to new1
434c04f (origin/master) 3 commit
8040a74 second commit
aeb73d2 second commit
7b86ff1 first commit
```

Рисунок 1.39 – Стан локального репозиторію

Переглянемо віддалений репозиторій (рисунок 1.40). Вміст локального та віддаленого репозиторіїв співпадає.

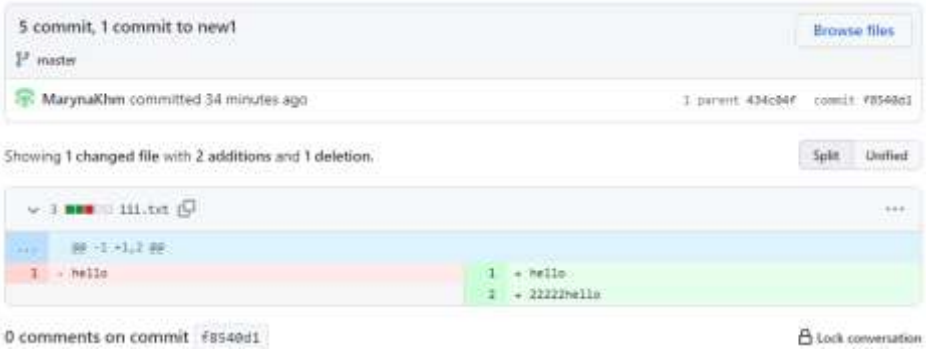


Рисунок 1.40 – Стан віддаленого репозиторію

Що потрібно зробити, щоб інша людина отримала всі ваші зміни?

Для цього знадобиться GitHub або будь-який інший сервіс для зберігання коду.

Якщо у людини раніше не було проєкту, то їй доведеться його «клонувати» **git clone <URL repo>** собі (рисунок 1.42, 1.43).

Адресу репозиторію на GitHub можна отримати, натиснувши на зелену кнопку Code (рисунок 1.41).

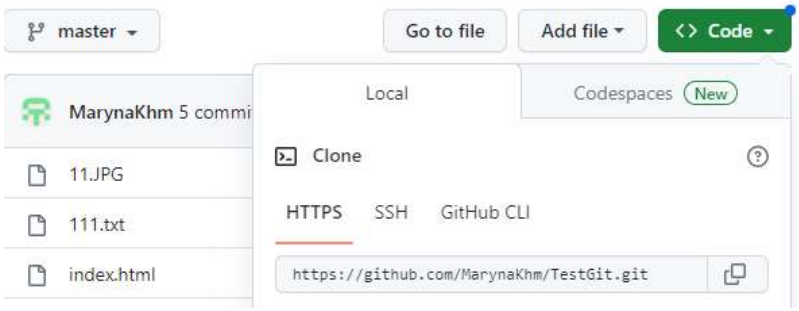


Рисунок 1.41 – Посилання на віддалений репозиторій

```
e:\Викладання\2025\ОКР_2025\TestGit>cd..

e:\Викладання\2025\ОКР_2025>git clone https://github.com/MarynaKhm/TestGit.git Second_test_GIT
Cloning into 'Second_test_GIT'...
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 17 (delta 3), reused 16 (delta 2), pack-reused 0
Receiving objects: 100% (17/17), 5.14 KiB | 5.14 MiB/s, done.
Resolving deltas: 100% (3/3), done.
```

Рисунок 1.42 – git clone <URL repo>

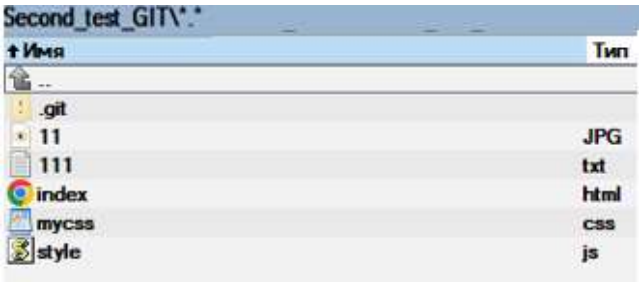


Рисунок 1.43 – Вміст локального репозиторія-клона

Перед тим, як створити новий функціонал і нову гілку, варто оновити master. Для цього потрібно знаходитися в цій гілці і виконати наступну команду **git checkout master** (рисунок 1.44):

```
e:\Викладання\2025\ОКР_2025\Second_test_GIT>git checkout master
Already on 'master'
Your branch is up to date with 'origin/master'.
```

Рисунок 1.41 – git checkout master

git pull origin master (рисунок 1.45):

```
e:\Викладання\2025\ОКР_2025\Second_test_GIT>git pull origin master
From https://github.com/MarynaKhm/TestGit
 * branch            master     -> FETCH_HEAD
Already up to date.
```

Рисунок 1.45 – git pull origin master

Які проблеми можуть виникнути при злитті?

Git старається автоматично зливати зміни, однак це не завжди можливо. Іноді виникають конфлікти.

Наприклад, коли в двох гілках були зміни в одному і тому ж рядку коду. Якщо таке сталося, то необхідно вирішити конфлікт вручну.

git checkout my1

git merge master

Якщо конфлікт, то буде відповідне повідомлення «Auto-merging 111.txt CONFLICT (content): Merge conflict in 111.txt. Automatic merge failed; fix conflicts and then commit the result»

Для перегляду вмісту файлу можна скористатись командою cat.

У файлі можна побачити наступні рядки (рисунок 1.46):

```
hello
22222hello
<<<<<<< HEAD
333333_my1
=======
333333_master
>>>>>>> master
```

Рисунок 1.46 – Конфлікт при злитті

Рядок ======= є «центром» конфлікту.

Весь вміст між цим центром і рядком <<<<<<< HEAD знаходиться в поточній гілці my1, на яку посилається покажчик HEAD.

А весь вміст між центром і рядком >>>>>>> master є вмістом гілки, звідки потрібно зливати.

Після внесення потрібних змін потрібно додати файл через git add <file name> як змінений і створити новий commit:

git add 111.txt

git commit -m "fixed conflict“

Переглянути зміни відносно двох гілок можна командою:

git diff <source branch> <target branch>

HTML [4] - це мова розмітки, яка представляє прості правила оформлення і компактний набір структурних і семантичних елементів розмітки (тегів).

HTML дозволяє описувати спосіб представлення логічних частин документа (заголовки, абзаци, списки і т.д.) і створювати вебсторінки різної складності.

Спочатку мова HTML (HyperText Markup Language) була задумана і створена як засіб структурування та форматування документів без прив'язки до засобів відображення. В ідеалі, гіпертекстовий документ повинен однаково виглядати на різних пристроях (монітор, мобільний телефон, принтер, медіа-проєктор і т.п.).

Гіпертекст - структура, що дозволяє встановлювати смислові зв'язки між елементами тексту та іншими документами.

Гіперпосилання - фрагмент тексту, який є вказівником на інший файл чи об'єкт.

Гіпертекстові документи обробляються «браузерами», які читають код розмітки та виводять документ у відформатованому вигляді [5].

Найбільш популярними сьогодні браузерами є Internet Explorer, Mozilla Firefox, Apple Safari, Google Chrome і Opera. Поряд із цими існує безліч інших браузерів, які використовують їх системні бібліотеки (т.зв. «рушій») або працюють на власному коді. Різноманітність браузерів та відмінності в їхній функціональності, а також початкова орієнтація HTML на підтримку різних пристроїв виведення, призводить розробників вебсайтів до необхідності вирішення питання крос-браузерності.

Крос-браузерність - властивість сайту відображатися і працювати у всіх популярних браузерах ідентично. Під ідентичністю розуміється відсутність розвалів верстання та здатність відображати матеріал з однаковим ступенем читабельності.

Вевузол або вебсайт - група вебсторінок, взаємопов'язаних загальними гіперпосиланнями.

Вебсторінки можуть містити; тексти, таблиці, логотипи, емблеми, графіку, банери, мультимедіа-файли, флеш-анімації, гіперпосилання, таблиці стилів - це файли з розширенням .css, у яких прописаний зовнішній вигляд елементів (таке оформлення надає сторінкам сучасного, естетичного стилю), скрипти - програми, які розширюють можливості сторінок, роблять її активним із зворотнім зв'язком (форми, реєстраційні листи, пишуться мовою Jawascript тощо), аплети – це програми, які завантажуються з сайту на комп'ютер клієнта при відкритті сторінки, створюють різні відеоефекти (перегортання сторінок, вихровий рух, ефект полум'я, деформації зображення, пишуться мовою Java з розширенням .class).

HTML 5 - це, платформа для створення вебзастоуноків, ніж стандарт, що продовжує традиції попередників. HTML 5 регламентує взаємодію з JavaScript за допомогою об'єктної моделі документа. HTTP - HyperText Transfer Protocol -

«протокол передачі гіпертексту») - протокол прикладного рівня передачі даних (спочатку - у вигляді гіпертекстових документів).

HTML -документ - текстовий файл з розширенням .html. Не використовуйте кириличні назви файлів і папок - давайте їм англійські назви!

Зазвичай, перегляд сайту починається з головної сторінки. зазвичай в корені сайту велика кількість різних файлів, як вебсервер дізнається, що йому завантажувати автоматично?

Звичайно, якщо шлях до файлу вказаний безпосередньо, жодних питань не виникає. Але в більшості випадків адреса сайту вказується коротко, без зайвих файлів на кінці. Тоді читаються налаштування сервера і визначається файл, який слід показати, а також, чи вказаний файл є в наявності.

Як правило, такий файл має імена index.html, index.htm, default.htm, це можна встановити самому.

Apache HTTP-сервер - вільний вебсервер. Apache є кросплатформним програмним забезпеченням, підтримує операційні системи Linux, BSD, Mac OS, Microsoft Windows. Перевагами Apache є надійність і гнучкість конфігурації. Для вебсервера Apache є конфігураційний файл .htaccess - це звичайний текстовий документ, який потрібно розмістити в корені сайту. У ньому слід прописати такий рядок: **DirectoryIndex index.html index.htm**, де через пробіли вказуються імена файлів, які слід переглядати на предмет наявності та запускати автоматично. Файл .htaccess пишеться без будь-якого розширення з обов'язковою крапкою на початку імені. htaccess може працювати на деяких серверах, неповні шляхи, працюють лише під управлінням вебсервера, на локальному комп'ютері це не працює. Якщо файл index.html відсутній у вказаній теці, браузер покаже список файлів, які містяться в ній.

HTML-документ складається з тексту, який є інформаційним вмістом і спеціальними засобами мови HTML - тегами розмітки, які визначають структуру та зовнішній вигляд документа при його відображенні браузером (рисунок 1.47).

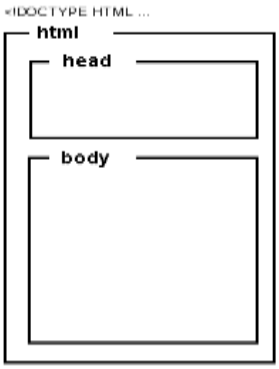


Рисунок 1.47 – Структура HTML-документа

<html> початок розмітки документа

<head> … **</head>**

<body>

… вміст сторінки

</body>

</html> кінець розмітки документа

Кожен тег призначений для вирішення певної задачі: роботи з текстом, посиланнями, графікою, таблицями і т.д.

Опис документа починається з вказівки його типу. Для документів HTML5, секція <! DOCTYPE> буде <! DOCTYPE HTML>.

У заголовку <head> вказують назву HTML-документа і інші параметри, які браузер буде використовувати при відображенні документа.

Тіло документа <body> - це та частина, в яку поміщається власне вміст HTML-документа. Тіло включає призначений для відображення текст і управляючу розмітку документа (теги), які використовуються браузером.

Стандарт вимагає, щоб секція DOCTYPE була присутня в документі, тому що це дозволяє прискорити і поліпшити обробку гіпертексту.

Заголовок <title> </title> призначений для розміщення метаінформації, яка описує вебдокумент як такий.

Мета-тег HTML - це елемент розмітки html, що описує властивості документа (метадані). Призначення мета-тега визначається набором його атрибутів, які задаються в тезі <meta>.

Мета-теги розміщують в блоці <head> ... </ head> вебсторінки. Вони не є обов'язковими елементами, але можуть бути дуже корисні.

Приклад опису метаданих:

<head>

<meta name = "author" content = "рядок"> - автор вебдокумента

<meta name = "date" content = "дата"> - дата останнього зміни вебсторінки

<meta name = "copyright" content = "рядок"> - авторські права

<meta name = "keywords" content = "рядок"> - список ключових слів

<meta name = "description" content = "рядок"> - короткий опис (реферат)

<meta name = "robots" content = "noindex, nofollow"> - заборона на індексування

<meta http-equiv = "content-type" content = "text / html; charset = UTF-8"> - тип і кодування

<meta http-equiv = "expires" content = "число"> - управління кешуванням

<meta http-equiv = "refresh" content = "число; URL = адреса"> - перенаправлення

</ head>

Приклад вебсторінки:

```
<!DOCTYPE HTML>
<html lang=“ua”>
<head>
  <meta http-equiv="content-type" content="text/html; charset="windows-1251">
  <title>Чому?</title>
</head>
<body>
<h1>Чому?</h1>
```

```
<h2>Чому? Чому? Чому?</h2>
<p>Тому:</p>
<ul>
  <li>1111;</li>
  <li>2222;</li>
  <li>3333;</li>
  <li>4444.</li>
</ul>
<p>Тому Тому</p>
<p>Тому Тому Тому Тому …</p>
</body>
</html>
```

Результат у браузері (рисунок 1.48):

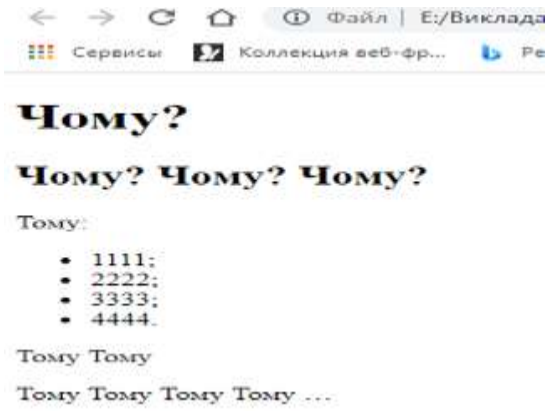


Рисунок 1.48 – Результат

Порядок виконання лабораторної роботи

1) ознайомитися з теоретичними відомостями;
2) виконати завдання;
3) оформити звіт;
4) продемонструвати результат на комп'ютері і захистити роботу.

ЛАБОРАТОРНИЙ ПРАКТИКУМ №2 HTML. Структура документа. Заголовки. Гіперпосилання. Форматування тексту. Кольори. Списки. Зображення. Фон. Таблиці, фрейми.

Мета: навчитись працювати з різними тегами, для оформлення вмісту вебдокумента.

Завдання

1) В тілі сторінок додати по декілька абзаців тексту (відповідно до тематики) використовуючи тег . Використовуючи гіперпосилання, реалізувати переходи між сторінками, на певний рядок поточною сторінки, певний рядок іншої сторінки, на сторінку в інтернеті). Застосувати атрибут, який задає колір гіперпосилань.

2) Відформатувати текст сторінок (виділення курсивом, напівжирним). Додати декілька картинок <img>. За допомогою атрибутів width і height зменшити і збільшити розмір зображення в 2 рази. Зробити картинку гіперпосиланням: при натисканні на картинку повинен відкриватися повнорозмірний варіант в новому вікні. Додати список, використовуючи різні типи форматування (нумерація , маркування , визначення <dl></dl>). Застосувати різні типи маркерів type. Додати графічний фон на сторінки (картинку або колір).

3) На сторінці додати таблицею 3 * 4. Залити кольором шапку з заголовками колонок. Додати заголовок до таблиці. В комірки першого рядка вставити картинки, другого рядка - гіперпосилання, 3 рядка - текст.

4) Створити складну таблицю, застосувавши різні види вирівнювання для різних рядків, об'єднання комірок (rawspan, calspan), групувння комірок(colgroup, col), використати відступи (cellspacing, cellpadding).

5) Вставити плаваючий фрейм

6) Завантажити проєкт на віддалений репозиторій (на GitHub).

Теоретичні відомості

Тег (html-тег, тег розмітки) [6] - управляюча символьна послідовність, яка задає спосіб відображення гіпертекстової інформації. HTML-тег складається з імені, за яким може йти необов'язковий список атрибутів. Весь тег (разом з атрибутами) заключається в кутові дужки <>: **<Імя_тега [атрибути]>**.

Як правило, теги є парними і складаються з початкового та кінцевого тегів, між якими і можна відслідковувати. Ім'я кінцевого тега збігається з ім'ям початкового, але перед ім'ям кінцевого тега ставиться риска / (<html> ... </ html>). Кінцеві теги ніколи не містять атрибутів. Деякі теги не мають кінцевого тега, наприклад тег <img>. Регістр символів для тегів не має значення.

Призначення html-тегів

– форматування тексту;
– опис кадрів та форм;
– форматування таблиць та списків;
– організації посилань на інші ресурси;
– вставка зображень та розширень HTML.

Приклади часто використовуваних тегів HTML:

<html> ... </ html> - контейнер гіпертексту;

<head> ... </ head> - контейнер заголовка документа;

<title> ... </ title> - назва документа (те, що відображається в заголовку вікна браузера);

<body> ... </ body> - контейнер тіла документа;

 ... </ div> - контейнер загального призначення (структурний блок);

<hN> ... </ hN> - заголовок N-ного рівня (N = 1 ... 6);

<p> ... </ p> - основний текст;

<a> ... - гіперпосилання;

 ... </ ol> - нумерований список;

 ... </ ul> - маркований список»

 ... </ li> - елемент списку;

<table> ... </ table> - контейнер таблиці;

<tr> ... </ tr> - рядок таблиці;

<td> ... </ td> - елемент таблиці;

<img> - зображення;

<form> ... </ form> - форма;

<i> ... </ i> - відображення тексту курсивом;

<b> ... </ b> - відображення тексту напівжирним шрифтом;

 ... </ em> - виділення (курсивом);

 ... </ strong> - посилення (напівжирним шрифтом);

 - примусовий розрив рядка;

<aside> </aside>— Визначає блок збоку від контенту для розміщення рубрик, посилань на архів, міток та іншої інформації»

<footer> </footer> - задає «підвал» сайту або розділу, в ньому може розташовуватися ім'я автора, дата документа, контактна та правова інформація»

<header> </header> - задає «шапку» сайту або розділу, в якій зазвичай розміщується заголовок.

Теги можуть бути вкладені, при цьому форматування внутрішнього тега має перевагу перед зовнішнім. При використанні вкладених тегів їх потрібно закривати, починаючи з останнього і рухаючись до першого:

```
<!-- Список як приклад використання вкладених тегів -->
<ol>
  <li>Елемент списку</li>
  <li>Другий елемент списку</li>
</ol>
```

```
<div>
  <h2>Заголовок другого рівня</h2>
  <p>та основний текст</p>
усередині логічного блоку
</div>
```

Атрибути - це пари виду «властивість = значення»:

<Тег атрибут = "значення"> ... </ тег>

Приклад: <h1 color="red" align="center">

Атрибути вказують в початковому тегу, кілька атрибутів поділяють одним або декількома пробілами, табуляцією або символами кінця рядка. Значення атрибута, якщо таке є, іде за знаком рівності, що стоїть після імені атрибута. Порядок запису атрибутів у тегу не важливий. Якщо значення атрибута - одне слово або число, то його можна казати після знака рівності, не виділяючи додатково. Всі інші значення необхідно брати в лапки "", особливо, якщо вони містять кілька слів, що розділені пробілами.

Незважаючи на необов'язковість лапок, їх все ж таки варто завжди використовувати.

Атрибути можуть бути обов'язковими та необов'язковими. Необов'язкові атрибути можуть не вказуватись, тоді для тега використовується значення цього атрибуту за замовчуванням. Якщо не вказано обов'язковий атрибут, вміст тега, швидше за все, буде відображено неправильно.

Короткий список деяких атрибутів і їх можливих значень:

style = "опис_стилів" - локальні стилі;

src = "адреса" - адреса (URI) джерела даних (наприклад картинки або скрипта);

align = "left | center | right | justify" - вирівнювання, за замовчуванням left (по лівому краю);

width = "число" - ширина елемента (в пікселях, піках, поінтах і ін.);

height = "число" - висота елемента (в пікселях, піках, поінтах і ін.);

href = "адреса" - гіперпосилання, адреса (URI) на який буде виконаний перехід;

name = "ім'я" - ім'я елемента;

id = "ідентифікатор" - унікальний (в межах вебсторінки) ідентифікатор елемента;

size = "число" - розмір елемента;

class = "ім'я_класу" - ім'я класу у вбудованій або пов'язаної таблиці стилів;

title = "рядок" - назва елемента;

alt = "рядок" - альтернативний текст.

Гіперпосилання - це особливим чином позначений фрагмент вебсторінки (текст, зображення та ін.), який пов'язаний з іншим документом. Для створення гіперпосилань використовується тег <a>. Гіперпосилання дозволяють переміщатися між пов'язаними вебсторінками.

Гіперпосилання можна розділити на наступні види:

– внутрішні - зв'язують документи всередині одного і того ж вузла;

– зовнішні - зв'язують вебсторінку з документами, що не належать даному вузлу;

– гіперпосилання на поштову адресу;

– мітки-якорі - дозволяють переходити відвідувачуі на певні розділи документа.

Перехід за посиланнями можна виконувати як на цілі документи, так і на спеціальним чином помічені (іменовані) фрагменти тексту:

 прив'язка до фрагменту тексту

 посилання на якір

 текст для кліку миші

 <IMG src = "посилання на рисунок">

Усередині тега <body> використовується атрибут, що задає колір гіперпосилань:

link - задає колір вихідних посилань;

vlink - задає колір відвіданих посилань;

alink - задає колір активних посилань (колір при натисканні миші).

Якщо потрібно зробити посилання на документ, який відкривається в новому вікні браузера, використовується атрибут target:

<a target="_blank" тега >

 нові надходження - перехід до рядка тієї ж сторінки, позначеної тегом <a name="new">

 примітки - перехід на сторінку сайту pag2 до рядка, позначеного тегом

<a name="new1">

 подробиці читайте друга сторінка - посилання на іншу сторінку того ж сайту

   - посилання на іншу сторінку того ж сайту, але посиланням є зображення.

 скачати програму - посилання з підказкою title

тест - зовнішнє посилання

Посилання можуть бути абсолютними і відносними.

Абсолютні посилання вказують, як правило, на зовнішній ресурс. Для них потрібно вказувати повний шлях:

 Абсолютне посилання

 Посилання на сторінку в каталозі

Відносні посилання, навпаки, використовують для переходу на внутрішні сторінки сайту. Для них потрібно вказувати шлях відносно сторінки, яка в посиланні:

Посилання на сторінку в кореневому каталозі

Посилання на фрагмент сторінки в поточному каталозі

Посилання на сторінку в підкаталозі поточного каталогу

Посилання на сторінку в підкаталозі кореневого каталогу

Посилання на сторінку у вищезазначеному каталозі.

Колір в HTML може бути заданий ключовими словами – назвами кольорів англійською мовою.

Альтернативним способом завдання кольору є вказівка коду кольору в системі RGB (від англ. Red, Green, Blue – червоний, зелений, синій).

Колірна модель RGB це одна з найпоширеніших моделей, що часто використовуються. Вона застосовується в приладах, що випромінюють світло, наприклад монітори, проєктори, фільтри та інші подібні пристрої.

FF0000 - яскраво-червоний (red)

00FF00 - яскраво-зелений (green)

0000FF - яскраво-синій (blue)

FFFF00 - жовтий (yellow) - суміш червоного і зеленого

000000 - чорний (black)

FFFFFF - білий (white)

Значення кольору вказується в тезі після символу решітки #:

Наприклад для тексту:

<font color = "# 808080"> сірий текст </ font>

Для фону всієї сторінки в тезі body атрибут bgcolor

<body bgcolor = "# FFFF00"> фон </ body>

Форматування тексту

Форматувати текст можна традиційними способами: виділяти курсивом, напівжирним, вибирати шрифт, розмір, колір, вирівнювати текстові фрагменти.

HTML дозволяє управляти відображенням тексту на сторінці.

<b> ... </ b> - виділення тексту жирним

<i> ... </ i> - виділення тексту курсивом

<u> ... </ u> - підкреслення тексту

<sub> ... </ sub> - форматувати текст як нижній індекс

<sup> ... </ sup> - форматувати текст як верхній індекс

 ... </ center> - вирівнювання тексту по центру

<pre>т е с т</pre> - зберігає вихідний вигляд відформатованого тексту з пробілами, розбивками на рядки і т.д. (т е с т)

<h3 align=left> ШЛЯХ ДО МУДРОСТІ </h3>

<pre><font size=3>

ШЛЯХ ДО МУДРОСТІ;

ШЛЯХ ДО МУДРОСТІ ;

ШЛЯХ ДО МУДРОСТІ ;

ШЛЯХ ДО МУДРОСТІ ;

ШЛЯХ ДО МУДРОСТІ ;

ШЛЯХ ДО МУДРОСТІ ;

ШЛЯХ ДО МУДРОСТІ ;,

ШЛЯХ ДО МУДРОСТІ ;

</pre>

Результат у браузері (рисунок 2.1):

Рисунок 2.1 – Результат використання <pre> </pre>

Форматування заголовків

<hN> ... </ hN> - вкладений текст, є заголовком документа рівня N, N приймає значення від 1 до 6. Найбільшим заголовком є <h1>, найменшим <h6>.

Форматування абзаців

** текст ** - новий абзац, можна використовувати лише початковий тег, наступний елемент означає не тільки початок наступного абзацу, але й кінець попереднього.

Атрибут **align = "..."** - визначає режим вирівнювання тексту:

left - по лівому краю (за замовчуванням)

center - по центру

right - по правому краю

justify - по ширині

Приклади вирівнювання абзаців:

за замовчуванням вирівнювання по лівому краю

центрування
всіх рядків абзацу,
 включаючи примусові розриви

 абзац вирівняний по правому краю

 абзац вирівняний по лівому краю і за замовчуванням

 цей абзац вирівняний одночасно по лівому та правому краям, але у старих версіях браузерів сприймається як вирівнювання по лівому краю

Результат у браузері (рисунок 2.2):

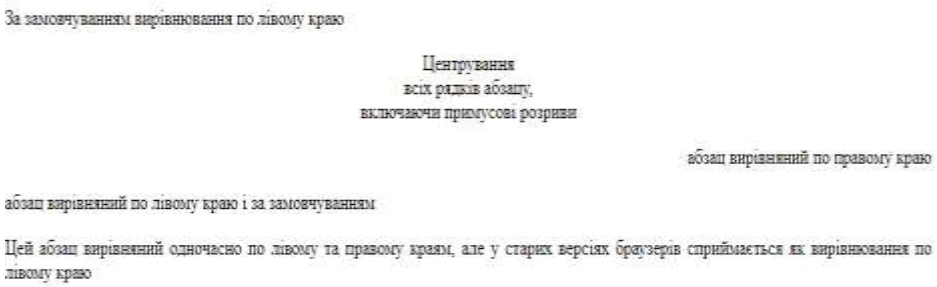


Рисунок 2.2 – Вирівнювання абзаців <p>

**тест
тест** - перенесення тексту на новий рядок без абзацного відступу.

тест<hr>тест - розділова лінія.

Атрибути тега hr:

align - встановлює вирівнювання роздільника (right, center чи left);

color – задає колір лінії;

size – визначає товщину лінії;

width - задає ширину лінії;

noshade - скасовує тривимірний ефект лінії.

Приклади лійній:

```
<hr noshade> заборона опуклості
<h2 align=center> приклади ліній</h2>
<hr size=2 width=80% align=center>
<p align=center>проста лінія товщиною 2 пікселя </p>
<hr size=20 width=20% align=center>
<p align=center>товста та коротка лінія </p>
<hr size=20 width=20% color=red noshade align=center>
<p align=center>така ж лінія червоного кольору</p>
 <p> <hr size=30 width=30 color=black noshade align=center>
<align=center>чорний квадрат 30x30 пікселів</p>
```

Результат у браузері (рисунок 2.3):

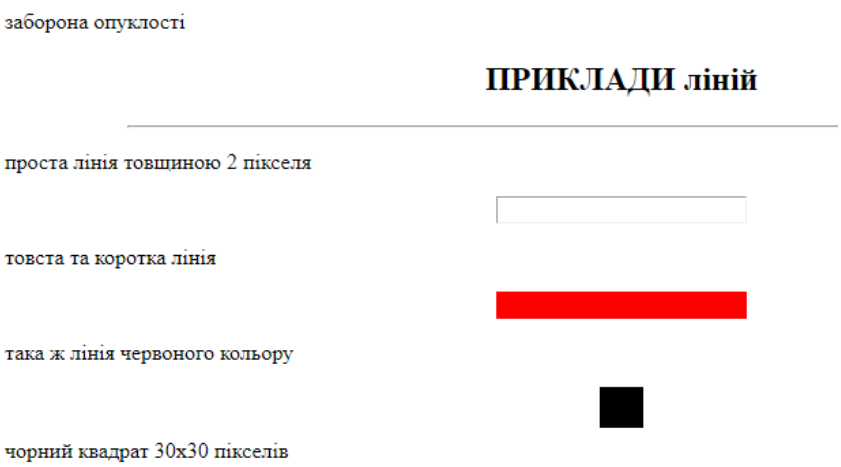


Рисунок 2.3 – Результат використання <hr>

Форматування списків

Маркований список:

перший елемент

другий елемент

Результат:

- перший елемент
- другий елемент

Спосіб маркування визначається атрибутом type:

type=“disc” - диск

type=“circle” - кружок

type=“square” - квадрат

Нумерований список:

перший елемент

другий елемент

Результат:

1. перший елемент
2. другий елемент

Спосіб нумерації визначається атрибутом type:

type="1" - 1, 2, 3, 4…

type="i" - i, ii, iii, iv…

Type="I" - I, II, III, IV…

type="a" - a, b, c, d…

type="A" - A, B, C, D…

Списки визначень:

<dl>

<dt>Термін</dt>

<dd>Визначення</dd>

</dl>

Результат:

Термін

Визначення

Фон сторінки

Для кольору фону на сторінці потрібно всередині початкового елемента вказати атрибут bgcolor = "колір". Колір визначається назвою або цифровим кодом.

<body bgcolor="blue">… </body>

<body bgcolor="00ff00">… </body>

Фонове зображення сторінки

Можна вказати адресу фонового зображення для сторінки в атрибуті background тега <body>.

Фонове зображення відображається у натуральну величину. Якщо розмір зображення менший за розмір вікна браузера, то зображення повторюється по горизонталі вправо і по вертикалі вниз.

<body background="bg1.jpg">

Зображення

Вставка зображень

IMG - це тег для створення посилання на графічний файл. З його допомогою можна також використовувати зображення в гіперпосиланнях, вставляти зображення в таблиці, використовувати зображення для фону сторінки. Не містить кінцевого тегу.

Обов'язковим атрибутом є **src** - шлях до графічного файлу.

Атрибут alt - виводить текст, який описує зображення, що завантажується.

Висоту та ширину зображення задають за допомогою атрибутів height та width (у пікселях або відсотках).

Рамка навколо об'єкта визначається атрибутом border.

border="0" – вимикає рамку

align="…" – визначає режим вирівнювання зображення відносно тексту в рядку:

top – по верхньому краю

middle – по центру рядка

bottom – по нижньому краю (за замовчуванням)

left – по лівому краю вікна

right – по правому краю вікна.

Браузер визначає розмір зображення автоматично. Для прискорення завантаження рекомендується вказувати розмір зображення атрибутами height та

width, щоб браузер не обчислював цей розмір після завантаження зображення. Також цими атрибутами можна розтягнути/стиснути зображення по горизонталі/вертикалі, але таке масштабування призведе до втрати якості.

<img src="1.gif" border=2 height =121 width=76 alt="портрет">

<img src="pic.gif" > зображення у центрі, текст зверху та знизу

<img src="pic.gif" align=left> зображення зліва

<img src="pic.gif" align=right> зображення

 <img src="посилання на рисунок">

Таблиця в HTML - це сукупність даних, розташованих і пов'язаних між собою за допомогою комірок, що розміщуються в рядках і колонках. Таблиця заповнюється даними через підрядник. Для вставки таблиць визначено 3 основних тега:

<table></table> - визначає початок/кінець таблиці

<tr></tr> - створює рядок таблиці

<td></td> - створює комірку

Вміст комірок поміщається в теги <td> ... </td>, які, в свою чергу, поміщаються в теги рядків <tr> ... </tr>, а вони вже - в тег <table> ... </table>.

Усі інші елементи таблиці – текст, зображення, списки – мають бути вкладеними у тег <td>…</td>.

Кількість тегів <tr>…</tr> визначає кількість рядків.

У кожному тезі рядка має бути таке ж число тегів <td>…</td> , рівне числу стовпців, інакше таблиця відобразиться неправильно.

Можна створювати вкладені таблиці: вкладати таблицю в комірку іншої таблиці (рисунок 2.4).

<table>

<tr> <td>1</td> <td>2</td> <td>3</td> </tr>

<tr> <td>4</td> <td>5</td> <td>6</td> </tr>

</table>

Рисунок 2.4 – Таблиця

<table>…</table> - визначає початок та кінець коду таблиці.

Атрибути:

align="…" - визначає режим вирівнювання таблиці. Значення: left, right, center

valign="…" - вирівнює текст у таблиці по вертикалі. Значення: top, bottom, middle, baseline

bordercolor = "колір" – колір рамки

bordercolorlight = колір - колір рамки зліва та зверху

bordercolordark = колір - колір рамки праворуч і знизу

title =“Текст“ - спливаюча підказка

width = число - ширина таблиці у відсотках чи пікселях.

Якщо ширина таблиці спочатку не задана, вона обчислюється виходячи з вмісту комірки.

Якщо ширина задана атрибутом width, наприклад width="100%", то ширина таблиці в такому випадку дорівнює ширині вікна.

Якщо атрибутом width число, то браузер розставляє переноси слів у тексті комірок в такий спосіб, щоб дотримати заданий розмір.

Атрибути для <tr>…</tr> та <td>…</td>:

align="…" – визначає режим вирівнювання вмісту комірок рядка left, center, right, justify, char - дозволяє вирівняти вміст комірки за заданим символом. Працює тільки в тому випадку, якщо значення атрибуту align встановлено як char.

background="URL" – URL зображення, яке заповнить фон комірок

bgcolor="колір" – колір фону комірок

valign="…" – визначає режим вирівнювання вмісту комірок рядка по вертикалі top, middle, bottom.

height="N" – висота комірки в пікселях

width="N" – ширина комірки в пікселях або відсотках від ширини таблиці.

Таблиці можна задати:

padding - заповнення,

spacing - відступ,

stuff – вміст.

Таблиці мають два атрибути, пов'язані з відступами:

cellspacing - порожнє місце між навколо вмісту комірки (рисунок 2.5):

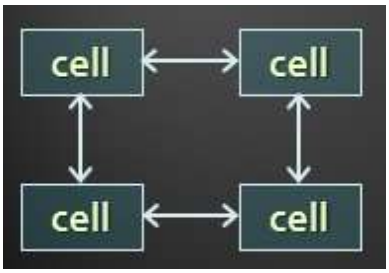


Рисунок 2.5 – cellspacing

cellpadding - порожнє місце між навколо вмісту комірки (рисунок 2.6):

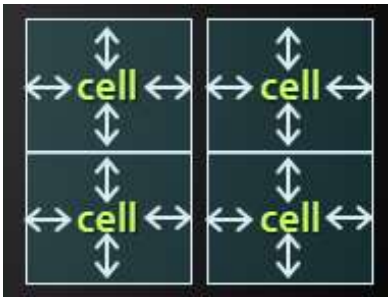


Рисунок 2.6 – cellpadding

Приклад:

```
<html>
 <head><title>Table Cells</title></head>
 <body>
  <table cellspacing="15" cellpadding="0">
   <tr><td>First</td>
   <td>Second</td></tr>
  </table>
  <br/>
  <table cellspacing="0" cellpadding="10">
   <tr><td>First</td><td>Second</td></tr>
  </table>
 </body>
</html>
```

Результат:

This table has cellspacing 15 and cellpadding 0

First Second

This table has cellspacing 0 and cellpadding 10

First Second

Рисунок 2.7 – Приклад використання cellpadding та cellspacing

Заголовок таблиці можна створити за допомогою відомих вам тегів <н1>—<н6>, але краще використовувати тег <caption>, який створює заголовок у таблиці.

Існує два види комірок у HTML таблицях комірки даних – вміст таблиці:

<td>…</td>

комірки заголовки – використовуються для підпису назв колонок:

<th>…</th>

```
<table width="100%" border="1" cellspacing="0" cellpadding="4">
<caption>… </caption>
<tr><th>…</th><th>…</th></tr>
<tr><td>…</td><td>…</td></tr>
<tr><td>…</td><td>…</td></tr>
</table>
```

Допускається також вкладення таблиць одна в іншу, тобто вмістом комірки може бути інша таблиця.

Приклад:

```
<table border="3">
   <tr>
    <td>Contact:</td>
    <td>
      <table border="3">
        <tr>
          <td>First name:</td>
          <td>Last name</td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```

Результат (рисунок 2.8):

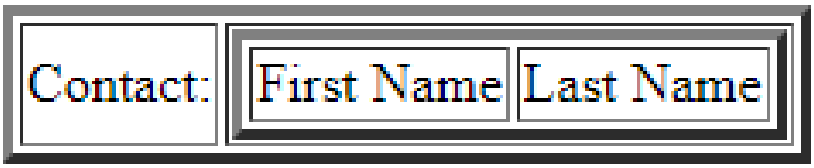


Рисунок 2.8 – Вкладені таблиці

Об'єднання комірок

Іноді буває необхідно створити в таблиці комірку, яка займає кілька стовпців або рядків. Для об'єднання комірок у тегах <td> та <тн> встановлюються такі атрибути:

colspan - визначає, скільки стовпців займає комірка (рисунок 2.9):

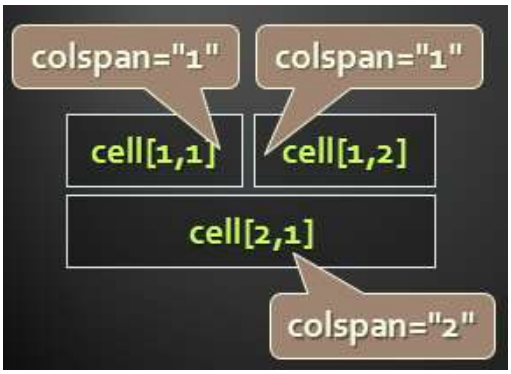


Рисунок 2.9 – colspan

rowspan - визначає, скільки рядків займає комірка (рисунок 2.10):

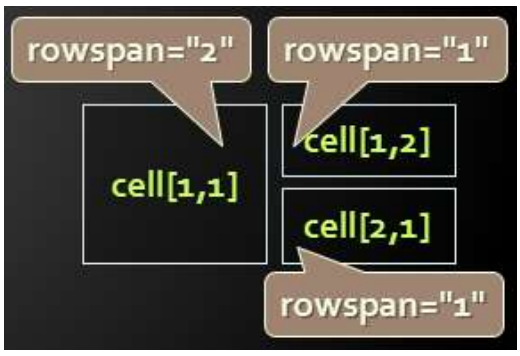


Рисунок 2.10 – rowspan

Часто при об'єднанні комірок допускають помилку, в результаті якої, один зі рядків виявляється довшим за інші рядки таблиці.

Слід пам'ятати таке:

– при об'єднанні n комірок за допомогю атрибуту colspan у поточному рядку залишається на n-1 комірку менше,

– а при використанні атрибуту rowspan число комірок скорочується на 1 в n-1 рядках нижче поточного.

Приклад:

```
<table cellspacing="0">
<tr class="1">
  <td>Cell[1,1]</td>
  <td colspan="2">Cell[2,1]</td>
</tr>
<tr class="2">
  <td>Cell[1,2]</td>
  <td rowspan="2">Cell[2,2]</td>
  <td>Cell[3,2]</td></tr>
<tr class="3">
  <td>Cell[1,3]</td>
  <td>Cell[2,3]</td></tr>
</table>
```

Результат (рисунок 2.11):

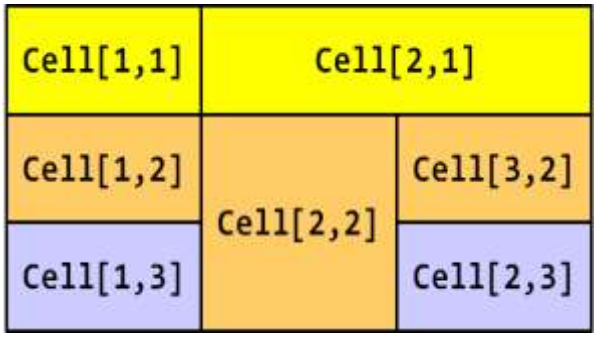


Рисунок 2.11 – Приклад використання colspan та rowspan

Групування стовпців таблиці

Це зручна функція, що дозволяє форматувати групи комірок як одне ціле, замість того, щоб встановлювати атрибути в тезі кожної індивідуальної комірки.

Установки атрибутів форматування в індивідуальних комірках мають більший пріоритет і скасовують установки для всієї групи.

Для групування використовуються теги.

<colgroup> - структурна група стовпців, застосовується для розбивки таблиці на стовпці різних типів, наприклад: стовпець заголовків та стовпці даних;

<col> - неструктурна група стовпців, застосовується довільного групування стовпців таблиці, які мають загальний формат даних.

<colgroup> та <col> містять набір атрибутів форматування.

Групування стовпців за допомогою <colgroup> та <col> відбувається на початку коду таблиці між <table> та першим рядком, створеним за допомогою <tr>.

В обох дескрипторах протяжність груп визначається за допомогою атрибуту span.

<colgroup span=2></colgroup> створює групу з двох перших стовпців таблиці.

Повторне введення <colgroup span=2></colgroup> створить наступну групу з двох стовпців - 3-го та 4-го.

Не можна створити групу з внутрішніх стовпців таблиці без створення групи попередніх стовпців.

<col> зазвичай використовується створення груп стовпців 2-го рівня всередині структурної групи, утвореної тегом <colgroup> (рисунок 2.12).

```
<table>
  <col атрибути>
  <tr>
   <td>...</td>
  </tr>
</table>
```

```
<table>
  <colgroup атрибути>
  <tr>
   <td>...</td>
  </tr>
</table>
```

Рисунок 2.12 – col, colgroup

Для додаткового форматування, установки параметрів форматування в тегу <col> мають більш високий пріоритет у порівнянні з однотипними установками в тегу <colgroup>.

Приклад:

```
<table>
<colgroup>
  <col style="width:100px"></col>
</colgroup>
<thead>
  <tr><th>Column 1</th><th>Column 2</th></tr>
</thead>
<tfoot>
  <tr><td>Footer 1</td><td>Footer 2</td></tr>
</tfoot>
<tbody>
```

```
  <tr><td>Cell 1.1</td><td>Cell 1.2</td></tr>
  <tr><td>Cell 2.1</td><td>Cell 2.2</td></tr>
</tbody>
</table>
```

Результат (рисунок 2.13):

Column 1	Column 2
Cell 1.1	Cell 1.2
Cell 2.1	Cell 2.2
Footer 1	Footer 2

Рисунок 2.13 –Використання col, colgroup

Плаваючий фрейм або лінійний фрейм, з'являється як окреме, плаваюче вікно для виведення інших документів. Він отримав свою назву з того факту, що може бути вбудованим в нормальний потік елементів сторінки або може зміщуватися вліво або вправо на сторінці з оточуючим його текстом. Фрейм може виводити один документ або може бути місцем, де виводяться кілька з'єднаних документів. Наприклад, кілька посилань на сторінці можуть виводити різні зображення у фреймі.

Плаваючі фрейми створюють за допомогою тега **<iframe>**, загальна форма якого показана на лістингу.

```
<iframe
  src = "url" // сторінка для початкового завантаження у фрейм
  name = "framename" // привласнює фрейму ім'я в якості вказівника для посилань
  frameborder = "1 | 0" // рамка
```

scrolling = "auto | yes | no" // панель прокрутки

</ Iframe>

Наступний код використовується для виведення і активації плаваючого фрейма. Властивості таблиці стилів замінюють більшість атрибутів фрейму. В тезі <iframe> атрибут src не заданий, тому фрейм відкриється без виведення документа, залишаючи фрейм порожнім.

```
<iframe name="TheFrame" scrolling="no" style="width:225px; height:200px; float:right; margin-left:15px; border:ridge 5px"></iframe>
```

Порядок виконання лабораторної роботи

1) ознайомитися з теоретичними відомостями;
2) виконати завдання;
3) оформити звіт;
4) продемонструвати результат на комп'ютері і захистити роботу

ЛАБОРАТОРНИЙ ПРАКТИКУМ №3 CSS. Внутрішні стилі. Стилі рівня документу. Зовнішні стилі. Оформлення тексту, поля, заповнення, межі. Застосування стилів для таблиць і списків

Мета: навчитись створювати стилі (зовнішні, внутрішні, рівня документу), підключати таблиці стилів до hrml-документів, розібратись в пріоритетності стилів.

Завдання

1) Створити зовнішній CSS файл. Підключити його до всіх сторінок. Використовуючи селектори (класи, ідентифікатори, унікальний ідентифікатор) налаштувати стиль шрифта (розмір, колір, стиль, міжрядковий інтервал, вирівнювання) для заголовка (HN), для тіла (BODY), посилань, задайте для тега BODY фон властивістю background-color. Застосувати стиль рівня документу для перевизначення стилю для посилань. Застосувати внутрішній стиль до абзацу. Використати оголошення !important.

2) Додати в CSS файл стилі для списків (маркованих, нумерованих, визначень та до таблиці, використовуючи розміри, кольори, шрифти, поля, заповнення, межі, фон.

3) Завантажити проєкт на віддалений репозиторій (на GitHub).

Теоретичні відомості

CSS (Cascading Style Sheets - каскадні таблиці стилів) - технологія управління зовнішнім виглядом елементів (тегів) вебсторінки [5]. CSS надає набагато більше можливостей по оформленню сторінки, ніж HTML.

Наприклад, за допомогою стилів CSS можна прибрати у посилань підкреслення, зробити у таблиці пунктирні кордону або навіть поміняти курсор миші.

Зараз, CSS – це стандарт для оформлення зовнішнього вигляду всіх сайтів.

До переваг використання CSS відносяться:

- централізоване управління відображенням множини документів за допомогою однієї таблиці стилів;
- спрощений контроль зовнішнім виглядом вебсторінок;
- наявність розроблених дизайнерських технік;
- можливість використання різних стилів для одного документа, в залежності від пристрою, за допомогою якого здійснюється доступ до веб сторінці.

CSS було затверджено стандартом HTML 4.0 також цим стандартом було затверджено і об'єктну модель браузера (Browser Object Model- BOM).

Об'єктна модель браузера визначає вміст вебдокументу, тобто, сама модель є набором об'єктів, які описують зазначений вміст. Оскільки BOM є унікальна для кожного браузера, виникали проблеми з міжплатформними застосунками. Далі, на місце об'єктної моделі браузера прийшла об'єктна модель документа (Document Object Model - DOM), що описує стандарт представлення вебсторінок, як набору об'єктів.

В html-документі елементи (теги) можуть перебувати в рамках інших елементів. Відносини між вкладеними елементами можуть бути батьківськими, дочірніми і братерськими.

Дерево документа - уявна деревоподібна структура елементів в html-документі, синонім поняття об'єктна модель документа (DOM).

Батьківський елемент - елемент, що містить в собі елемент який розглядається.

<p> ... </ strong> </ p>, елемент <p> є батьківським по відношенню до .

Пращур - елемент на кілька рівнів вище і містить в собі елемент який розглядається.

<body> ... <p> ... </ strong> </ p> ... </ body>, <body> є пращуром strong.

Дочірній елемент - елемент, що знаходиться усередині іншого елемента.

<p> ... </ strong> </ p>, елемент є дочірнім по відношенню до <p>.

Нащадок - елемент, що знаходиться всередині елемента, що розглядається і знаходиться на кілька рівнів нижче.

<body> ... <p> ... </ strong> </ p> ... </ body>, є нащадком <body>.

Братерський елемент - елемент, який має спільний батьківський елемент з даним.

<p> ... </ strong> ... <img ...> </ p>, елементи <img> і є братніми.

Синтаксис CSS

У стилях задається набір правил відображення в парах «властивість - значення», і те, до яких елементів їх застосовувати (селектор):

селектор

{

властивість 1: значення1;

властивість2: значення2;

властивість 3: значеннння3 значення4;

}

Правила записуються всередині фігурних дужок і відокремлюються один від одного крапкою з комою. Між властивостями і їх значеннями ставиться двокрапка.

CSS, як і HTML, ігнорує пробіли. Можна додавати коментарі, поміщаючи їх між / * і * /.

Селектор визначає, до яких елементів (тегів) сторінки будуть застосовуватися правила, задані парами «властивість - значення».

Властивістю (атрибутом) стилю називається один із параметрів елемента вебсторінки (у окремих випадках схожі з атрибутами тегів в HTML, але це різні речі).

В якості селектора може бути:

– **назва тега** - тоді стиль застосується до всіх таким тегами:

a {font-size: 12pt; text-decoration: none} / * CSS-коду задає всіх посиланнях 12-й розмір шрифту і прибирає підкреслення * /

table {border: black solid 1px} / *у всіх таблиць рамка буде чорного кольору, суцільна (solid) та шириною1 піксель* /

– **кілька тегів через кому** - тоді стиль застосується для всіх перерахованих тегів:

h1, h2, h3, h4, h5, h6 {color: red} / * всі заголовки червоними * /

– **кілька тегів через пробіл** - звернення до нащадка через батька**:**

table a {font-size: 120%} / * правило відноситься до всіх тегів A, вкладених в тег TABLE. Розмір шрифту збільшиться на 20% від базового* /

– **id елемента**. У стилях унікальний ідентифікатор вказується після знаку # - правила застосуються до тегу з атрибутом id="ідентифікатор":

CSS

#supersize {font-size: 200%}

HTML

 COOK

Не можна вносити в документ кілька елементів з однаковим id!

– **класи** (.main, .header, .nav)**.** Часто потрібно, щоб стиль застосовувався не до всіх тегів на сторінці, а тільки до деяких елементів

(наприклад, не до всіх посилань на сторінці, а тільки до тих, які розташовані в меню сайту). Для цього використовуються класи тег.імя_класса {...}:

CSS

тег.ім'я_класу {властивість1: значення; властивість2: значення; ... }

HTML

<тег class="ім'я_класу" > … </тег>

Правила, вказані після такого селектора, діятимуть лишена на теги з атрибутом class="ім'я_класу".

Можна не вказувати ім'я тега, тоді правила будуть застосовуватись до всіх тегів з відповідним значенням атрибуту class.

Приклад 1:

Для всіх тегів з атрибутом class = "class1" додамо підкреслення тексту і зменшимо розмір шрифту, а для тега <a> приберемо підкреслення:

CSS

```
.class1 {text-decoration: underline;  font-size: 80%}
a.class1 {text-decoration: none;}
```

У HTML-коді вкажемо для тегів ім'я класу:

```
<h1 class = "class1"> Мої улюблені сайти </ h1>
<a href="http://google.com" class="class1"> Google </a><br>
<a href="http://google.com" class="class1">Google</a><br>
<a href="http:// google .com" class="class1"> Google </a>
```

Результат у браузері:

Мої улюблені сайти

Google

Google

Google

Можна вказувати для одного елемента кілька класів через пробіл.

Приклад 2:

```
<title>Класи</title>
  <style>
   P { /* Звичайний абзац */
    text-align: justify; /* Вирівнювання тексту за шириною */
    }
   P.cite { /* Абзац із класом cite */
    color: navy; /* Колір тексту */
    margin-left: 20px; /* Відступ зліва */
    border-left: 1px solid navy; /* Рамка ліворуч від тексту */
    padding-left: 15px; /* Відстань від лінії до тексту */
    }
  </style>
 </head>
 <body>
  <p>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</p>
  <p class="cite">БББББББББББББББББББББББББББББББ</p>
 </body>
```

Результат у браузері (рисунок 3.1):

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

БББББББББББББББББББББББББББББББ

Рисунок 3.1 – Використання класів у якості селектора

Перший абзац вирівняний по ширині з текстом чорного кольору (цей колір задається браузером за замовчуванням), а наступний, до якого застосовано клас з ім'ям cite - відображається синім кольором і з лінією зліва.

Можна, також, використовувати класи і без вказівки тега.

Приклад 3:

```
<title>Класи</title>
  <style>
   .gost {
    color: green; /* Колір тексту */
    font-weight: bold; /* Жирний */
   }
   .term {
    border-bottom: 1px dashed red; /* Підкреслення під текстом */
   }
  </style>
 </head>
 <body>
  <p>ЯЯЯЯ <span class="gost">ТИИИ</span>, МИИИИ
    <b class="term">ВИИИИ</b>!!!!.
  </p>
 </body>
```

Результат у браузері (рисунок 3.2):

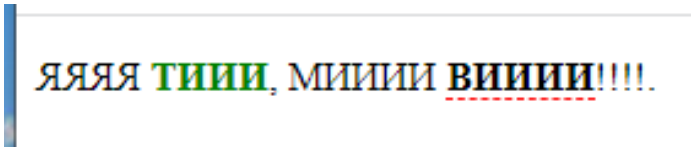


Рисунок 3.2 – Використання класів у якості селектора

Ідентифікатори, як правило використовуються для доступу і управління елементами веб-сторінки за допомогою скриптів

Класи застосовуватися для зміни стилів елементів.

Але немає різниці, через що задавати стилі, але потрібно звертати увагу на особливості ідентифікаторів і класів.

Ідентифікатори:

– у коді документа кожен ідентифікатор унікальний і повинен бути включений лише один раз;

– стиль через ідентифікатор має вищий пріоритет, ніж клас.

Класи:

– класи можуть використовуватися в коді неодноразово;

– класи можна комбінувати між собою, додаючи кілька класів до одного тегу.

Універсальний селектор

Іноді потрібно встановити одночасно один стиль для всіх елементів вебсторінки, наприклад, задати шрифт або зображення тексту. В цьому випадку допоможе універсальний селектор, який відповідає будь-якому елементу вебсторінки.

Символ * - універсальний селектор. Правила застосуються до всіх елементів документа.

* {Опис правил стилю}

У деяких випадках вказувати універсальний селектор не обов'язково. Наприклад, записи * .class і .class є ідентичними за своїм результатом.

```
<title>Універсальний селектор</title>
<style>
* {
font-family: Arial, Verdana, sans-serif; /*Рублений шрифт для тексту */
font-size: 96%; /*Розмір тексту */
}
</style>
```

Стилі CSS можуть включатися в HTML-документ 3 різними способами:

Зовнішні стилі (глобальні) - зберігаються в окремому файлі .css. підключаються тегом:

<link rel = "stylesheet" type = "text / css" href = "адреса_стилю">

Основна перевага: один стиль може використовуватися відразу в декількох документах HTML.

У зовнішніх файлах потрібно зберігати стилі, загальні для всього сайту, вони впливають відразу на безліч тегів у великій кількості документів. Це стає дуже зручним, якщо сайт містить багато сторінок. Наприклад, ми хочемо поміняти на всіх сторінках сайту колір фону і розмір шрифту. Якщо все сторінки підключають один і той же зовнішній стиль CSS, досить в ньому задати новий колір фону і розмір шрифту. Інакше доведеться редагувати кожну сторінку окремо. Якщо на сайті кілька десятків, то це стає дуже трудомістким завданням.

CSS-файл може знаходитися і на іншому сайті - у цьому випадку необхідно вказати його абсолютний URL-адресу.

Приклад:

Створимо файл style.css:

```
.class1 {text-decoration: underline;  font-size: 80%}
a.class1 {text-decoration: none;}
```

Створимо саму сторінку links.html:

```
<html>
<head>
<link rel="stylesheet" type="text/css“ href="style.css">
</head>
<body>
<h1 class="class1">My favorite Sites</h1><br>
<a href="http:// google.com" class="class1"> Google 1</a><br>
<a href="http://google.com" class="class1">Google 2</a><br>
```

```
<a href="http:// google.com" class="class1"> Google 3 </a>
</body>
</html>
```

Результат у браузері (рисунок 3.4):

Рисунок 3.4 – Зовнішні стилі

При відкритті цієї сторінки браузер клієнта завантажить також файл style.css і застосує правила CSS до документа.

Наприклад, за допомогою CSS можна відключити у посилань підкреслення. Засобами HTML цього зробити неможливо. CSS значно розширює можливості оформлення сторінки.

Використання CSS дозволяє розділити оформлення та вміст документа.

У нашому прикладі правила оформлення містяться у файлі style.css, а вміст – у links.html. Такий поділ спрощує редагування сайту надалі.

Рекомендується для оформлення використовувати лише засоби CSS, відмовитися від використання таких тегів, як <font> , <s> , <u> , , атрибутів align, border, color, height, widthi т.д.

Стилі рівня документа (зв'язані) - застосовуються до всього документа, записуються всередині тега <style> ... </ style>, який вкладається в тег <head> ... </ head> в документі HTML.

Такий спосіб оформлення стилів використовується, коли потрібно застосувати однакові стилі відразу до безлічі HTML-елементів (тегів) в одному документі.

Приклад:

```
<html>
<head>
<link rel="stylesheet" type="text/css"href="style.css">
<style>
a {color: red; text-decoration: none;}
</style>
</head>
<body>
<h1 class="class1">Мої улюблені сайти</h1>
<a href="http:// google.com " class="class1"> Google 1 </a><br>
<a href="http://google.com" class="class1">Google 2 </a><br>
<a href="http:// google.com " class="class1"> Google 3</a>
</body>
</html>
```

Результат у браузері (рисунок 3.5):

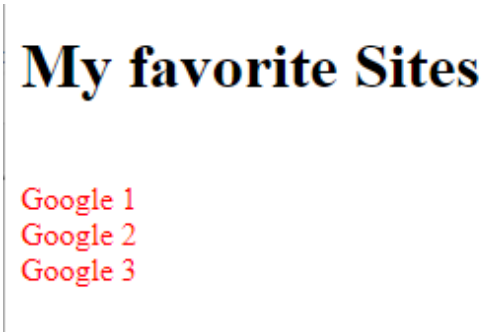


Рисунок 3.5 – Стилі рівня документу

Внутрішні стилі (inline) - використовуються, коли потрібно вказати стилі конкретного єдиного елемента.

Внутрішній стиль записується в атрибуті style і застосовується тільки до вмісту цього тега:

<p style="color: red;"> ... </ p>

Бажано не використовувати такий спосіб завдання стилю, бо він не відповідає принципу поділу вмісту та оформлення.

Імпорт CSS. У поточну стильову таблицю можна імпортувати вміст CSS-файлу за допомогою команди @import. Цей метод допускається використовувати спільно зі зв'язаними або глобальними стилями, але не з внутрішніми стилями.

@import url ("ім'я файлу") типи носіїв;

@import "ім'я файлу" типи носіїв;

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

Оголошення **!important**. Якщо ви зіткнулися з екстреним випадком і вам необхідно підвищити значимість будь-якої властивості, можна додати до неї оголошення !important:

p {color: red !important;}

p {color: green;}

Також !important перекриває внутрішні стилі. Занадто часте застосування !important – не добре. В основному, дане оголошення прийнято використовувати лише тоді, коли конфлікт стилів не можна перемогти іншими способами.

Порядок застосування стилів

Каскадність CSS – це механізм, завдяки якому до елементу HTML-документа може застосовуватися більш ніж одне правило CSS. Правила можуть виходити з різних джерел: із зовнішньої та внутрішньої таблиці стилів, від механізму спадкування, від батьківських елементів, від класів та id, від селектора тега, від атрибуту style і т. д.

Оскільки в цих випадках часто відбувається конфлікт стилів, була створена система пріоритетів: в кінцевому варіанті застосовується той стиль, який походить від селектора (inline id, class, tag) з вищим пріоритетом.

На рисунку 3.6 вказана вага кожного вага (значимість) кожного селектора. Чим більше вага, тим вище пріоритет.

Селектор тега:	1
Селектор класса:	10
Селектор ID:	100
Inline-стиль:	1000

Рисунок 3.6 – Значимість селекторів

Коли селектор складається з кількох інших селекторів, необхідно порахувати їхню загальну вагу.

Як обчислюється пріоритет: за кожен селектор додається 1 у відповідну комірку. В інших комірках стоять нулі. Щоб отримати загальну вагу, необхідно «об'єднати» всі числа в комірках (рисунок 3.7).

Селектор	ID	Клас	Тег	Загальна вага
p	0	0	1	1
.your_class	0	1	0	10
p.your_class	0	1	1	11
#your_id	1	0	0	100
#your_id p	1	0	1	101
#your_id .your_class	1	1	0	110
p a	0	0	2	2
#your_id #my_id .your_class p a	2	1	2	212

Рисунок 3.7 – Загальна вага селекторів селекторів

Якщо сталося так, що два селектори мають однакову вагу, то пріоритет надається тому стилю, який знаходиться нижче в коді.

Якщо для одного елемента заданий стиль і в зовнішній, і у внутрішній таблицях, то пріоритет віддається стилю в таблиці, яка знаходиться нижче в коді.

Наприклад, у внутрішній таблиці стилів заданий червоний колір для тегів <p>, а у зовнішній - зелений колір для цих же тегів. У HTML-документі ви підключили зовнішню таблицю стилів, а потім додали внутрішню таблицю за допомогою тега <style> </ style>. В результаті колір тегів <p> буде червоним.

Це - один із способів управляти значимістю стилів. Ще один спосіб підвищити пріоритет - спеціально збільшити вагу селектора, наприклад, додавши до нього id або клас.

Стиль, вказаний в атрибуті style, перекриває стиль, вказанийв тезі <style> або зовнішньому файлі CSS.

Приклад:

```
<html>
<head>
<style>
a {color: red; text-decoration: none;}
</style>
</head>
<body>
<a href= "http://INT.ua" Style="color:green">INT</a>
</body>
</html>
```

Результат у браузері (рисунок 3.8):

Рисунок 3.8 – Пріоритетність стилів

Селектор id (#) має більший пріоритет, ніж селектор класу(.)), а той, у свою чергу, більший, ніж звичайний селектор тега:

Приклад:

```
<html>
<head>
<style>
A {color: red; text-decoration: none; font-size: 120%} -A –3 пріоритет
.links {color: blue; text-decoration: underline} - . - 2 пріоритет
#greenlink{color: green}- # -1 пріоритет
</style>
</head>
<body>
<a href= "http://INT.ua" class="links“ id="greenlink“> INT.ua</a>
</body>
</html>
```

У браузері посилання буде зеленим і з підкресленням, розмір шрифту збільшений на 20%.

Іншою важливою особливістю CSS є те, що деякі атрибути успадковуються від батьківського елемента до дочірнього.

Наприклад, якщо атрибут font-size заданий для тега <body>, то він успадковується всіма елементами на сторінці. Коли властивість розміру задається у відсотках, воно буде обчислено виходячи з значення для батьківського елемента.

CSS-властивості: розміри, кольори, шрифти, текст [6].

Розміри в CSS можна задавати в різних одиницях виміру:

– em - поточна висота шрифту;

– pt - пункти (друкарський одиниця виміру шрифту);

– px - піксель. Пікселі це одиниці фіксованого розміру, які використовуються для всього, що читається на екрані. Один піксель дорівнює одній точці на екрані.;

– % - відсоток.

table {font-size: 12px}

Одиниці em залежать від розміру шрифту та можуть бути свої для кожного елемента у документі. Якщо елемент із шрифтом 2px, 1em означає 2px.

Вказівка розмірів (наприклад, для відступів) em означає, що вони задаються відносно шрифту і який би шрифт не був у користувача - великий (на великому екрані) або дрібний (на мобільному пристрої), ці розміри залишаться пропорційними. Одиниця em- масштабована одиниця, яка використовується для вебдокументів.

У CSS є ще одиниця rem 1 rem (від «root em», тобто. «кореневий em») - розмір шрифту кореневого елемента в документі. На відміну від em який може бути для кожного елемента свій, rem для всього документа той самий.

Шрифт – набір букв та знаків. У комп'ютері шрифт представляє собою файл, в якому описано, як повинні відображатися на екрані або друкованому варіанті різні символи: літери, цифри, знаки пунктуації та ін. Часто шрифти містять лише нарис для латинського алфавіту та не мають, Наприклад, підтримки кирилиці. Існують Unicode-шрифти, які містять символи всіх мов. Основні формати файлів шрифтів: TTF –TrueType та його розширення OTF –OpenType.

Типи шрифтів:

– serif - шрифти з засфчками: Times New Roman, Georgia;

– sans-serif - рубані шрифти (шрифти без засічок) Arial, Impact, Tahoma, Verdana;

– cursive - курсивні шрифти: Comic Sans MS;

– fantasy - декоративні шрифти, наприклад: Curlz MT.

– monospace - моноширинні шрифти, ширина кожного символу однакова: : Courier New, Lucida Console.

Засічками називають елементи на кінцях штрихів букв. Порівняємо букву шрифту Times New Roman і букву шрифту Arial. Використання шрифтів із засічками полегшує читання тексту на папері, тому такі шрифти зазвичай використовують для набору основного тексту в книгах. Для вебсайтів основний текст найчастіше набирають шрифтом без засічок.

Текст

CSS дозволяє управляти властивостями шрифту і тексту.

font-family - задає сімейство (тип) шрифта. Можна вказати кілька значень через кому. Браузер перевірить перший шрифт зі списку: якщо шрифт встановлений на комп'ютері користувача, то браузер застосує його, якщо немає - перейде до другого шрифту і т.д. Останнім в списку зазвичай вказується загальний тип шрифту serif, sans-serif, cursive, fantasy або monospace.

font-family: Georgia, Times New Roman, serif

font-size - розмір шрифту. Може здаватися абсолютним значенням в пунктах (pt) або пікселях (px) або відносним - у відсотках (%) або в em.

font-size: 12px

або

font-size: 150%

font-style - задаєє нарис (зображення) тексту:

– normal (звичайний);

– italic (курсив);

– oblique (нахилений).

Курсивний нарис є зміненою версією шрифту, який імітує рукописний текст з нахилом праворуч. нахилений нарис виходить зі звичайного нахилом букв.

font-weight - дозволяє змінити рівень жирності тексту: normal (звичайний), bold (напівжирний). Дія аналогічно тегу <b>.

color - задає колір тексту

h1, h2, h3, h4, h5, h6 {color: # ff0000}

або

h1, h2, h3, h4, h5, h6 {color: red}

line-height - міжрядковий інтервал (інтерліньяж), вказує відстань між рядками тексту. Може здаватися числом, як множник від поточного розміру шрифту, у відсотках, а також в пунктах (pt), пікселях (px) та інших одиницях виміру.

line-height: 1.5; / * Полуторний інтервал * /

text-decoration - задає оформлення тексту.

Варіанти:

– line-through (перекреслений);

– overline (лінія над текстом);

– underline (підкреслення);

– none (відключення ефектів).

a {text-decoration: none}

text-align - вирівнювання тексту в блоці:

– left (по лівому краю);

– center (по центру);

– right (по правому краю);

– justify (по ширині).

p {text-align: justify}

text-indent - відступ першого рядка («новий рядок»). Довжина відступу може здаватися в процентах (%) від ширини текстового блоку, пікселях (px), пунктах (pt) та ін.

p {text-indent: 1.25cm}

Властивості font-style, font-variant, font-weight, font-size, font-family і line-height можна задати в одному правилі:

font: font-style font-weight font-size / line-height font-family

Значення font-size і font-family є обов'язковими, решта можна не вказувати.

h1 {font: bold 14pt / 1.5 sans-serif}

Колір в CSS адається як і в HTML - шістнадцятковими цифрами: по 2 на кожен базовий колір моделі RGB (червоний, зелений, синій). Також можна використовувати стандартні назви кольорів англійською.

a.content {color: black}

a.menu {color: # 3300aa}

Допускається скорочувати шістнадцяткове значення до 3 цифр: запис #3300AA можна замінити на #30A. Також використовується конструкція rgb(…), яка дозволяє встановити червону, зелену та синю компоненти у десятковому (від 0 до 255) або відсотковому вигляді:

a.content {color: rgb(0%, 0%, 0%);}

a.menu {color: rgb(51,0,170);}

URL задаються конструкцією url (...)

body {background-image: url (images / bg.jpg);}

Приклад:

```
<html>
<head>
<title>ID</title>
<style type="text/css">
body {background-color: #c5ffa0;}
p {color: #0000ff; font-size:14px;}
#rose {color: #ff00ff; font: italic 16px Arial;}
</style>
</head>
<body>
<p>This is a normal paragraph</p>
```

```
<p id="rose"> And this paragraph is unique </p>
</body>
</html>
```

Результат у браузері (рисунок 3.9):

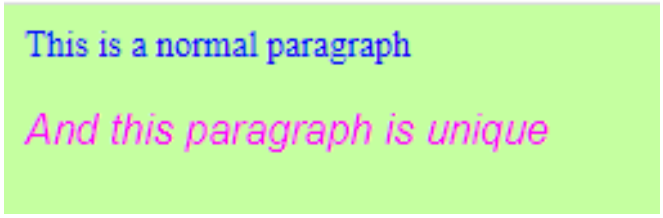


Рисунок 3.9 – Властивості CSS

У CSS кожен елемент знаходиться в блоці, якому можна задати значення:

– полів-відступів (**margin**);

– заповнення (**padding**);

– рамка (**border**).

Поле (margin) є відступом елемента від сусідніх заповнення (padding) - порожня область між рамкою та вмістом (рисунок 3.10).

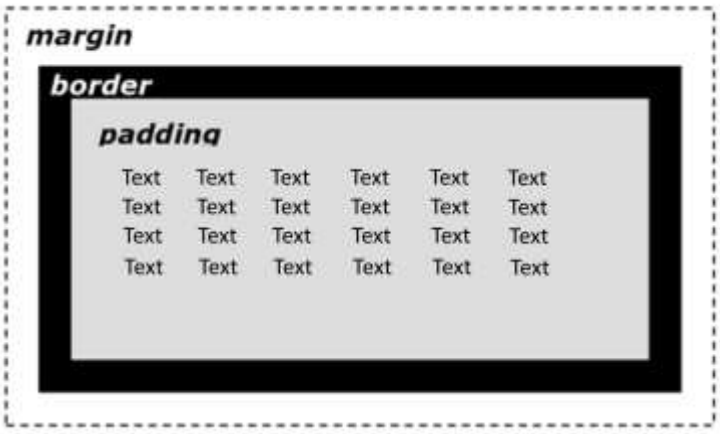


Рисунок 3.10 – Властивості margin, padding

Ширина полів (margin) і заповнення (padding) задається наступними CSS властивостями:

– margin-top, margin-right, margin-bottom, margin-left - для верхньої, правої, нижньої, лівої сторони поля;

– margin - скорочений запис. Задає значення відразу для всіх сторін.

CSS-властивості: поля, заповнення, кордони

В CSS кожен елемент розташовується в блоці, якому можна задати значення полів (margin), заповнення (padding) і кордони (border). Поле є відступом елемента від сусідніх, а заповнення - порожній областю між кордоном і вмістом (див. Рис. Нижче).

```
p {margin: 10px}
```

аналогічно запису

```
p {
margin-top: 10px;
margin-right: 10px;
margin-bottom: 10px;
margin-left: 10px;
}
```

Якщо для margin вказати:

– два значення через пробіл, то перше з них задаватиме ширину верхнього та нижнього поля, а друге – лівого та правого;

– три значення, то перше присвоюватиметься верхньому полю, друге – лівому і праве, а третє – нижньому;

– чотири значення, вони по черзі вказуватимуть верхнє, праве, нижнє та ліве поля.

padding-top, padding-right, padding-bottom,padding-left – встановлюють ширину заповнення зверху, праворуч, знизу та зліва від вмісту відповідно. padding - встановлює значення відразу для всіх сторін.

Для margin і padding можна задавати значення auto. В цьому випадку браузер сам автоматично розрахує величину полів та заповнення.

Для рамок border можна, встановити товщину, колір та стиль:

– border-width - товщина рамки;

– border-color - колір рамки (за замовчуванням – чорний);

– border-style - стиль рамки. Може приймати значення solid (за замовчуванням), dotted, dashed, double, groove, ridge, inset або outset.

Існує скорочений запис: властивість border задає одночасно товщину, колір та стиль. Значення вказуються через пробіл у будь-якому порядку.

текст

Можна задавати стилі окремо для верхньої, правої, нижньої та лівої межі, але це рідко використовується на практиці.

Можливо передавати в border-width, border-color і border-style не один, а до чотирьох параметрів, як для margin і padding.

Також існують властивості для товщини, кольору та стилю кожної межі: border-

Для визначення розміру CSS існують властивості **width** і **height**.

Частіше всього ширину та висоту задають у пікселях (px) або у відсотках (%) від ширини батьківського елемента. top-width, border-right-color, border-bottom-style.

Приклад:

```
<head>
<title>приклад</title>
<style>
P {font-size: 10pt}
#text1 {
border: 1px solid black; }
#text2 {
```

```
border: 1px solid black;
width: 300px;}
#text3 {
border: 1px solid black;
width: 50%;}
</style>
<body>
<p id="text1">111111111</>
<p id="text2">2222222222</>
<p id="text3">3333333333333</>
</body>
```

Результат у браузері (рисунок 3.11):

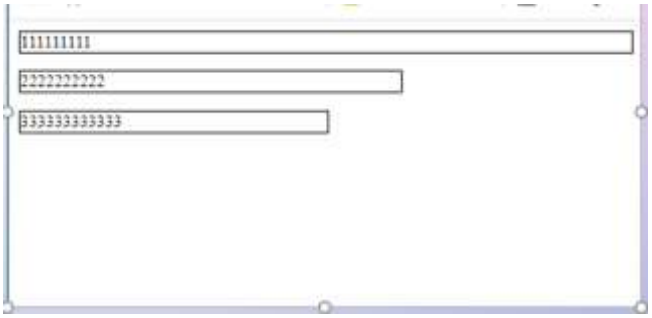


Рисунок 3.11 – Властивості border, width і height

Якщо користувач змінить розмір вікна, пропорційно зміниться ширина елементів, де вона була задана у відсотках. Якщо зменшити розмір вікна браузера, то у першому і третьому абзаці зменшиться ширина, а висота збільшиться, щоб вмістився весь текст, розмір другого абзацу залишиться незмінним, з'являться лінії прокручування.

Поведінка браузерів відрізняється, якщо для елемента задані і ширина, і висота, а вміст не вміщується в ці розміри.

Можна задавати мінімальні та максимальні розміри елемента властивостями: min-width, min-height і max-width, max-height.

Загальні розміри елемента складаються так:

Ширина = width + padding + border + margin

Висота = height + padding + border + margin

Тобто, width і height задають лише розміри вмісту, не включаючи поля, заповнення та рамки (рисунок 3.12).

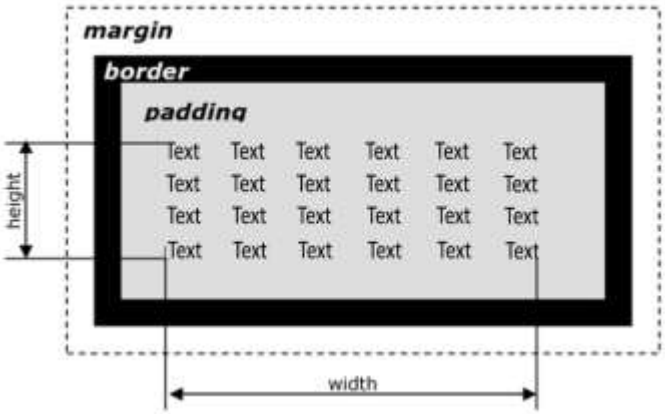


Рисунок 3.12 – Властивості width і height

Фон, як і в мові HTML, в CSS фоном може бути заливка кольором або зображення. Фонове зображення може бути повторюваним.

background-color – встановлює колір фона.

```
td.head {background-color: #ffff00;}
```

background-image – встановлює зображення в якості фону:

```
body {background-image: url(images/bg.jpg);}
```

background-attachment – задає поведінку фонового зображення під час прокручування:

- scroll - фон прокручується разом із вмістом.
- fixed - робить нерухомим фон.
- Inherit - успадковує значення батьківсього елементу.
- local - фон фіксується з урахуванням поведінки елемента.

За замовчуванням задається значення scroll. Якщо елемент має прокручування, то фон буде прокручуватись разом із вмістом, але фон, який виходить за рамки елемента, залишається на місці.

background-position - початкове положення фонового зображення по горизонталі (left, center, right) та вертикалі (top, center, bottom).

Замість ключових слів можна вказувати відстань у пікселях або відсотках.

background-position: <позиція>, <позиція>

де <позиція> = [left | center | right | <відсотки> | <значення>], | [top | center | bottom | <відсотки> | <значення>] |inherit.

top left = left top = 0% 0% (в лівому верхньому куті)

top= top center = center top = 50% 0% (по центру вгорі)

background-repeat – вказує, у якому напрямку має повторюватись фонове зображення:

- repeat – по горизонталі та вертикалі (за умовчанням);
- repeat-x - тільки по горизонталі;
- repeat-y - тільки по вертикалі;
- no-repeat - вимкнути повторення.

```
body{
background-image:url(‘11.jpg');
background-repeat: repeat-x;
background-position: 80px 100px;}
```

Таблиці. Властивості CSS можуть застосовуватися до таблиць, їх рядків та комірок для того, щоб задати властивості тексту та шрифту, керувати фоном, полями, межами, розмірами тощо.

Створимо таблицю та застосуємо до неї CSS-стилів. У таблицю впишемо текст.

Для заголовка таблиці використовуємо тег<th>…</th>.

Приклад:

```
<html> <head>
<title>Yaer/Browser
</title>
</head>
<body>
<table> <tr>
<th>Yaer/Browser</th>
<th>IE</th>
<th>Firefox</th>
<th>Chrome</th>
<th>Opera</th> </tr>
<tr> <td>2020</td>
<td>61.43%</td>
<td>24.40%</td>
<td>4.55%</td>
<td>2.37%</td> </tr>
<tr><td>2021</td>
<td>69.13%</td>
<td>22.67%</td>
<td>3.58%</td>
<td>2.18%</td> </tr>
 <tr> <td>2022</td>
<td>77.83%</td>
<td>16.86%</td>
<td>2.65%</td>
<td>1.84%</td> </tr>
<tr> <td>2023</td>
<td>79.38%</td>
```

```
<td>14.35%</td>
<td>4.70%</td>
<td>0.50%</td> </tr>
</table>
 </body> </html>
```

Без CSS-оформлення таблиця виглядатиме як на рисунку 3.13. За замовчування вміст комірок-заголовків відображається жирним шрифтом з вирівнюванням по центру.

Результат у браузері (рисунок 3.13):

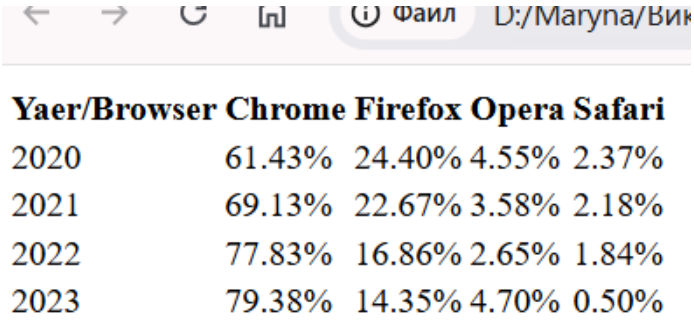


Рисунок 3.13 – Таблиця без CSS

Додамо у тег <head>…</head> тег <style>…</style>, а до тега <table>…</table> атрибут id="browser_stats".

Для комірок-заголовків встановимо сірий фон та відступ вмісту від бордера (padding) в половину висоти рядка, для комірок з даними – вирівнювання по правому краю та padding 0,3 від висоти рядка.

Навколо таблиці задаємо подвійну рамку, а для комірок – звичайну одинарну.

```
<style>
/*стиль таблиці*/
table#browser_stats {
border: 3px double black;
```

```
}
/*стиль комірок-заголовків*/
table#browser_stats th {
border: 1px solid black;
background-color: grey;
padding: 0.5em;
}
/*стиль комірок з даними*/
table#browser_stats td {
border: 1px solid black;
padding: 0.3em;
text-align: right;
}
</style>
```

Результат у браузері (рисунок 3.14):

Year/Browser	Chrome	Firefox	Opera	Safari
2020	61.43%	24.40%	4.55%	2.37%
2021	69.13%	22.67%	3.58%	2.18%
2022	77.83%	16.86%	2.65%	1.84%
2023	79.38%	14.35%	4.70%	0.50%

Рисунок 3.14 – Таблиця з CSS

Розділимо цю таблицю двома лініями на 3 частини: назви браузерів, роки та дані з відсотками. Назви браузерів та дані з відсотками вирівняємо по центру, роки – по правому краю. Задамо однакову ширину для стовпців з інформацією про браузери (дані з відсотками).

```
<html>
```

```
<head>
<title>Популярність…</title>
<style>
table#browser_stats {
border-collapse: collapse; }
table#browser_stats th {
border-bottom: 1px solid black; }
table#browser_stats td {
padding: 0.3em;
text-align: center;
width: 70px; }
.lc {
text-align: right;
border-right: 1px solid black;
width: 100px; }
</style>
</head>
<body>
<table>
<tr>
<table id="browser_stats">
<tr>
<th class="lc">Year/Browser
</th>
<th>Chrome</th>
<th>Firefox</th>
<th>Opera</th>
<th>Safari</th>
```

```
</tr>
<tr>
<td class="lc">2020</td>
<td>61.43%</td>
<td>24.40%</td>
<td>4.55%</td>
<td>2.37%</td>
</tr>
<tr>
<td class="lc">2021</td>
<td>69.13%</td>
<td>22.67%</td>
<td>3.58%</td>
<td>2.18%</td>
</tr>
<tr>
<td class="lc">2022</td>
<td>77.83%</td>
<td>16.86%</td>
<td>2.65%</td>
<td>1.84%</td>
</tr>
<tr>
<td class="lc">2023</td>
<td>79.38%</td>
<td>14.35%</td>
<td>4.70%</td>
<td>0.50%</td>
```

```
</tr>
</table>
</body>
</html>
```

Результат у браузері (рисунок 3.15):

Year/Browser	Chrome	Firefox	Opera	Safari
2020	61.43%	24.40%	4.55%	2.37%
2021	69.13%	22.67%	3.58%	2.18%
2022	77.83%	16.86%	2.65%	1.84%
2023	79.38%	14.35%	4.70%	0.50%

Рисунок 3.15 – Таблиця з CSS

Порядок виконання лабораторної роботи

1) ознайомитися з теоретичними відомостями;
2) виконати завдання;
3) оформити звіт;
4) продемонструвати результат на комп'ютері і захистити роботу.

ЛАБОРАТОРНИЙ ПРАКТИКУМ №4 CSS. Контекстні селектори. Сусідні селектори. Дочірні селектори.

Мета: розібратись чим відрізняються сусідні дочірні, контекстні селектори і навчитись їх застосовувати.

Завдання

1) Застосувати контекстні, сусідні, дочірні селектори
2) Завантажити проєкт на віддалений репозиторій (на GitHub).

Теоретичні відомості

Сусідні селектори

     <tt> </tt>.

Сусідними тут є теги та , а також та tt>. При цьому та <tt> До сусідніх елементів не належать через те, що між ними розташований контейнер .

Для керування стилем сусідніх елементів використовується символ плюс (+), який встановлюється між двома селекторами.

Селектор 1 + селектор 2 {Опис правил стилю}

Пробіли навколо плюса не обов'язкові, стиль при такому записі застосовується до Селектора 2, але тільки в тому випадку, якщо він є сусіднім для Селектора 1 і слідує відразу після нього. Важливо сусідство і порядок.

Приклад:

```
<style>
b + i {
color: red; /*Червоний колір тексту */
}
</style>
```

</head>

<body>

p>NE ZASTOSOVANO LllllllPpppppp ddddddd, ZASTOSOVANOaaaaaaa <tt>eeeeeee</tt>.

</body>

Результат у браузері (рисунок 4.1):

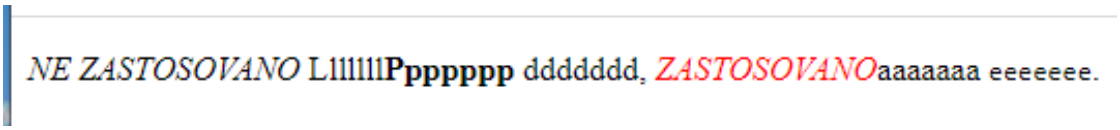


Рисунок 4.1 – Сусідні селектори

Контекстні селектори

При створенні вебсторінки часто доводиться вкладати одні теги всередину інших. Щоб стилі для цих тегів використовувалися коректно, допоможуть селектори, які працюють тільки в певному контексті. Наприклад, поставити стиль для тега тільки коли він розташовується всередині контейнера . Таким чином можна одночасно встановити стиль для окремого тега, а також для тега, який знаходиться всередині іншого.

Контекстний селектор складається з простих селектор розділених пробілом. Так, для селектора тега синтаксис буде наступний:

Тег1 Тег2 {...}

В цьому випадку стиль буде застосовуватися до Тегу2 коли він розміщується всередині Тега1, як показано нижче.

<Тег1>

<Тег2> ... </ Тег2>

</ Тег1>

Приклад:

<title>Контекстні селектори</title>

```
<style>
 p b {
  font-family: Times, serif; /* Сімейство шрифту */
  color: navy; /* Синій колір тексту */
 }
</style>
</head>
<body>
 <div><b>BOLD</b></div>
 <p><b>BOLD AND COLOR</b></p>
</body>
```

В даному прикладі показано звичайне застосування тега <b> і цього ж тега, коли він вкладений всередину абзацу <p>. При цьому змінюється колір і шрифт тексту.

Результат у браузері (рисунок 4.2):

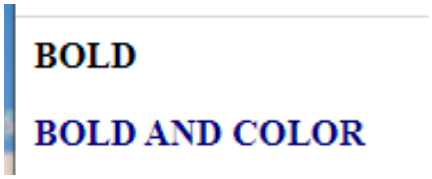


Рисунок 4.2 – Контекстні селектори

Не обов'язково контекстні селектори містять лише один вкладений тег. Залежно від ситуації можна застосовувати два і більше послідовно вкладених один в інший тегів. Тег1 Тег2 Тег3 Тег4 { ... }

Більш широкі можливості контекстні селектори дають під час використання ідентифікаторів та класів. Це дозволяє встановлювати стиль тільки для того елемента, який розташовується в межах певного класу.

клас1 Тег1 { ... }.

Не обов'язково контекстні селектори містять тільки один вкладений тег. Залежно від ситуації допустимо застосовувати два і більш послідовно вкладених один в одного тегів.

Ширші можливості контекстні селектори дають при використанні ідентифікаторів і класів. Це дозволяє встановлювати стиль тільки для того елемента, який розташовується усередині певного класу, як показано в наступному прикладі.

Приклад:

```
<title>Контекстні селектори</title>
  <style>
   a {
    color: green; /* зелений колір тексту для всіх посилань */
   }
   .menu {
    padding: 7px; /* поля навколо тексту */
    border: 1px solid #333; /* параметри рамки */
    background: #fc0; /* колір фону */
   }
   .menu a {
    color: navy;
   }
  </style>
 </head>
<body>
  <div class="menu">
   <a href="1.html">ukranian</a> |
   <a href="2.html">english</a> |
   <a href="3.html">german</a>
```

```
  </div>
  <p><a href="text.html">link</a></p>
 </body>
```

Результат у браузері (рисунок 4.3):

Рисунок 4.3 – Контекстні селектори з використанням ідентифікатора, класів

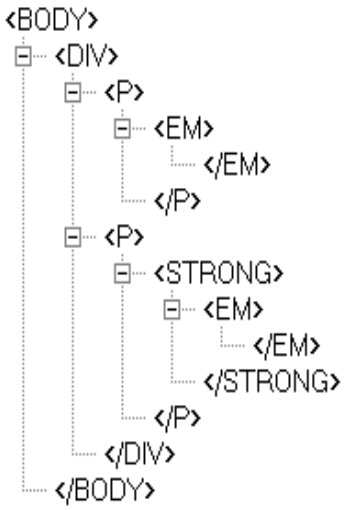
У цьому прикладі використовується два типи посилань. Перше посилання, стиль якого задається за допомогою селектора A, діятиме на всій сторінці, а стиль другого посилання .menu A застосовується лише до посилань всередині елемента з класом menu.

При такому підході легко керувати стилем однакових елементів (зображень та посилань), оформлення яких повинно відрізнятися у різних частинах веб-сторінки.

Дочірні селектори

Дочірнім називається елемент, який безпосередньо розташовується всередині батьківського елементу.

Напишемо код,_в якому застосовується кілька контейнерів, які в коді розміщуються один в одному. Наочніше це видно з дерева елементів, так називається структура відносин тегів документа між собою (рисунок 4.4).



:Рисунок 4.4 – Дерево елементів

Приклад 1:

```
<html>
 <head>
  <title>LABEL</title>
 </head>
 <body>
  <div>
    <p><em>111111111111111</em></p>
 <p><strong><em>222222222222</em></strong>, 333333333333333</p>
  </div>
 </body>
</html>
```

Результат у браузері (рисунок 4.5):

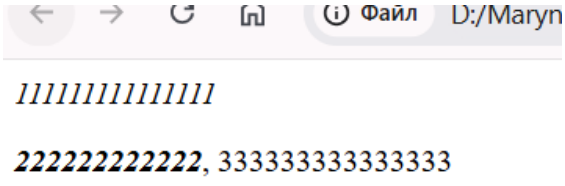


Рисунок 4.5 – Вкладені елементи

На рисунку в зручному вигляді представлена вкладеність елементів та їхня ієрархія. Тут дочірнім елементом по відношенню до тегу виступає тег <p>. Водночас тег не є дочірнім для тега оскільки він розташований у контейнері <p>.

Дочірнім селектором вважається такий, який у дереві елементів знаходиться прямо всередині батьківського елемента. Синтаксис застосування таких селекторів є наступним.

Селектор 1 > Селектор 2 { Опис правил стилю }

Стиль застосовується до Селектора 2, але тільки у тому випадку, якщо він є дочірнім для Селектора 1.

В нашого прикладі, стиль вигляду P > EM {color: red;} буде встановлений для першого абзацу документа, оскільки тег знаходиться всередині контейнера <p> і не дасть жодного результату для другого абзацу. А все через те, що тег у другому абзаці розташований у контейнері , тому порушується умова вкладеності.

За своєю логікою дочірні селектори схожі на контекстні селектори.

Різниця між ними:

– стиль до дочірнього селектора застосовується лише в тому випадку, якщо він є прямим нащадком, тобто, розташовується безпосередньо всередині батьківського елемента;

– для контекстного селектора допустимий будь-який рівень вкладеності.

Приклад 2:

```
<html>
<head>
    <title>дочірні серектори</title>
  <style>
   div i { /* контекстний селектор */
```

```
 color: green; /* зелений колір тексту */
 }
 p > i { /* дочірній селектор */
 color: red; /* червоний колір тексту */
 }
</style>
</head>
<body>
<div>
<p><i>1111111</i>, aaaaaaaaaaa <i>11111111</i>
aaaaaaaaaaa</p>
<i>222222222</i>
</div>
</body>
</html>
```

Результат у браузері (рисунок 4.6):

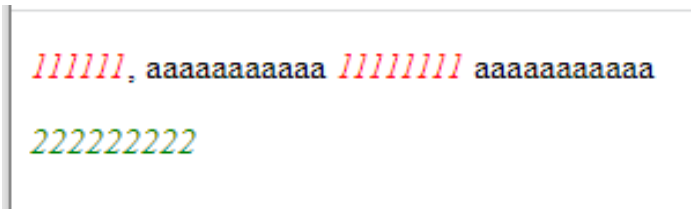


Рисунок 4.6 – Результат

На тег <i> у прикладі діють одночасно два правила: контекстний селектор (тег <i> розташований усередині) та дочірній селектор (тег <i> є дочірнім по відношенню до <p>). При цьому правила є рівнозначними, оскільки всі умови для них виконуються і не суперечать одна одній. У подібних випадках застосовується стиль, розташований у коді нижче, тому курсивний текст

відобразиться червоним кольором. Варто поміняти правила місцями та поставити DIV I нижче, як колір тексту зміниться з червоного на зелений.

Приклад 2:

```
<html>
<head>
   <meta charset="windows-1251">
  <title>Дочірні серектори</title>
  <style>
 P > I { /* Дочірній селектор */
    color: red; /* Червоний колір тексту */
   }
 DIV I { /* Контекстний селектор */
    color: green; /* Зелений колір тексту */
   }
  </style>
 </head>
 <body>
  <div>
   <p><i>1111111</i>, aaaaaaaaaa <i>11111111</i>
   aaaaaaaaaa</p>
   <i>222222222</i>
  </div>
 </body>
</html>
```

Результат у браузері (рисунок 4.7):

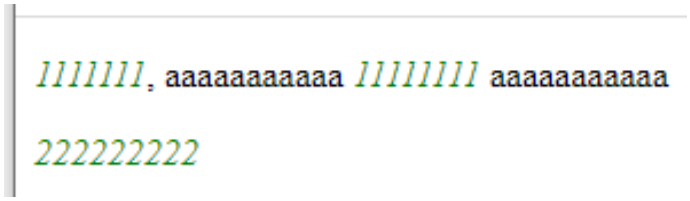


Рисунок 4.7 – Результат

В більшості випадків від додавання дочірніх селекторів можна відмовитися, замінивши їх контекстними селекторами. Однак використання дочірніх селекторів розширює можливості по управлінню стилями елементів, що в підсумку дозволяє отримати потрібний результат.

Найзручніше застосовувати зазначені селектори для елементів, які мають ієрархічну структуру: таблиці, списки.

Порядок виконання лабораторної роботи

1) ознайомитися з теоретичними відомостями;
2) виконати завдання;
3) оформити звіт;
4) продемонструвати результат на комп'ютері і захистити роботу

ЛАБОРАТОРНИЙ ПРАКТИКУМ №5 Блочні елементи. Рядкові елементи. Позиціонування. Псевдокласи. Псевдоелементи

Мета: ознайомитись з можливими варіантами позиціонування елементів на сторінці, розібратись, як будуть розміщені елементи на сторінці, при застосуванні різних видів позиціонування.

Завдання

1) Застосувати блочні елементи ... </ div>, рядкові елементи ... </ span>. Для позиціонування блоків застосувати властивість position. Зробити один або декілька блоків плаваючими, застосувавши атрибут float.

Використати:

- псевдокласи для посилань (:link, :visited, :active, :hover);
- псевдоклас :first-child;
- псевдоелементи: :before, :after.

2) Завантажити проєкт на віддалений репозиторій (на GitHub).

Теоретичні відомості

Псевдокласи

Є способи прив'язки правил оформлення CSS до елементів документа HTML: за назвою тега, по імені класу, по ID і т.д [5].

В CSS також існує кілька псевдокласів. За допомогою псевдокласів можна задати стиль в залежності від стану елемента або його положення в документі.

Для посилань визначено 4 псевдокласи:

:link - посилання, які не відвідувалися користувачем;

:visited - відвідані посилання;

:active - активна (натиснута) посилання;

:hover - посилання, на яку наведений курсор.

:focus - застосовується до елементу при отриманні ним фокусу.

Наприклад, для текстового поля форми отримання фокусу означає, що курсор встановлений в поле, і за допомогою клавіатури можна вводити в нього текст.

Приклад:

```
<html>
<head>
<title>test</title>
<style>
a:link, a:visited {
color: black;
font-family: verdana, sans-serif;
text-decoration: none;}
a:hover{
color: #de7300;
text-decoration: underline; }
</style>
</head>
<body>
<a href="index.html">one</a><br>
<a href="hobby.html">second</a><br>
<a href="photo.html">third</a><br>
</body> </html>
```

Результат у браузері (рисунок 5.1):

Рисунок 5.1 – Псевдокласи для посилань

Псевдоклас **focus** визначає стиль для елемента, що отримує фокус.

Приклад:

```
<html>
 <head>
<title>Pseudo_class</title>
  <style>
   input:focus {
    color: red;
   }
  </style>
 </head>
 <body>
  <form action="">
   <p><input type="text" value="black color"></p>
   <p><input type="text" value=" black color "></p>
  </form>
 </body>
</html>
```

Результат у браузері (рисунок 5.2):

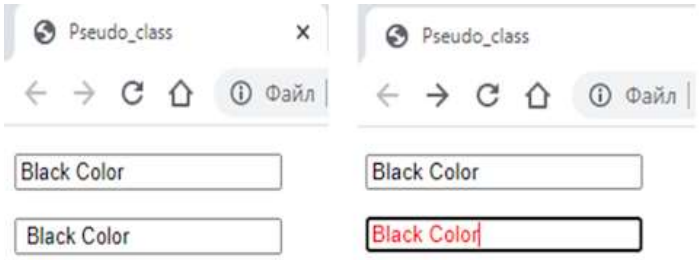


Рисунок 5.2 – Псевдоклас :focus

Псевдоклас **:first-child**, що має відношення до дерева документа.

До цієї групи належать псевдокласи, які визначають положення елемента в дереві документа і застосовують до нього стиль в залежності від його статусу. :first-child застосовується до першого дочірньому елементу селектора, який розташований в дереві елементів документа.

Приклад 1:

```
<html>
 <head>
  <title>Pseudo_class</title>
  <style type="text/css">
   b:first-child {
    color: red;    }
  </style>
 </head>
 <body>
<p><b>aaaa</b> 2222 <b>1111</b>
 2222 <b>1111</b> 2222</p>
  <p><b>bbbb</b> 2222 <b>1111</b>
 2222 <b>1111</b></p>
 </body>
</html>
```

Результат у браузері (рисунок 5.3):

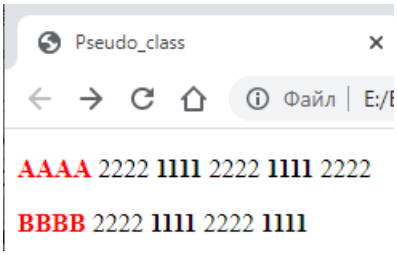


Рисунок 5.3 – Псевдоклас :first-child

В даному випадку псевдоклас :first-child додається до селектора B і встановлює червоний колір тексту. Контейнер <b> зустрічається кілька разів, червоним кольором буде виділено лише текст «AAAA». В інших випадках вміст контейнера <b> відображається чорним кольором. З наступним абзацом все починається знову, оскільки батьківський елемент змінився. Тому «BBBB» також буде виділено червоним кольором.

Псевдоклас :first-child найзручніше використовувати в тих випадках, коли потрібно задати різний стиль для першого і інших однотипних елементів. для абзаців можна додати відступ першого рядка. Для цього застосовують властивість text-indent зі значенням відступу.

Але, щоб змінити стиль першого абзацу і прибрати для нього відступ потрібно скористатися псевдокласом :first-child.

Приклад 2:

```
<html>
 <head>
  <title>Pseudo_class</title>
  <style>
   p {text-indent: 1em; /* indent first line */
   }
   p:first-child { text-indent: 0; /* For the first paragraph remove the indent */
   }
  </style>
 </head>
 <body>
  <p>11111111111111111111</p>
  <p>222222</p>
  <p>3333333333333</p>
 </body>
```

</html>

Результат у браузері (рисунок 5.4):

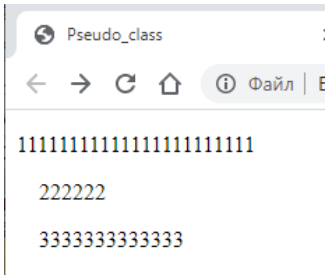


Рисунок 5.4 – Псевдоклас :first-child

Перший абзац тексту не містить відступу першого рядка, а для решти його встановлено.

Псевдоклас **:lang** , що задає мову тексту.

Для документів, що одночасно містять тексти кількома мовами, мають бути дотримані правила синтаксису, властиві для тієї чи іншої мови. За допомогою псевдокласів можна змінювати стиль оформлення іноземних текстів, а також деякі налаштування.

:lang визначає мову, яка використовується у документі. У коді HTML мова встановлюється через атрибут lang, він зазвичай додається до тега <html>. За допомогою псевдокласу :lang можна задавати певні налаштування, характерні для різних мов, наприклад, вид лапок у цитатах.

Елемент:lang(мова) { ... }

В якості мови можуть бути такі значення: ua- українська, en- англійська; de- німецька. Для відображення типових лапок в прикладі використовується стильова властивість quotes, а саме перемикання мови і відповідного виду лапок відбувається через атрибут lang, що додається до тегу <q>.

Приклад:

<html>

<head>

```
<title>lang</title>
<style>
p { font-size: 150%;}
 q:lang(de) {
quotes: "\201E" "\201C"; /*quotes for DE*/ }
 q:lang(en) {
quotes: "\201C" "\201D"; /* quotes for EN*/ }
 q:lang(FR) { /* quotes for FR*/
 quotes: "\00AB" "\00BB"; }
</style>
</head>
<body>
<p>FR: <q lang="fr">Devis</q>.</p>
<p>DE: <q lang="de">Zitate</q>.</p>
<p>EN: <q lang="en">Quotes</q>.</p>
</body>
</html>
```

Результат у браузері (рисунок 5.5):

Рисунок 5.5 – Псевдоклас :lang

Псевдоелементи дозволяють задати стиль елементів не визначених у дереві елементів документа, а також генерувати вміст, якого немає у вихідному коді тексту.

Селектор: Псевдоелемент {Опис правил стилю}

Псевдоелементи не можуть застосовуватися до внутрішніх стилів, лише до зовнішньої таблиці чи стилів рівня документу.

Кожен псевдоелемент може застосовуватися тільки до одного селектору, якщо потрібно встановити відразу декілька псевдоелементів для одного селектора, правила стилю повинні додаватися до них окремо:

.foo:first-letter {color: red}

.foo:first-line {font-style: italic}

Псевдоелементи, їх опис та властивості.

:after - застосовується для вставки призначеного контенту після вмісту елемента. Цей псевдоелемент працює спільно зі стильовим властивістю **content**, яке визначає вміст для вставки.

Приклад:

```
<html>
 <head>
  <meta charset="utf-8">
  <title>Pseudo_element</title>
  <style>
   p.new:after {
    content: "NEW CONTENT";
   }
  </style>
 </head>
 <body>
  <p class="new">11111</p>
```

```
 <p>22222</p>
</body>
</html>
```

Результат у браузері (рисунок 5.6):

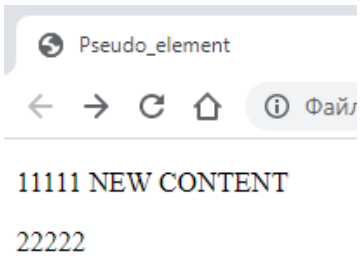


Рисунок 5.6 – Псевдоклас :after

В цьому прикладі до вмісту абзацу з класом new додається додатковий тепкст, який виступає значенням властивості content.

:before - за своєю дією аналогічний псевдоелементу: after, але вставляє контент до вмісту елементу.

Приклад:

```
<head>
<title>Pseudo_element</title>
 <style>
  ul {
   padding-left: 0;
   list-style-type: none; }
  li:before {
   content: "\20aa "; }
 </style>
</head>
<body>
 <ul>
```

```
    <li>Red</li>
    <li>Green</li>
    <li>Blue</li>
    <li>Black</li>
  </ul>
 </body>
```

Результат у браузері (рисунок 5.7):

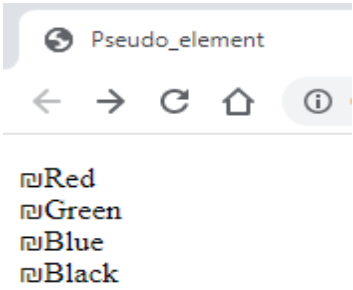


Рисунок 5.7 – Псевдоклас :before

В даному прикладі псевдоелемент :before встановлюється для селектора LI, що визначає елементи списку.

аргументом не обов'язково виступає текст, можуть застосовуватися також символи юнікоду. Додавання бажаних символів відбувається за допомогою значення для властивості content. І :after і :before дають результат тільки для тих елементів, які мають вміст, тому додавання до селектора img або input нічого не дасть.

:first-line - застосовується для блочних елементів. Задає форматування першого рядка тексту. До псевдоелементу :first-line можуть застосовуватися не всі стильові властивості. Допустимо використовувати властивості, що відносяться до шрифту, зміни кольору тексту і фону, а також: clear, line-height, letter-spacing, text-decoration, text-transform, vertical-align і word-spacing.

Приклад:

```
<head>
```

```
<title>first-line</title>
<style>
p:first-line {text-decoration: underline}
</style>
</head>
<body>
<p>:first-line - Applies to block elements. Specifies the formatting of the first line of text. The :first-line pseudo-element can be fully styled.</p>
</body>
```

Результат у браузері (рисунок 5.8):

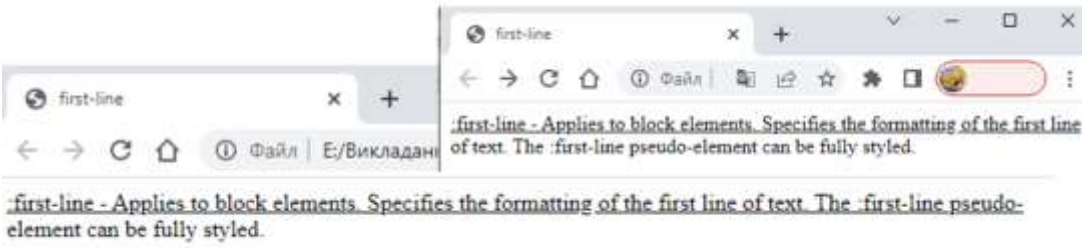


Рисунок 5.8 – Псевдоклас :first-line

:first-letter - дозволяє задати форматування першої літери тексту.

Приклад:

```
<head>
<title>:first-letter </title>
<style>
P:first-letter {
color: red;
font-size: 200%;
border: red solid 1px;
padding: 2px;
margin: 2px; }
```

```
</style>
</head>
<body>
<p>Applies to block elements.</p>
</body>
```

Результат у браузері (рисунок 5.9):

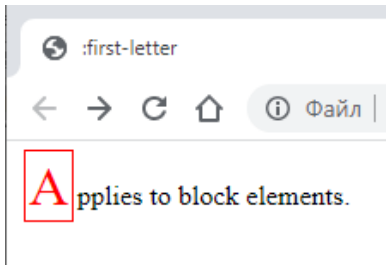


Рисунок 5.9 – Псевдоклас :first-letter

Теги DIV і SPAN

Більшість тегів, які ми розглядали і застосовували до них стилі CSS, мають певне призначення і виконують задану функцію: таблиці, заголовки, параграфи тощо. Але іноді потрібно застосувати стилі до фрагмента вмісту, не виділеного окремим тегом.

Теги … і … використовуються там, де не підходить інший тег. Самі вони не визначають ніякого форматування, але зручні для прив'язки до них стилів.

При цьому, DIV є блочним елементом, SPAN – рядковим.

Основною відмінністю між блочними та рядковими елементами полягає в наступному:

– рядкові елементи йдуть один за одним у рядку тексту;

– блочні - розташовуються один під одним.

До рядкових елементів відносяться такі теги, як <a>, <img>, <input>, <select>, , <sub>, <sup>та ін.

До блочних: , <form>, <section>, <h1> …<h6>, <ol>, <p>, <table>, <ul> та деякі інші.

Для тега вказано стильове правило.

Приклад:

```
<span style="background-color: #eeeeee">Riadlovi elementi</span>
<sub>roztashovuiutsia v riadku</sub>
<img src=“11.jpg“ alt=“v riadku">
<sup>I idut odin za odnim</sup>
```

Результат у браузері (рисунок 5.10):

Рисунок 5.10 – Тег

Для тега < div > вказано стильове правило.

Приклад:

```
<html><head>
<title>Block elements</title>
<style>
h3, div, table {
border: black dotted 1px;
margin: 5px;
padding: 5px;}
</style></head>
<body>
<h3>Header</h3>
<div>Content &lt;div>;
```

```
<div>Insert &lt;div>; 1</div>
<div>Insert &lt;div>; 2</div>
</div>
<table>
<tr><td>Table with one cell</td></tr>
</table>
</body>
</html>
```

Результат у браузері (рисунок 5.11):

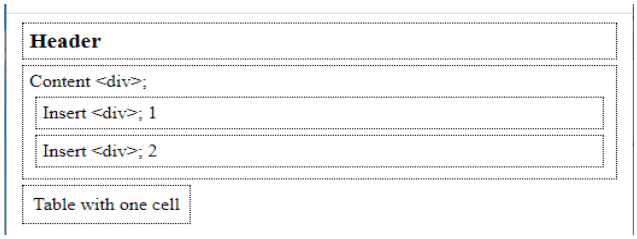


Рисунок 5.11 – Тег

Блочний тег <form> встановлює форму на вебсторінці. Форма призначена для обміну даними між користувачем та сервером. Область застосування форм не обмежена відправкою даних на сервер, за допомогою клієнтських скриптів можна отримати доступ до будь-якого елементу форми, змінювати його та ін.

Рядковий тег <input> є одним з елементів форми і дозволяє створювати різні елементи інтерфейсу та забезпечити взаємодію з користувачем, призначений для створення текстових полів, різних кнопок, перемикачів та прапорців.

Атрибути:

– name - ім’я поля, призначене для того, щоб обробник форми міг його ідентифікувати:

– type - повідомляє браузеру, до якого типу відноситься елемент форми (кнопка, текстове поле, перемикач):

– value - значення елемента.

Блочний тег <button> створює на веб-сторінці кнопки і за своєю дією нагадує результат, що отримується за допомогою тега <input> (з атрибутом type="button | reset | submit").

На відміну від цього тега, <button> пропонує розширені можливості створення кнопок. Наприклад, на подібній кнопці можна розміщувати будь-які елементи HTML, у тому числі зображення. Використовуючи стилі, можна визначити вигляд кнопки шляхом зміни шрифту, кольору фону, розмірів та інших параметрів.

Рядковий тег <select> дозволяє створити елемент інтерфейсу у вигляді списку, а також список з одиночним або множинним вибором.

Поле <textarea> є елементом форми для створення області, в яку можна вводити кілька рядків тексту. На відміну від тега <input> у текстовому полі допустимо робити перенесення рядків, вони зберігаються при надсиланні даних на сервер.

Блочний тег <section> дозволяє групувати елементи веб-документу.

За допомогою CSS можна точно встановити положення елемента на сторінці.

Режимом позиціонування управляє властивість **position -** встановлює, яким чином обчислюється положення елемента у площині екрана. Має кілька режимів:

– **position: static** - режим за замовчуванням, елементи відображаються як завжди – у тому порядку, як вони йдуть в коді (за правилами HTML). Використання властивостей left, top, right, bottom та z-index не призведуть до будь-яких результатів;

– **position: relative** - відносне позиціонування елемента. В даному випадку елемент, розміщується відносно свого поточного положення в потоці документа, тобто візуальних змін не видно. Але потім, користуючись властивостями top, right, bottom і left, можна рухати відносно позиціонований

елемент у потрібному напрямку (початковою точкою відліку буде початкове положення елемента);

– **position: absolute** - задає абсолютне позиціонування елемента. абсолютно позиціонований елемент повністю відокремлюється від загального потоку HTML-документа. Інші елементи сторінки будуть поводитись так, ніби абсолютно позиціонованого елемента не існує. абсолютно позиціонований елемент можна переміщати відносно батьківського (за замовчуванням це вікно браузера) за допомогою властивостей top, right, bottom і left;

– **position: fixed** - фіксоване позиціонування елемента. це схоже на абсолютне позиціонування, але різниця в тому, що на відміну від абсолютно позиціонованого елемента фіксований елемент залишається на місці під час прокручування сторінки;

– **position: inherit** - вказує на спадковість властивостей від свого батьківського елемента.

Приклад:

```
<head>
<title>Position: absolute</title>
<style>
div {
width: 100px;
height: 100px;
border: 3px double black;
padding: 5px;
position: absolute;
}
DIV#first{
background-color: #c0dcc0;
top: 40px;
```

```
left: 40px;
}
div#second{
background-color: #c0c0dc;
top: 80px;
left: 100px;
}
</style>
</head>
<body>
<div id="first">1</div>
<div id="second">2</div></body>
```

Результат у браузері (рисунок 5.12):

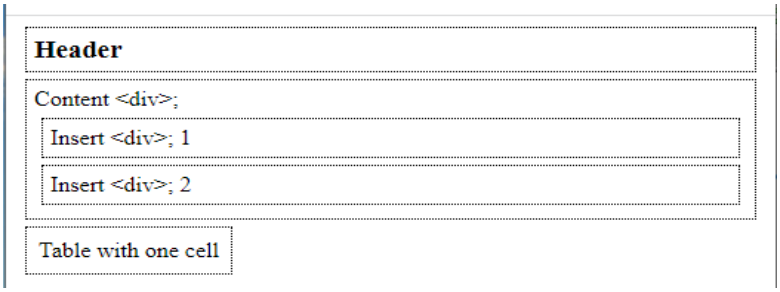


Рисунок 5.12 – Властивість position: absolute

Для блоків задається відступ від верхнього та лівого краю властивостями top і left. Оскільки другий блок оголошений у HTML-коді нижче, він перекриває перший блок на сторінці (рисунок 5.13).

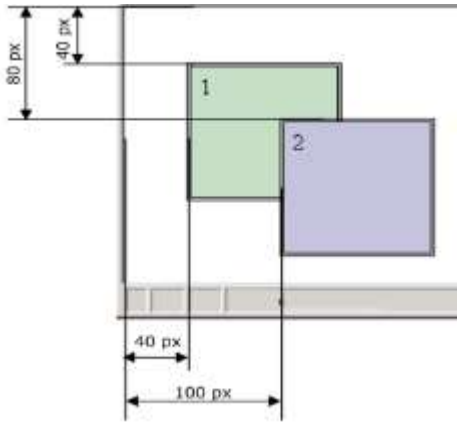


Рисунок 5.13 – Порядок накладання елементів

Для керування порядком накладання елементів одного на інший необхідно використовувати властивість **z-index**. Значенням z-index є додатне або від'ємне число, що задає «висоту», на якій розташований елемент. Елементи з більшим z-index накладаються зверху елементів з меншим z-index.

Щоб в попередньому прикладі перший блок був над другим, необхідно задати z-index для першого блоку, наприклад, рівним 2, а для другого - 1.

Плаваючі елементи

За замовчуванням блочні елементи йдуть строго один під одним. Змінити цей порядок можна зробивши елементи «плаваючими». Для цього служить CSS атрибут **float**. Він задає, по якій стороні буде вирівнюватися елемент: лівої (left) або правої (right).

Плаваючий елемент буде прагнути до лівої чи правої стороні батьківського елемента, а з інших сторін він може обтікати текстом або при цьому потрібно пам'ятати, що властивість float не працює одночасно з позиціонуванням position.

Приклад 1:

```
<html>
<head>
```

```
<title> floating elements</title>
<style>
div#floating{
float: left;
}
</style>
</head>
<body>
The float CSS..<br>
<div id="floating"><img src="11.jpg"
width="120" height="100"></div>
The float<br>
The float<br>
</body>
</html>
```

Результат у браузері (рисунок 5.14):

Рисунок 5.14 – Властивість float

Приклад 2:

```
<html> <head>
<title>floating elements</title>
<style>
div#main {
border: double black 3px;
```

```
width: 150px;
height: 120px;
padding: 5px; }
div.lefty {
border: dashed black 1px;
width: 30px;
height: 30px;
float: left;
margin: 5px;
text-align: center; }
</style> </head>
<body>
<div id="main">
<div class="lefty">1</div>
<div class="lefty">2</div>
<div class="lefty">3</div>
<div class="lefty">4</div>
<div class="lefty">5</div>
</div> </body> </html>
```

Результат у браузері (рисунок 5.15):

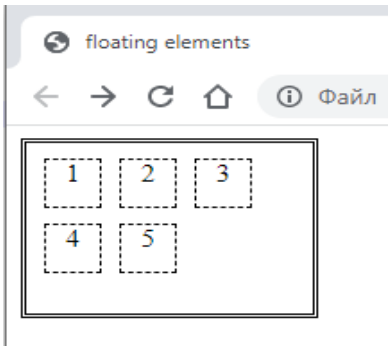


Рисунок 5.15 – Кілька плаваючих блоків

Перший блок вирівнюється по лівому краю батьківського контейнера. Другий блок теж прагне лівого краю, але оскільки місце вже зайняте першим блоком, другий блок стає (обтікає) праворуч від першого. Аналогічно надходить третій блок. Четвертий блок вже не може стати праворуч від третього, тому він поміщається нижче інших і вирівнюється по лівому краю. І п'ятий блок обтікає четвертий праворуч.

Можна одночасно використовувати блоки з вирівнюванням по лівому та правому краю (рисунок 5.16).

```
<div style="float: left">&larr;left</div>
<div style="float: right">right&rarr;</div>
```

Рисунок 5.16 – Властивість float, :left, :right

Ще однією властивістю, пов'язаною з плаваючими елементами, є **clear** - забороняє обтікання елемента з лівої (left), правої (right) або з обох сторін (both). Значення none – обтікання дозволено.

При створенні сайтів плаваючі елементи, властивості float і clear можуть використовуватись для створення каркасу сторінок.

Приклад 3:

```
<head>
<title>Clear</title>
<style>
div {
border: solid black 1px;
width: 75px;
}
```

```
div.floating{
float: left;
}
</style>
</head>
<body>
<div class="floating">Block1</div>
<div class="floating">Block2</div>
<div style="clear:both">A block with the prohibition of flow</div>
<div class="floating">Block3</div>
<div class="floating">Block4</div>
<div class="floating">Block5</div>
</body>
```

Результат у браузері (рисунок 5.17):

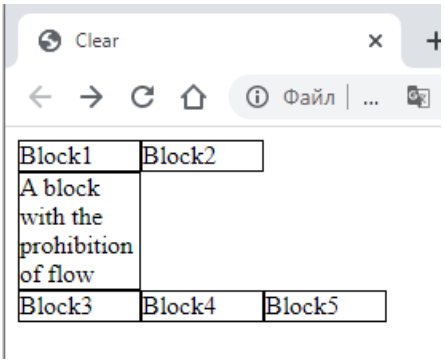


Рисунок 5.17 – Властивість float, clear

Порядок виконання лабораторної роботи

1) ознайомитися з теоретичними відомостями;
2) виконати завдання;
3) оформити звіт;
4) продемонструвати результат на комп'ютері і захистити роботу

ЛАБОРАТОРНИЙ ПРАКТИКУМ №6 JavaScript. Внутрішні, зовнішні скрипти. Змінні. Умови. Цикли. Функції. DOM. BOM. Браузер: документ(document)

Мета: навчитись створювати скрипки мовою JavaScript, підключати їх до html-документу, навчитись використовувати браузерні функції, методи DOM.

Завдання

1) Застосувати функції alert, prompt, confirm. Оформити внутрішніми скриптами.

Написати та викликати власну функцію «Діалог з користувачем», застосувати змінні, умовне розгалуження, цикли. Функцію помістити в зовнішній файл-скрипт. Підключити до html-документу.

Написати та викликати власну функцію виводу інформації про розробника сторінки з параметрами (прізвище, ім’я, посада). Параметру «посада» задати значення за замовчуванням.

Написати функцію порівняння двох рядків, більший вивести на екран, використовуючи alert.

За допомогою об’єкта document змінити фон сторінки на 30 секунд.

2) За допомогою об’єкта location перенаправити браузер на іншу сторінку.

Використати методи пошуку елементів getElementById, querySelectorAll.

Використати наступні властивості DOM-вузла: innerHTML, outerHTML, nodeValue / data, textContent.

Внести зміни в документи/сторінку, використовуючи document.write, document.createElement(tag), document.createTextNode(text) та методи вставки node.append (... nodes or strings), node.prepend (... nodes or strings), node.after (...

nodes or strings), node.replaceWith (... nodes or strings), метод видалення вузлів node.remove ().

3) Завантажити проєкт на віддалений репозиторій (на GitHub).

Теоретичні відомості

JavaScript була створено для того, щоб роботи вебсторінки «живими» [7].

Програми на цій мові називаються скриптами. Вони можуть бути вбудовані у HTML та виконувати автоматичну роботу при завантаженні вебсторінок.

Сьогодні JavaScript може виконуватися не лише у браузері, але і на сервері або на будь-якому іншому пристрої, що має спеціальну програму, що називається “рушієм" JavaScript.

У браузері є власний рушій, який:

– читає (парсить) текст скрипта;

– потім він компілює скрипт в машинну мову.

Після цього машинний код запускається і працює досить швидко. Рушій застосовує оптимізацію на кожному етапі і навіть проглядає компільований скрипт під час його роботи, аналізує дані, що проходять через нього і застосовує оптимізацію до машинного коду, покладаючись на отримані знання. У результаті скрипти працюють дуже швидко.

Скрипти розповсюджуються та виконуються як простий текст.

Програми на JavaScript можуть бути вставлені в будь-яке місце HTML-документа за допомогою тега <script>.

Приклад:

```
<head>
<title></title>
</head>
```

```
<body>
<p>Before script</p>
<script>
alert("Hello!!!");
</script>
<p>After script</p>
</body>
```

Результат у браузері (рисунок 6.1):

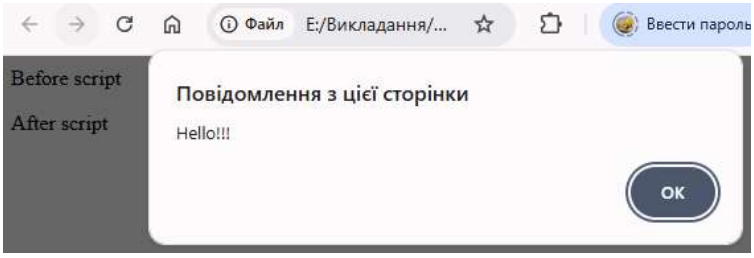


Рисунок 6.1 – Скрипт, вбудований в HTML-документ

Зовнішні скрипти. Якщо у вас багато JavaScript-коду, ви можете помістити його в окремий файл.

Файл скрипта можна підключити до HTML за допомогою атрибута src:

```
<script src="/path/to/script.js"></script>
```

Тут /path/to/script.js - це абсолютний шлях до скрипта від кореня сайту. Також можна вказати відносний шлях від поточної сторінки. Наприклад, src = "script.js" буде означати, що файл "script.js" знаходиться в цій теці.

Можна вказати і повну URL-адресу:

```
<script src="https://path/path/path/to/script.js"></script>
```

Для підключення декількох скриптів використовується кілька ключових слів:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
```

Як правило, тільки найпростіші скрипти поміщаються в HTML. Більш складні виділяються в окремі файли. Користь від окремих файлів в тому, що браузер завантажить скрипт окремо і зможе зберігати його в кеші. Інші сторінки, які підключають той же скрипт, зможуть брати його з кеша замість повторного завантаження з мережі. І таким чином файл буде завантажуватися з сервера тільки один раз.

Якщо атрибут src встановлений, вміст тега script буде ігноруватися. В одному тезі <script> не можна використовувати одночасно атрибут src і код всередині.

Цей не працює:

```
<script src="file.js">
  alert(1); // вміст ігнорується, оскільки є атрибут src
</script>
```

Потрібно вибрати: або зовнішній скрипт <script src = "...">, або звичайний код всередині тега <script>.

Вищенаведений код можна розділити на два скрипта:

```
<script src="file.js"></script>
<script>
  alert(1);
</script>
```

Так як ми використовуємо браузер як демо-середовище, розглянемо кілька браузерних функцій, а саме: alert, prompt і confirm.

alert - вона показує повідомлення і чекає, поки користувач натисне кнопку «ОК» (рисунок 6.2).

```
alert("Hello");
```

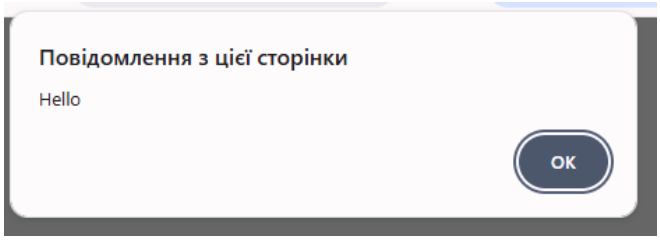


Рисунок 6.2 – Функція alert

prompt - відобразить модальне вікно з текстом, полем для введення тексту і кнопками OK / Скасувати (рисунок 6.3).

Функція prompt приймає два аргументи:

result = prompt(title, [default]);

title - текст для відображення у вікні.

default - необов'язковий другий параметр, який встановлює початкове значення в поле для тексту у вікні.

Користувач може написати текст в поле введення і натиснути OK. Введений текст буде присвоєно змінній result. Користувач також може скасувати введення натисканням на кнопку «Скасувати» або натиснувши на клавішу Esc. В цьому випадку значенням result стане null.

let age = prompt('How many years old?', 100);

alert (`you have ${age} years old!`); // Тобі 100 років!

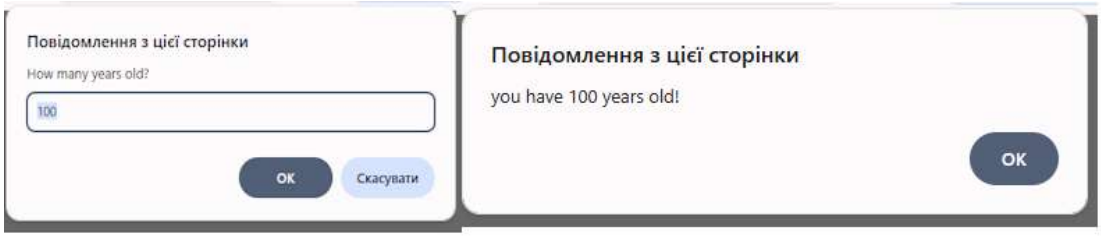


Рисунок 6.3 – Функція prompt

confirm - функція confirm відображає модальне вікно з текстом питання question і двома кнопками: OK / Скасувати (рисунок 6.4).

result = confirm(question);

Результат - true, якщо натиснута кнопка OK. В інших випадках - false.

let ishead = confirm("Are you the head?");

alert(ishead); // true, якщо натиснута OK

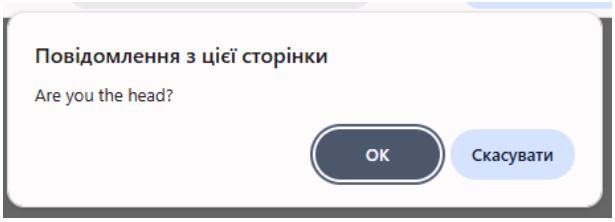


Рисунок 6.4 – Функція confirm

Структура коду представляє собою набір інструкцій - це синтаксичні конструкції і команди, які виконують дії.

Коментарі можуть бути вставлені в будь-якому місці скрипта. Вони не впливають на його виконання, рушій просто ігнорує їх. Однорядкові коментарі починаються з подвійної нахиленої риски //.

Частина рядка після // вважається коментарем. Такий коментар може як займати весь рядок, так і бути вставленим після інструкції.

// Цей коментар займає весь рядок

alert('Привіт');

alert('світ'); // цей коментар іде за інструкцією

JavaScript-застосунком зазвичай потрібно працювати з інформацією. наприклад: чат - інформація може включати користувачів, повідомлення та багато іншого.

Змінні використовуються для зберігання цієї інформації. Змінна - це іменоване сховище для даних.

Для створення змінної в JavaScript використовується ключове слово let.

Наступна інструкція створює (оголошує) змінну з ім'ям «message»:

let message;

Тепер можна записати в неї дані, використовуючи оператор присвоювання «=»:

```
let message;
message = 'Hello'; // зберегти рядок
```

Рядок зберігається в області пам'яті, пов'язаної зі змінною.

Ми можемо отримати до неї доступ, використовуючи ім'я змінної:

```
let message;
message = 'Hello!';
alert(message); // показує вміст змінної
```

Можна поєднати оголошення змінної і запис даних в один рядок:

```
let message = 'Hello!'; // визначаємо змінну і присвоюємо їй значення
```

Також можемо оголосити кілька змінних в одному рядку:

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

Змінну message можна уявити як коробку з назвою "message" і вмстом-значенням "Hello!" всередині:

Ми можемо покласти будь-яке значення в коробку. Ми також можемо змінити його стільки разів, скільки захочемо:

```
let message;
 message = 'Hello!';
 message = 'World!'; // значення змінено
alert(message);
```

При зміні значення старі дані видаляються зі змінної.

Ми також можемо оголосити дві змінні і скопіювати дані з однієї в іншу.

```
let hello = 'Hello world!';
let message; // копіюємо значення 'Hello world' зі змінної hello в змінну message
message = hello;
// тепер дві змінні містять однакові дані
```

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

Змінна може бути оголошена лише один раз. Повторне оголошення тієї ж змінної є помилкою:

```
let message = “yes";
 // повторенння ключового слова 'let' приводить до помилки
let message = “no";
// SyntaxError: 'message' has already been declared
```

В JavaScript є два обмеження, що стосуються іменування змінних:

– ім'я змінної має містити тільки букви, цифри або символи $ і _;
– перший символ не повинен бути цифрою.

Приклади допустимих імен:

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

Якщо ім'я містить кілька слів, зазвичай використовується верблюжа нотація, тобто, слова слідують одне за іншим, де кожне наступне слово починається з великої літери: MyVeryLongName.

Існує список зарезервованих слів, які не можна використовувати в якості імен змінних, тому що вони використовуються самою мовою. Наприклад: let, class, return і function зарезервовані.

Наведений нижче код дає синтаксичну помилку:

```
let let = 5; // не можна назвати змінну "let", помилка!
let return = 5; // також не можна назвати змінну "return", помилка!
```

Константи

Щоб оголосити константну, використовуйте const замість let:

```
const myBirthday = '10.10.1910';
```

Їх не можна змінити. Спроба зробити це призведе до помилки:

```
const myBirthday = '10.10.1910 ';
```

myBirthday = '01.01.2010'; // помилка, константу не можна перезаписати!

Якщо програміст впевнений, що змінна ніколи не буде змінюватися, то можна оголосити її як const.

Область видимості змінних

В JavaScript є три області видимості: глобальна, область видимості функції і блокова. Область видимості змінної - це ділянка вихідного коду програми, в якій змінні і функції видно і їх можна використовувати.

Змінна, оголошена поза функцією або блоком, називається глобальною. Глобальна змінна доступна в будь-якому місці вихідного коду.

Змінна, оголошена всередині функції або блока, називається локальною. Локальна змінна доступна в будь-якому місці всередині тіла функції/блока , в якій/якому вона була оголошена.

Локальна змінна створюється кожного разу заново при виконанні функції та знищується при виході з неї (при завершенні роботи функції).

Локальна змінна має перевагу перед глобальною змінною з тим же ім'ям, це означає, що всередині функції буде використовуватися локальна змінна, а не глобальна:

```
var x = "global";   // Глобальна змінна
 function checkscope() {
  var x = "local";  // Локальна змінна з тим же ім’ям
  document.write(x);    // Використовується локальна змінна, а не глобальна
}
checkscope();          // => "локальна"
```

Змінна, оголошена всередині блоку за допомогою ключового слова let, називається блоковою. Блокова змінна доступна в будь-якому місці всередині блоку, в якому вона була оголошена:

```
let num = 0;
{  let num = 5;
```

```
  console.log(num);     // 5
  { let num = 10;
    console.log(num); } // 10
  console.log(num); }     // 5
console.log(num);       // 0
```

У старих скриптах ви також можете знайти інше ключове слово: var замість let:

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

Ключове слово var - майже те ж саме, що і let. Воно оголошує змінну, але трохи по-іншому. Є відмінності між let і var:

– змінні var не мають блокової області видимості, вони обмежені, як мінімум, тілом функції;

– оголошення (ініціалізація) змінних var відбувається на початку виконання функції (або скрипта для глобальних змінних).

Типи даних - значення в JavaScript завжди відноситься до даних певного типу. Наприклад, це може бути рядок або число. Є вісім основних типів даних в JavaScript.

Змінна в JavaScript може містити будь-які дані. В один момент там може бути рядок, а в іншій - число:

```
// Не буде помилкою
let message = "hello";
message = 123456;
```

Тип «number»

```
let n = 123;
n = 12.345;
```

Числовий тип даних (number) представляє як ціле число, так і числа з плаваючою крапкою.

Існує безліч операцій для чисел, наприклад, множення *, ділення /, додавання +, віднімання - і так далі.

Окрім звичайних чисел, існують так звані «спеціальні числові значення», які відносяться до цього типу даних: Infinity, -Infinity і NaN.

Infinity є математичною нескінченністю ∞.

Ми можемо отримати його в результаті поділу на нуль:

alert(1 / 0); // Infinity

Або поставити його явно:

alert(Infinity); // Infinity

NaN означає обчислювальну помилку. Це результат неправильної або невизначеної математичної операції, наприклад:

alert("не число" / 2); // NaN, таке ділення є помилкою

Будь-яка операція з NaN повертає NaN.

Математичні операції – безпечні. Ми можемо робити що завгодно: ділити на нуль, працювати з нечисловими рядками як з числами і т.д.

Скрипт ніколи не зупиниться з фатальною помилкою. У гіршому випадку ми отримаємо NaN як результат виконання.

Спеціальні числові значення відносяться до типу «число». Звичайно, це не числа в звичному значенні цього слова.

В JavaScript тип «number» не може містити числа більше, ніж (253-1), або менше, ніж - (253 -1) для від'ємних чисел. Для більшості випадків цього достатньо. Але іноді нам потрібні гігантські числа.

BigInt - даний тип дає можливість працювати з цілими числами довільної довжини. Щоб створити значення типу BigInt, необхідно додати n в кінець числового літерала:

const bigInt = 1234567890123456789012345678901234567890n;

Тип рядок (string)

Рядок в JavaScript повинен бути укладений в лапки.

В JavaScript існує три типи лапок (рисунок 6.5):

- подвійні лапки: "Привіт".
- одинарні лапки: 'Привіт'.
- зворотні лапки: `Привіт`.

```
let str = "Привіт";
let str2 = 'Одинарні лапки';
let phrase = `Зоротні лапки дозволяють вбудовувати змінні ${str}`;
```

Подвійні або одинарні лапки є «простими», між ними немає різниці в JavaScript.

Зворотні ж лапки мають розширену функціональність. Вони дозволяють нам вбудовувати вирази в рядок, укладаючи їх в $ {...}.

```
let name = "Ivan";
 // Вставимо змінну
alert(`Hello, ${name}!` );
 // Вставимо вираз
alert(`Result: ${1 + 2}` );
```

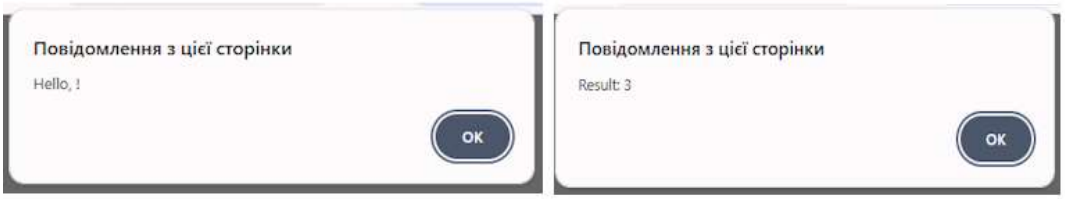


Рисунок 6.5 – Різні лапки

Немає окремого типу даних для одного символу.

Булевий тип (boolean) може приймати тільки два значення: true (істина) і false (брехня).

Такий тип, як правило, використовується для зберігання значень так / ні: true означає «так, правильно», а false означає «ні, не правильно».

```
let nameFieldChecked = true; // так, поле помічено
```

let ageFieldChecked = false; // ні, поле не помічено

Булеві значення також можуть бути результатом порівнянь:

let isGreater = 4 > 1;

alert(isGreater); // true (результат порівняння буде "так")

Спеціальне значення null не відноситься до жодного з типів. Воно формує окремий тип, який містить тільки значення null:

let age = null;

В JavaScript null не є «вказівник на неіснуючий об'єкт» або «нульовим вказівник», як в деяких інших мовах. Це просто спеціальне значення, яке представляє собою «нічого», «порожньо» або «значення невідомо».

Спеціальне значення undefined означає, що значення не було присвоєне. Якщо змінна оголошена, але їй не присвоєно жодного значення, то її значенням буде undefined:

let age;

alert(age); // "undefined"

Тип object (об'єкт) - особливий. В об'єктах зберігають колекції даних або більш складні структури. Об'єкт може бути створений за допомогою фігурних дужок {...} з необов'язковим списком властивостей. Властивість - це пара «ключ: значення», де ключ - це рядок («ім'я властивості»), а значення може бути чим завгодно.

Порожній об'єкт можна створити, використовуючи один з двох варіантів синтаксису:

let user = new object(); // синтаксис «конструктор об’єкта»

let user = {}; // синтаксис «літерал об’єкта»

Зазвичай використовують варіант з фігурними дужками {...}. Таке оголошення називають літералом об'єкта або літеральною нотацією.

let user = {   // об’єкт

  name: "IVAN", // під ключом "name" зберігається значення "IVAN"

```
  age: 30        // під ключом "age" зберігається значення 30
};
```

Для звернення до властивостей використовується запис через крапку:

```
// отримуємо властивості об’єкта:
alert( user.name ); // IVAN
alert( user.age ); // 30
```

Значення може бути будь-якого типу:

```
user.isAdmin = true;
```

Для видалення властивості ми можемо використовувати оператор delete:

```
delete user.age;
```

Для властивостей, імена яких складаються з декількох слів, доступ до значення «через крапку» не працює:

```
user.likes birds = true // це викличе синтаксичну помилку
```

Для таких випадків існує альтернативний спосіб доступу до властивостей через квадратні дужки:

```
let user = {};
 // присвоєння значеннння властивості
user["likes birds"] = true;
 // отримання значеннння властивості
alert(user["likes birds"]); // true
delete user["likes birds"];
```

Існує спеціальний оператор "in" для перевірки існування властивості в об'єкті.

Синтаксис оператора: "key" in object. Зліва від оператора in повинно бути ім'я властивості. Зазвичай це рядок в лапках.

```
let user = { name: "John", age: 30 };
alert( "age" in user ); // true, user.age існує
alert( "blabla" in user ); // false, user.blabla не існує
```

Тип symbol (символ) використовується для створення унікальних ідентифікаторів в об'єктах.

За специфікацією, в якості ключів для властивостей об'єкта можуть використовуватися тільки рядки або символи. Ні числа, ні логічні значення не підходять, дозволені лише ці два типи даних.

let id = Symbol(); // створюємо новий символ - id

let id = Symbol("id"); // створюємо символ id з описом "id"

Символи гарантовано унікальні. Навіть якщо ми створимо безліч символів з однаковим описом, це все одно будуть різні символи. Опис - це просто мітка, яка ні на що не впливає.

Наприклад, ось два символи з однаковим описом - але вони не рівні:

let id1 = Symbol("id");

let id2 = Symbol("id");

alert(id1 == id2); // false

Символи Symbol не перетворюються автоматично в рядки. Більшість типів даних в JavaScript можуть бути неявно перетворені в рядок. Наприклад, функція alert приймає практично будь-яке значення, автоматично перетворює його в рядок, а потім виводить це значення, не повідомляючи про помилку. Символи ж особливі і не перетворюються автоматично.

let id = Symbol("id"); alert(id); // TypeError: Cannot convert a Symbol value to a string

Перетворення символа можна здійснити за допомогою метода .toString()

let id = Symbol("id"); alert(id.toString()); // Symbol(id), працює

або через властивість symbol.description

let id = Symbol("id"); alert(id.description); // id

Символи дозволяють створювати «приховані» властивості об'єктів, до яких не можна ненавмисно звернутися і перезаписати їх з інших частин програми.

Також, якщо той самий код використовується у різних програмах, то в ключі об'єкта краще використовувати значення типу Symbol. Це дає можливість використовувати те саме ім'я ключа в різних частинах коду і уникнути дублювання.

Оператор typeof повертає тип аргументу. Це корисно, коли ми хочемо обробляти значення різних типів по-різному або просто хочемо зробити перевірку.

У нього є дві синтаксичні форми:

– синтаксис оператора: typeof x;

– синтаксис функції: typeof (x).

Виклик typeof x повертає рядок з ім'ям типу:

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof 10n // "bigint"
typeof true // "boolean"
typeof "foo" // "string"
typeof Symbol("id") // "symbol"
typeof Math // "object"  (1)
typeof null // "object"  (2)
typeof alert // "function"  (3)
```

Math - це вбудований об'єкт, який надає математичні операції і константи. Тут він служить прикладом об'єкта.

Результатом виклику typeof null є "object". Це офіційно визнана помилка в typeof. Звичайно, null не є об'єктом. Це спеціальне значення з окремим типом.

Виклик typeof alert повертає "function", тому що alert є функцією. В JavaScript немає спеціального типу «функція». Функції відносяться до об'єктного типу.

Перетворення типів

Найчастіше оператори і функції автоматично приводять передані їм значення до потрібного типу.

Наприклад, alert автоматично перетворює будь-яке значення до рядка. Математичні оператори перетворюють значення до чисел.

Існує 3 найбільш широко використовуваних перетворення: рядкове, чисельне і логічне.

Рядкове - відбувається, коли нам потрібно щось вивести. Може бути викликано за допомогою String (value).

```
let value = true;
alert(typeof value); // boolean
value = String(value); // value це рядок "true"
alert(typeof value); // string
```

Чисельне - відбувається в математичних операціях. Може бути викликано за допомогою Number (value) (рисунок 6.6).

Значення	Стає
undefined	NaN не-число (Not-A-Number)
null	0
true / false	1 / 0
string	Пробільні символи по краях обрізаються. Якщо залишається порожній рядок, то отримуємо 0, інакше з непорожнього рядка «зчитується» число. Якщо помилка - результат NaN.

Рисунок 6.6 – Чисельне перетворення

```
let age = Number("рядок замість числа");
alert(age); // NaN, перетворення не вийшло
alert( Number(" 123 ") ); // 123
alert( Number("123z") ); // NaN (помилка зчитування числа на символі "z")
alert( Number(true) ); // 1
```

alert(Number(false)); // 0

Логічне - відбувається в логічних операціях. Може бути виконано за допомогою Boolean (value) (рисунок 6.7).

Значення	Стає
0, null, undefined, NaN, ""	false
будь-яке інше значення	true

Рисунок 6.7 – Логічне перетворення

Випадки, в яких часто припускаються помилок:

– undefined при чисельному перетворенні стає NaN, не 0;

– "0" і рядки з одним пробілом типу " " при логічному перетворенні завжди true.

alert(Boolean(1)); // true

alert(Boolean(0)); // false

alert(Boolean("Привіт!")); // true

alert(Boolean("")); // false

alert(Boolean("0")); // true

alert(Boolean(" ")); // пробіл, не пустий рядок це true

Базові оператори

Терміни: «унарний», «бінарний», «операнд». Операнд - те, до чого застосовується оператор.

Наприклад, в множенні 5 * 2 є два операнди: лівий операнд дорівнює 5, а правий операнд дорівнює 2.

Унарним називається оператор, який застосовується до одного операнду. Наприклад, оператор унарний мінус "-" змінює знак числа на протилежний:

let x = 1;

```
x = -x;
alert(x); // -1, застосували унарний мінус
```

Бінарним називається оператор, який застосовується до двох операндів. Той же мінус існує і в бінарній формі:

```
let x = 1, y = 3;
alert( y - x ); // 2, бінарний мінус віднімає значення
```

Підтримуються наступні математичні оператори:

- додавання +;
- віднімання -;
- множення *;
- ділення /;
- взяття залишку від ділення %;
- піднесення до ступеню **.

Додавання рядків за допомогою бінарного +:

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Приведення до числа, унарний +:

```
// Не впливає на числа
let x = 1;
alert( +x ); // 1
let y = -2;
alert( +y ); // -2
// перетворює не числа в числа
alert( +true ); // 1
alert( +"" );   // 0
```

Необхідність перетворювати рядки в числа виникає часто. Наприклад, зазвичай значення полів HTML-форми - це рядки. Якщо їх потрібно скласти, то бінарний плюс складе їх як рядки:

```
let apples = "2";
let oranges = "3";
alert( apples + oranges ); // "23", бінарний плюс об’єднує рядки.
```

Використовуємо унарний плюс, щоб перетворити до числа:

```
let apples = "2";
let oranges = "3";
// операнди попередньо перетворено в числа
alert( +apples + +oranges ); // 5
```

Присвоєння

Коли змінній щось присвоюють, наприклад, x = 2 * 2 + 1, то спочатку виконається арифметика, а вже потім відбудеться присвоювання = зі збереженням результату в x.

```
let x = 2 * 2 + 1;
alert( x ); // 5
```

Присвоєння = повертає значення. Виклик x = value записує value в x і повертає його.

Інкремент / декремент

Однією з найбільш частих числових операцій є збільшення або зменшення на одиницю. Для цього існують оператори:

Інкремент ++ збільшує змінну на 1:

```
let counter = 2;
counter++;        // працює як counter = counter + 1
alert( counter ); // 3
```

Декремент -- зменшує змінну на 1:

```
let counter = 2;
```

```
counter--;        // працює як counter = counter - 1
alert( counter ); // 1
```

Іінкремент / декремент можна застосувати тільки до змінної. Спроба використовувати його на значенні, призведе до помилки.

Оператори ++ і -- можуть бути розташовані не тільки після, але і до змінної.

Логічні оператори: || (АБО), && (І), ! (НЕ).

Дані оператори можуть застосовуватися до значень будь-яких типів. Отримані результати також можуть мати різний тип.

Оператор || (АБО):

```
result = a || b;
```

Існує всього чотири можливі логічні комбінації:

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false.
```

Як ми можемо спостерігати, результат операцій завжди дорівнює true, за винятком випадку, коли обидва аргументи false.

Якщо значення не логічного типу, то вони до нього приводиться в цілях обчислень.

При виконанні АБО || з декількома значеннями:

```
result = value1 || value2 || value3;
```

Оператор || виконує наступні дії:

1) обчислює операнди зліва направо;

2) кожен операнд конвертує в логічне значення. Якщо результат true, зупиняється і повертає початкове значення цього операнда;

3) якщо всі операнди є помилковими (false), повертає останній з них.

Значення повертається в початковому вигляді, без перетворення.

Оператор **&&** (І):

result = a && b;

І повертає true, якщо обидва опернди істинні, а інакше - false:

alert(true && true); // true

alert(false && true); // false

alert(true && false); // false

alert(false && false); // false

При декількох підряд операторах І:

result = value1 && value2 && value3;

Оператор && виконує наступні дії:

1) обчислює операнди зліва направо;

2) кожен операнд перетворює в логічне значення. Якщо результат false, зупиняється і повертає початкове значення цього операнда;

3) якщо всі операнди були істинними, повертається останній.

І повертає перше помилкове значення, або останнє, якщо нічого не знайдено

! (НЕ):

result = !value;

Оператор приймає один аргумент і виконує наступні дії:

– спочатку приводить аргумент до логічного типу true / false;

– потім повертає протилежне значення.

alert(!true); // false

alert(!0); // true

Оператори порівняння:

– більше / менше: a> b, a <b.

– більше / менше або дорівнює: a> = b, a <= b.

– дорівнює: a == b. для порівняння використовується подвійний знак рівності ==. Один знак рівності a = b означає присвоювання.

– не дорівнює. В JavaScript записується як a != b.

Результат порівняння має логічний тип.

Всі оператори порівняння повертають значення логічного типу:

true

false

```
alert( 2 > 1 );  // true
alert( 2 == 1 ); // false
alert( 2 != 1 ); // true
```

Результат порівняння можна присвоїти змінній, як і будь-яке значення:

```
let result = 5 > 4; // результат порівняння присвоєно result
alert( result ); // true
```

Порівняння рядків. Щоб визначити, що один рядок більше іншого, JavaScript використовує алфавітний порядок.

```
alert( 'Я' > 'А' ); // true
alert( 'Коти' > 'Кода' ); // true
alert( 'дієвий' > 'ді' ); // true
```

Алгоритм порівняння двох рядків:

– спочатку порівнюються перші символи рядків;

– якщо перший символ першого рядка більше (менше), ніж перший символ другого, то перший рядок більше (менше) другого. Порівняння завершено;

– якщо перші символи рівні, то таким же чином порівнюються вже другі символи рядків;

– порівняння триває, поки не закінчиться один з рядків;

– якщо обидва рядки закінчуються одночасно, то вони рівні. Інакше, більшим вважається більш довгий рядок.

При порівнянні різних типів JavaScript приводить кожне з них до числа:

alert('2' > 1); // true, рядок '2' стане числом 2

alert('01' == 1); // true, рядок '01' стане числом 1

Логічне значення true стає 1, а false - 0.

alert(true == 1); // true

alert(false == 0); // true

Нестрога рівність. Використання звичайного порівняння == може викликати проблеми. Наприклад, воно не відрізняє 0 від false:

alert(0 == false); // true

Та ж проблема з пустим рядком:

alert('' == false); // true

Це відбувається через те, що операнди різних типів перетворюються оператором == до числа. У підсумку, і порожній рядок, і false стають нулем.

Оператор строгої рівності = = = перевіряє рівність без приведення типів. Якщо a і b мають різні типи, то перевірка a === b негайно повертає false без спроби їх перетворення.

alert(0 === false); // false

Ще є оператор строгої нерівності ! ==, аналогічний ! =. Оператор строгої рівності робить код більш очевидним і залишає менше місця для помилок.

Порівняння з **null** і **undefined** - особливе:

При строгій рівності === ці значення різні, так як різні їх типи.

alert(null === undefined); // false

При нестрогій рівності == ці значення дорівнюють один одному і не рівні ніяким іншим значенням - це спеціальне правило мови.

alert(null == undefined); // true

При використанні математичних операторів і операторів порівняння (< > <= >=) значення null / undefined перетворюються до чисел: null стає 0, а undefined – NaN.

Порівняємо null з нулем:

alert(null > 0); // (1) false

alert(null == 0); // (2) false

alert(null >= 0); // (3) true

Чому такий результат: нестрога рівність і порівняння > <> = <= працюють по-різному. Порівняння перетворять null в число, розглядаючи його як 0. Тому вираз (3) null > = 0 істинний, а null> 0 помилковий.

З іншого боку, для нестрогої рівності == значень undefined і null діє особливе правило: ці значення ні до чого не приводяться, вони дорівнюють один одному і не рівні нічому іншому. Тому (2) null == 0 помилковий.

Значення undefined незрівнянно з іншими значеннями:

alert(undefined > 0); // false (1)

alert(undefined < 0); // false (2)

alert(undefined == 0); // false (3)

Порівняння (1) і (2) повертають false, тому що undefined перетворюється в NaN, а NaN - це спеціальне числове значення, яке повертає false при будь-яких порівняннях. Ннестрога рівність (3) повертає false, тому що undefined дорівнює тільки null, undefined і нічому більше.

Умовне розгалуження: if, '?'

Якщо потрібно виконати різні дії в залежності від умов, то можна використовувати інструкцію if і умовний оператор ?.

Інструкція if (...) обчислює умову в дужках і, якщо результат true, то виконує блок коду.

let year = prompt ('В якому році була ти народився?', '');

```
if (year == 2001) {
alert ('Ти молодець' );
alert("!" );
}
```

Інструкція if може містити необов'язковий блок «else» - інакше. Він виконується, коли умова помилкова.

```
if (умова) {
  alert( 'так!' );
} else {
  alert( 'ні!' );
```

Іноді, потрібно перевірити кілька варіантів умови. Для цього використовується блок else if.

Умовиний оператор ?»

```
let result = умова ? значення1 : значення2;
```

Спочатку обчислюється умова: якщо вона істинна, тоді повертається значення1, в іншому випадку - значення2.

```
let accessAllowed = (age > 18) ? true : false;
```

При написанні скриптів часто постає завдання зробити однотипну дію багато разів. Можна скористатись **циклами**.

Цикл while (код з тіла циклу виконується, поки умова condition істинна):

```
while (condition) {
  // код або
  // "тіло циклу"
}
```

Цикл do ... while - перевірку умови можна розмістити під тілом циклу,:

```
do {
  // тіло циклу
```

} while (condition);

Цикл спочатку виконає тіло, а потім перевірить умову condition, і поки її значення дорівнює true, вона буде виконуватися знову і знову.

Цикл for:

```
for (початкова умова; умова; крок) {
 // ... тіло циклу ...
}
```

Цикл виконує alert (i) для і від 0 до (але не включаючи) 3:

```
for (let i = 0; i < 3; i++) { // виведе 0, 1, 2
 alert(i); }
```

– початок i = 0 - виконується один раз при вході в цикл;

– умова i < 3 - перевіряється перед кожною ітерацією циклу. Якщо вона обчислюється в false, цикл зупиниться;

– крок i ++ - виконується після тіла циклу на кожній ітерації перед перевіркою умови;

– тіло alert (i) - виконується знову і знову, поки умова обчислюється в true.

Тобто, початкова умова виконується один раз, а потім кожна ітерація полягає в перевірці умови, після якої виконується тіло і крок.

```
// for (let i = 0; i < 3; i++) alert(i)
// виконати початкову умову
let i = 0;
// якщо умова == true → виконати тіло, виконати крок
if (i < 3) { alert(i); i++ }
// якщо умова == true → виконати тіло, виконати крок
if (i < 3) { alert(i); i++ }
// якщо умова == true → виконати тіло, виконати крок
if (i < 3) { alert(i); i++ } //кінець, бо i == 3
```

Зазвичай цикл завершується при обчисленні умови в false. Але ми можемо вийти з циклу в будь-який момент за допомогою спеціальної директиви **break**.

Директива **continue** - полегшена версія break. При її виконанні цикл не переривається, а переходить до наступної ітерації (якщо умова все ще true).

Код з тіла циклу виконується, поки умова condition істинна.

Наприклад, цикл нижче виводить i, поки i <3:

```
let i = 0;
while (i < 3) { // виводить 0, 1, 2
  alert( i );
  i++; }
```

Конструкція switch замінює собою відразу кілька if. Вона являє собою більш наочний спосіб порівняти вираз відразу з декількома варіантами. Вона має один або більше блоків case і необов'язковий блок default.

```
switch(x) {
  case 'value1':  // if (x === 'value1') ...
    [break]
   case 'value2':  // if (x === 'value2') ...
    [break]
   default:     ...
    [break] }
```

Змінна x перевіряється на строгу рівність першому значенню value1, потім другого value2 і так далі. Якщо відповідність встановлено - switch починає виконуватися до відповідної директиви case і далі, до найближчого break (або до кінця switch). Якщо жоден case не співпав - виконується (якщо є) варіант default.

В switch перевірка на рівність завжди строга. Значення повинні бути одного типу, щоб виконувалася рівність.

Функції

Найчастіше нам треба повторювати одну і ту ж дію в багатьох частинах програми. Щоб не повторювати один і той же код в багатьох місцях, придумані функції.

Для створення функцій ми можемо використовувати оголошення функції.

```
function showMessage() {
  alert('Hello!');
}
```

Спочатку йде ключове слово function, після нього ім'я функції, потім список параметрів в круглих дужках через кому (у вищенаведеному прикладі він порожній), код функції, всередині фігурних дужок.

Функція може бути викликана за її іменем: showMessage ().

```
function showMessage() {
  alert('Hello!');
}
showMessage();
showMessage();
```

Виклик showMessage () виконує код функції. Тут ми побачимо повідомлення двічі.

Змінні, оголошені всередині функції, видно тільки всередині цієї функції:

```
function showMessage() {
  let message = "Hello, JavaScript!"; // локальна змінна
  alert( message ); }
showMessage(); Hello, JavaScript!
alert( message ); // помилка, область видимості змінної  тільки в середині функції.
```

У функції є доступ до зовнішніх змінних:

```
let userName = 'World';
```

```
function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message); }
showMessage(); // Hello, world
```

Функція має повний доступ до зовнішніх змінних і може змінювати їх значення (рисунок 6.8):

```
let userName = 'World';
function showMessage() {
userName = "World 1"; // змінюємо значення зовнішньої змінної
let message = 'Hello, ' + userName;
alert(message); }
alert( userName ); // World перед викликом функції
showMessage();
alert( userName ); // World 1, значення зовнішньої змінної було змінено функцією
```

Рисунок 6.8 – Змінюємо значення змінної

Зовнішня змінна використовується тільки, якщо всередині функції немає такої локальної. Якщо однойменна змінна оголошується всередині функції, тоді вона перекриває зовнішню.

```
let userName = 'World';
function showMessage() {
let userName = "World 1"; // оголошуємо локальну змінну
let message = 'Hello, ' + userName; // World 1
```

alert(message); } // функція буде використовувати свою локальну змінну userName

showMessage();

alert(userName); // World не змінено, функція не чіпала зовнішню змінну

Рисунок 6.9 – Глобальна і локальна змінна

Ми можемо передати всередину функції будь-яку інформацію, використовуючи параметри (які називаються аргументами функції):

function showMessage(from, text) { // аргументи: from, text

alert(from + ': ' + text); }

showMessage('World', 'Hello!'); // World: Hello! (*)

showMessage('World', 'How?'); // World : How? (**)

Якщо параметр не вказано, то його значенням стає undefined.

Наприклад, вищезгадана функція showMessage (from, text) може бути викликана з одним аргументом:

showMessage("World");

Це не призведе до помилки. Такий виклик виведе "World: undefined". У виклику не вказано параметр text, тому передбачається, що text === undefined.

Якщо ми хочемо задати параметру text значення за замовчуванням (рисунок 6.10), ми повинні вказати його після = в оголошенні:

function showMessage(from, text = "текст не додано") { alert(from + ": " + text); }

showMessage("World");

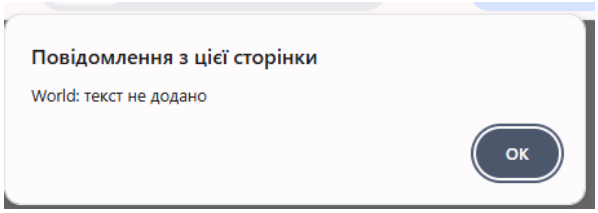


Рисунок 6.10 – Параметри за замовчуванням

Тепер, якщо параметр text не вказано, його значенням буде «текст не додано». В даному випадку «текст не додано» це рядок, але на його місці міг бути і більш складний вираз.

Функція може повернути результат, який буде переданий в код, який її викликав.

```
function sum(a, b) {
return a + b; }
let result = sum(1, 2);
alert( result ); // 3
```

Директива return може перебувати в будь-якому місці тіла функції. Як тільки виконання доходить до цього місця, функція зупиняється, і значення повертається в код, який її викликав (присвоюється змінній result).

Функція JavaScript - це значення особливого типу.

Синтаксис, який ми використовували до цього, називається Function Declaration (Оголошення Функції):

```
function sayHi() {
alert("hello");
}
```

Існує ще один синтаксис створення функцій, який називається Function Expression (Функціональний Вираз):

```
let sayHi = function() {
alert("hello"); };
```

У даному коді функція створюється і явно присвоюється змінній, як будь-яке інше значення. По суті все одно, як ми визначили функцію, це просто значення, збережене в змінній sayHi.

В JavaScript функції - це значення, тому ми і працюємо з ними, як зі значеннями. Функція - не звичайне значення, в тому сенсі, що ми можемо викликати його за допомогою дужок: sayHi (). Але все ж це значення. Тому можемо робити з ним те ж саме, що і з будь-яким іншим значенням.

Відмінності Function Declaration від Function Expression:

– 1 відмінність - синтаксис: Function Declaration: функція оголошується окремою конструкцією «function ...» в основному потоці коду.

```
// Function Declaration
function sum(a, b) {
return a + b; }
// Function Expression
let sum = function (a, b) {
return a + b; };
```

Function Expression: функція, створена всередині іншого виразу або синтаксичної конструкції. В даному випадку функція створюється в правій частині «виразу присвоєння» =

– 2 відмінність - полягає, в тому, коли створюється функція рушієм JavaScript.

Function Expression створюється, коли виконання доходить до неї і потім вже може використовуватися.

Після того, як потік виконання досягне правої частини виразу присвоювання let sum = function ... - з цього моменту, функція вважається створеною і може бути використана (присвоєна змінній, викликана і т.д.).

Function Declaration можна використовувати у всьому скрипті (або блоці коду, якщо функція оголошена в блоці). Тобто, коли рушій JavaScript готується

виконувати скрипт або блок коду, перш за все він шукає в ньому Function Declaration і створює всі такі функції. Можна вважати цей процес стадією ініціалізації. І тільки після того, як всі оголошення Function Declaration будуть оброблені, продовжиться виконання.

В результаті, функції, створені, як Function Declaration, можуть бути викликані раніше своїх визначень.

Функції, оголошені за допомогою Function Expression, створюються тоді, коли виконання доходить до них.

– З відмінність - особливість Function Declaration полягає в їх блоковій області видимості.

У строгому режимі, коли Function Declaration знаходиться в блоці {...}, функція доступна всюди всередині блоку. Але не зовні нього.

У більшості випадків, коли нам потрібно створити функцію, переважно використовується стиль Function Declaration, тому що функція буде видима до свого оголошення в коді. Це дозволяє більш гнучко організовувати код, і покращує його читабельність. Де не можна використати синтаксис Function Declaration, тоді використовується синтаксис Function Expression.

Існує більш короткий синтаксис для створення функцій, який краще, ніж синтаксис Function Expression - функції-стрілки:

let func = (arg1, arg2, ...argN) => expression

Такий код створює функцію func з аргументами arg1..argN і обчислює expression праворуч з їх використанням і повертає результат. Це більш короткий варіант такого запису:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
```

JavaScript у браузерному оточенні використовується для роботи зі сторінкою - отриманням елементів і управлінням ними [8].

Є кореневий об'єкт window, який виступає в 2 ролях (рисунок 6.11):

– глобальним об'єктом для JavaScript-коду;

– вікном браузера і має в своєму розпорядженні методи для управління ним.

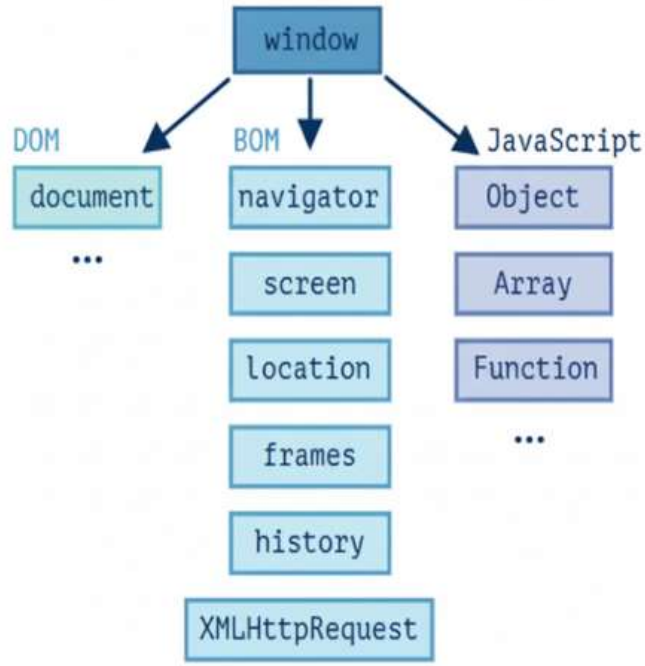


Рисунок 6.11 – Об’єкт Window

Window, як глобальний об'єкт (рисунок 6.12):

```
function sayHi() {
alert("hello"); }
// глобальні функції доступні як методи глобального об’єкта
window.sayHi();
```

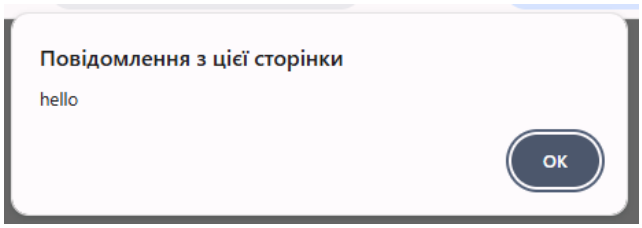


Рисунок 6.12 – Window, як глобальний об'єкт

Window, як об'єкт вікна браузера, щоб дізнатися його висоту (рисунок 6.13):

alert(window.innerHeight);

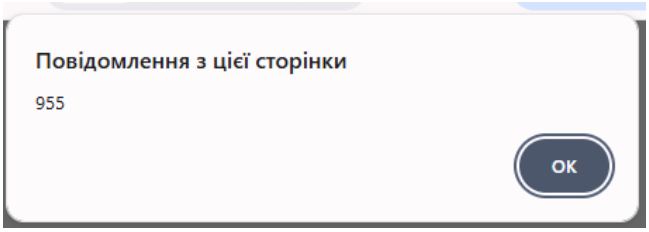


Рисунок 6.13 – Window, як об'єкт вікна браузера

Існує багато властивостей і методів для управління вікном браузера.

DOM (Document Object Model) - об'єктна модель документа, яка представляє весь вміст сторінки у вигляді об'єктів, які можна змінювати.

Об'єкт document - основна «точка входу». З його допомогою ми можемо щось створювати або змінювати на сторінці.

document.body.style.background="red"; // колір фону - червоний

setTimeout(() => document.body.style.background = "", 1000); // через секунду фон зміниться на попередній

Ми використали тільки document.body.style, але можливості по управлінню сторінкою ширші.

DOM - не тільки для браузерів. Специфікація DOM описує структуру документа і надає об'єкти для маніпуляцій зі сторінкою. Різні властивості і методи описані в специфікації: DOM Living Standard на https://dom.spec.whatwg.org.

BOM (Browser Object Model) - це додаткові об'єкти, що надаються браузером (оточенням), щоб працювати з усім, крім документа.

Об'єкт **navigator** дає інформацію про сам браузер і операційну систему. Серед безлічі його властивостей найвідомішими є: navigator.userAgent -

інформація про поточний браузер, і navigator.platform - інформація про платформу (може допомогти в розумінні того, в якій ОС відкритий браузер - Windows / Linux / Mac і так далі).

Об'єкт **location** дозволяє отримати поточний URL і перенаправити браузер за новою адресою (рисунок 6.14):

alert(location.href); // показує поточний URL

if (confirm("go to the site?")) {

location.href="https://html.spec.whatwg.org"}; // перенаправляє браузер на іншу URL }

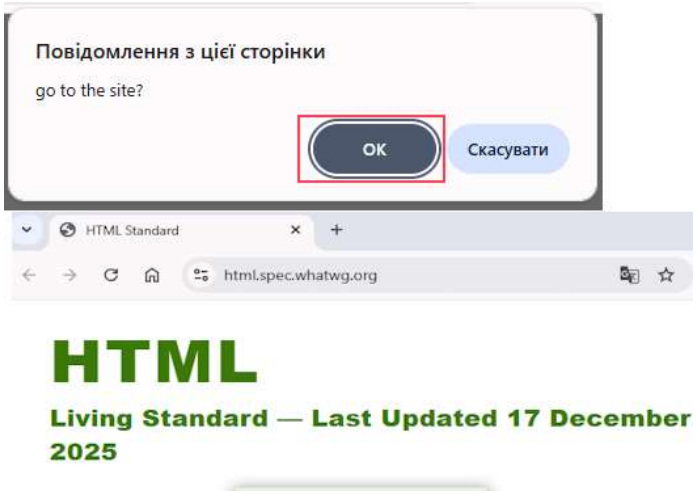


Рисунок 6.14 – Об'єкт location

Функції alert / confirm / prompt теж є частиною BOM: вони не відносяться безпосередньо до сторінки, але є методами об'єкта вікна браузера для комунікації з користувачем.

BOM є частиною загальної специфікації HTML. https://html.spec.whatwg.org - теги, атрибути, об'єкти, методи і специфічні для кожного браузера розширення DOM.

DOM-дерево. Основою HTML-документа є теги. Відповідно до об'єктної модел] документа (DOM), кожен HTML-тег є об'єктом. Вкладені теги є дітьми батьківського елемента. Текст, який знаходиться всередині тега, також є

об'єктом. Всі ці об'єкти доступні за допомогою JavaScript, ми можемо використовувати їх для зміни сторінки [8].

Наприклад, document.body - об'єкт для тега <body>. Якщо запустити наступний код, то <body> стане червоним на 3с:

```
document.body.style.background = “red”;
setTimeout(() => document.body.style.background = “ “, 3000);
```

Приклад DOM (рисунок 6.15):

```
<html>
<head>
  <title>DOM</title>
</head>
<body>
  DOM DOM
</body>
</html>
```

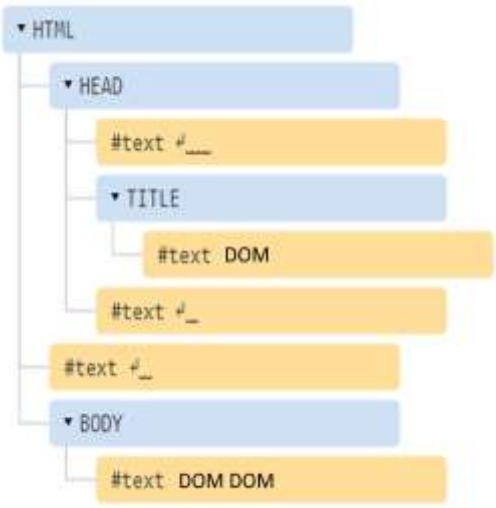


Рисунок 6.15 – DOM

DOM - це представлення HTML-документа у вигляді дерева тегів. Кожен вузол цього дерева - це об'єкт.

Теги є вузлами-елементами. Вони утворюють структуру дерева: <html> - це кореневий вузол, <head> і <body> його дочірні вузли і т.д.

Текст всередині елементів утворює текстові вузли, позначені як #text. Текстовий вузол містить в собі тільки рядок тексту. У нього не може бути нащадків, тобто він знаходиться завжди на самому нижньому рівні.

Спеціальні символи в текстових вузлах:

– перенесення рядка: ↲ (в JavaScript він позначається як \n);

– пробіл. Ці символи враховуються моделлю.

Існує всього два винятки з цього правила:

– з історичних причин пробіли і перенесення рядка перед тегом <head> ігноруються;

– якщо ми записуємо щось після тега </body>, браузер автоматично переміщує цей запис в кінець body, оскільки специфікація HTML вимагає, щоб весь вміст був всередині <body>. Тому після закриваючого тега </ body> не може бути ніяких пробілів.

Якщо браузер стикається з некоректно написаним HTML-кодом, він автоматично коригує його при побудові DOM. Якщо <html> немає в документі - він буде в дереві DOM, браузер його створить.

Навігація по DOM-вузлам

Всі операції з DOM починаються з об'єкта document [8]. Це головна «точка входу» в DOM. З нього ми можемо отримати доступ до будь-якого вузла.

Основні посилання, за якими можна переходити між вузлами DOM (рисунок 6.16):

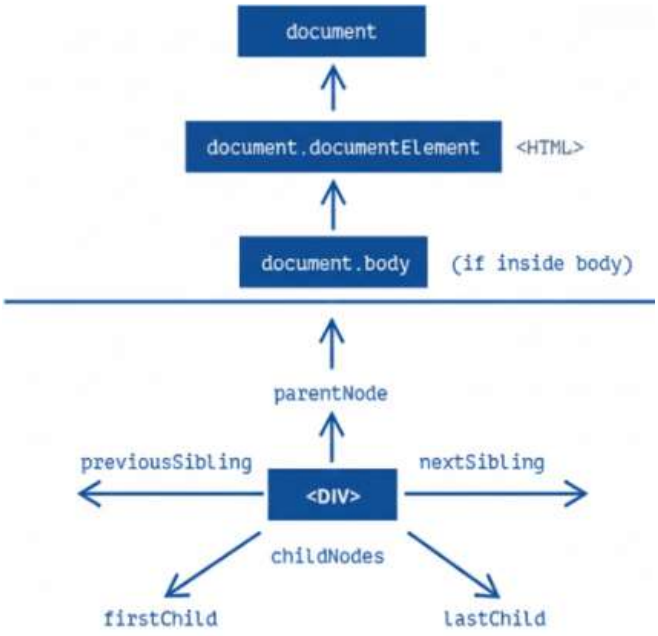


Рисунок 6.16 – Навігація між вузлами

Самі верхні елементи дерева доступні як властивості об'єкта document:

<html> = document.documentelement відповідає тегу <html>.

<body> = document.body вузол тега <body>

<head> = document.head тег <head> доступний як document.head.

Є однин нюанс: document.body може дорівнювати null. Не можна отримати доступ до елементу, якщо його ще не існує в момент виконання скрипта. Зокрема, якщо скрипт знаходиться в <head>, document.body в ньому недоступний, тому що браузер його ще не прочитав. Тому, в прикладі нижче перший alert виведе null:

```
<html>
<head>
 <script>
  alert( "from head: " + document.body ); // null, <body> нема
 </script>
</head>
```

```
<body>
 <script>
  alert( "from body: " + document.body ); // htmlbodyelement є
 </script>
</body>
</html>
```

Дочірні вузли (або діти): childNodes, firstChild, lastChild - елементи, які є безпосередніми дітьми вузла, тобто, елементи, які лежать безпосередньо всередині даного. Наприклад, <head> і <body> є дітьми елемента <html>. Нащадки - всі елементи, які лежать всередині даного, включаючи дітей, їхніх дітей і т.д.

Колекція **childNodes** містить список всіх дітей, включаючи текстові вузли.

Приклад:

```
<html>
<body>
 <div>start</div>
 <ul>
  <li>info</li>
 </ul>
 <div>end</div>
 <script>
  for (let i = 0; i < document.body.childnodes.length; i++) {
   alert( document.body.childnodes[i] );} // text,div,text,ul,…,script
 </script>
html-code
</body>
</html>
```

Результат у браузері (рисунок 6.17):

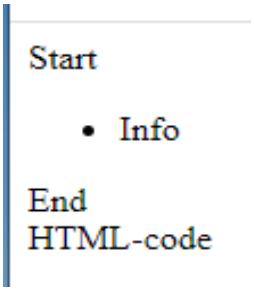


Рисунок 6.17 – Всі дочірні елементи document.body

Самі верхні елементи дерева доступні як властивості об'єкта document:

<html> = document.documentElement

Самий верхній вузол документа: document.documentElement. В DOM він відповідає тегу <html>.

<body> = document.body

Інший часто використовуваний DOM-вузол - вузол тега <body>: document.body.

<head> = document.head

Тег <head> доступний як document.head.

Властивості **firstChild** і **lastChild** забезпечують швидкий доступ до першого і останнього дочірнього елементу.

Якщо у тега є дочірні вузли, умова нижче завжди вірна:

elem.childNodes[0] === elem.firstChild

elem.childNodes[elem.childNodes.length - 1] === elem.lastChild

Для перевірки наявності дочірніх вузлів існує також спеціальна функція elem.hasChildNodes ().

childNodes схожий на масив, а це колекція, тому є два важливих наслідки з цього:

– для перебору колекції ми можемо використовувати уикл for..of:

for (let node of document.body.childNodes) {

alert(node); } // покаже всі вузли з колекції;

– методи масивів не працюватимуть, бо колекція - це не масив:

alert(document.body.childNodes.filter); // undefined (в колекції нема методу filter).

Якщо нам хочеться використовувати саме методи масиву, то ми можемо створити справжній масив з колекції, використовуючи array.from:

alert(array.from(document.body.childNodes).filter); // масив

Сусідні і батьківські вузли.

Сусіди - це вузли, у яких один і той же батько.

В наступному прикладі <head> і <body> сусіди:

<html>

<head>...</head><body>...</body>

</html>

<body> - «наступний» або «правий» сусід <head> також можна сказати, що <head> «попередній» або «лівий» сусід <body>.

Наступний вузол того ж батька (наступний сусід) - у властивості **nextSibling**, а попередній - в **previousSibling**. Батько доступний через **parentNode**.

// батьком <body> є <html>

alert(document.body.parentNode === document.documentElement); // виведе true

// після <head> йде <body>

alert(document.head.nextSibling); // HTMLBodyElement

// перед <body> є <head>

alert(document.body.previousSibling); // HTMLHeadElement

Навігація між елементами

Ці властивості схожі на ті, що стосуються вузлів, тільки в кількох місцях додається слово Element (рисунок 6.18):

– **children** - колекція дітей, які є елементами;

– **firstElementChild**, **lastElementChild** - перший і останній дочірній елемент;

– **previousElementSibling**, **nextElementSibling** - сусіди-елементи;

– **parentElement** - батько-елемент.

Властивість parentElement повертає батька-елемента, а parentNode повертає батька-вузла. Зазвичай ці властивості однакові: вони обидві отримують батька.

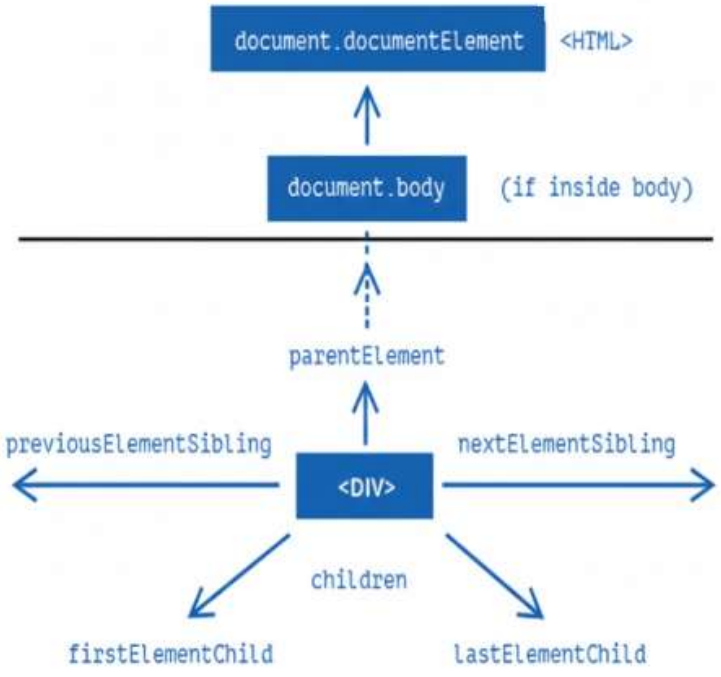


Рисунок 6.18 – Навігація між елементами

За винятком document.documentElement:

alert(document.documentElement.parentNode); // виведе document

alert(document.documentElement.parentElement); // виведе null

Тому що батьком кореневого вузла document.documentElement (<html>) є document. Але document - це не вузол-елемент, так що parentNode поверне його, а parentElement ні.

Якщо замінити childNodes на children, цикл виводить тільки елементи:

```
for (let elem of document.body.children) {
 alert( elem] ); // div, ul, div, script}
```

Пошук: getElement *, querySelector *

Властивості навігації по DOM хороші, коли елементи розташовані поруч. Якщо ні, то для цього в DOM є додаткові методи пошуку.

document.getElementById або просто id

Якщо у елемента є атрибут id, то ми можемо отримати його викликом document.getElementById(id), де б він не знаходився.

Приклад:

```
<html>
<body>
<div id="elem">
  <div id="elem-content">Element</div>
</div>
<script>
  // отримати елемент
  let elem = document.getElementById('elem');
  // зробити його фон червоним
  elem.style.background = 'red';
</script>
</body>
</html>
```

Результат у браузері (рисунок 6.19):

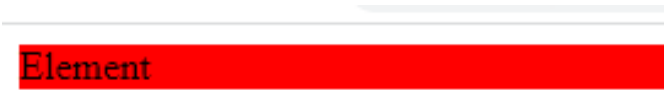


Рисунок 6.19 – Метод getElementById(id)

Метод getElementById можна викликати тільки для об'єкта document. Він здійснює пошук по id по всьому документу.

Метод **querySelectorAll - у**ніверсальний метод пошуку.

elem.querySelectorAll(css), він повертає всі елементи всередині elem, що задовольняють даному CSS-селектору. querySelectorAll повертає статичну колекцію.

Приклад (скрипт отримує всі елементи <li>, які є останніми нащадками в <ul>):

```
<html>
<body>
<ul>
  <li>1</li>
  <li>2</li>
</ul>
<ul>
  <li>3</li>
 <li>4</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');
  for (let elem of elements) {
    alert(elem.innerHTML); // "2", "4"
  }
</script>
</body>
</html>
```

Результат у браузері (рисунок 6.20):

Рисунок 6.20 – Метод querySelectorAll(css)

Псевдокласи в CSS-селекторі, зокрема **:hover** і **:active**, також підтримуються. Наприклад, document.querySelectorAll(':hover') поверне колекцію (в порядку вкладеності: від зовнішнього до внутрішнього) з поточних елементів під курсором миші.

Метод **elem.querySelector(css)** повертає перший елемент, який відповідає цьому CSS-селектору. Результат такий же, як при виклику elem.querySelectorAll (css) [0], але він спочатку знайде всі елементи, а потім візьме перший, в той час як elem.querySelector знайде тільки перший і зупиниться.

Метод **elem.matches (css)** нічого не шукає, а перевіряє, чи задовольняє elem CSS-селектору, і повертає true або false.

Метод elem.closest(css) шукає найближчого пращура, який відповідає CSS-селектору. Сам елемент також включається в пошук.

Пращури елемента - батько, батько батька, його батько і так далі.

Метод closest піднімається вгору від елемента і перевіряє кожного з батьків. Якщо він відповідає селектору, пошук припиняється. Метод повертає або пращура, або null, якщо такий елемент не знайдений.

Приклад:

```
<html>
<body>
<h1>Head</h1>
<div class="contents">
 <ul class="book">
```

```
    <li class="chapter">1</li>
    <li class="chapter">2</li>
  </ul>
</div>
<script>
  let chapter = document.querySelector('.chapter'); // LI
  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV
  alert(chapter.closest('h1')); // null (тому що h1 - не батьківський)
</script>
</body>
</html>
```

Результат у браузері (рисунок 6.21):

Рисунок 6.21 – Метод closest(css)

getElementsBy*

Існують також інші методи пошуку елементів по тегу, класу і так далі.

elem.**getElementsByTagName(tag)** шукає елементи з даним тегом і повертає їх колекцію. Передавши "*" замість тега, можна отримати всіх нащадків.

elem.**getElementsByClassName(className)** повертає елементи, які мають даний CSS-клас.

document.**getElementsByName(name)** повертає елементи з заданим атрибутом name.

Всі методи "getElementsBy*" повертають живу колекцію. Такі колекції завжди відображають поточний стан документа і автоматично оновлюються при його зміні.

Класи DOM-вузлів (рисунок 6.22).

У різних DOM-вузлів можуть бути різні властивості. Наприклад, у вузла, відповідного тегу <a>, є властивості, пов'язані з посиланнями, а у тегу <input> - властивості, пов'язані з полем введення і т.д.

Текстові вузли відрізняються від вузлів-елементів. Але у них є спільні властивості і методи, тому що всі класи DOM-вузлів утворюють єдину ієрархію.

Кожен DOM-вузол належить відповідному вбудованому класу.

Коренем ієрархії є EventTarget, від нього успадковується Node і інші DOM-вузли.

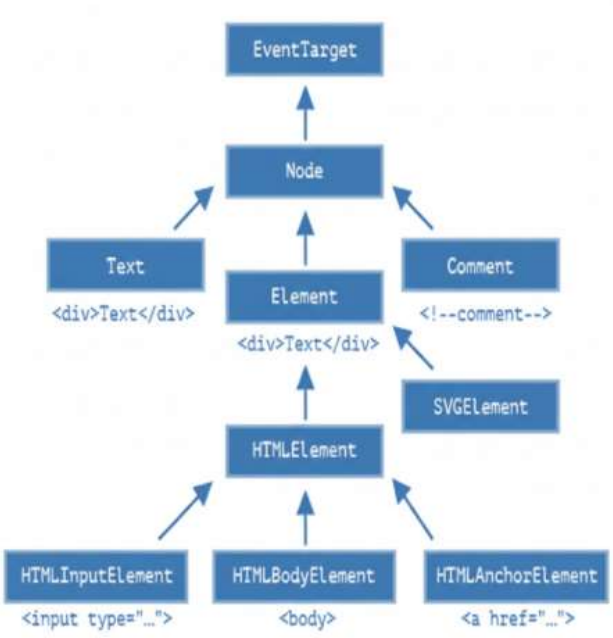


Рисунок 6.22 – Ієрархія класів DOM

EventTarget - це кореневий абстрактний клас. Об'єкти цього класу ніколи не створюються. Він служить основою, завдяки якій всі DOM-вузли підтримують події.

Node - також є «абстрактним» класом, і служить основою для DOM-вузлів. Він забезпечує базову функціональність: parentNode, nextSibling, childNodes і т.д. (Це геттери). Об'єкти класу Node ніколи не створюються. Є класи вузлів, які успадковуються від Node:

– **Text** - для текстових вузлів,

– **Element** - для вузлів-елементів

– **Comment** - для вузлів-коментарів.

Element - це базовий клас для DOM-елементів. Він забезпечує навігацію на рівні елементів: nextElementSibling, children і методи пошуку: getElementsByTagName, querySelector. Браузер підтримує не тільки HTML, але також XML і SVG. Клас Element служить базою для наступних класів: SVGElement, XMLElement і HTMLElement.

HTMLElement - є базовим класом для всіх інших HTML-елементів. Від нього успадковуються конкретні елементи:

HTMLInputElement - клас для тега <input>,

HTMLBodyElement - клас для тега <body>,

HTMLAnchorElement - клас для тега <a>, ... і т.д, кожному тегу відповідає свій клас, який надає певні властивості і методи.

Таким чином, повний набір властивостей і методів даного вузла збирається в результаті успадкування.

Наприклад, DOM-об'єкт для тега <input> належить до класу HTMLInputElement. Він отримує властивості і методи з (в порядку успадкування):

– HTMLInputElement - цей клас надає специфічні для елементів форми властивості;

– HTMLElement - надає загальні для HTML-елементів методи (і геттери / сеттери);

– Element - надає методи елемента;

– Node - надає загальні властивості DOM-вузлів;

– EventTarget - забезпечує підтримку подій, і він успадковується від Object, тому доступні також методи «звичайного об'єкта», такі як hasOwnProperty.

Для того, щоб дізнатися ім'я класу DOM-вузла, зазвичай у об'єкта є властивість **constructor**. Вона посилається на конструктор класу, і у властивості constructor.name міститься його ім'я:

```
alert( document.body.constructor.name ); // HTMLBodyElement
```

Перевірити успадкування можна також за допомогою **instanceof**:

```
alert( document.body instanceof Element ); // true
alert( document.body instanceof Node ); // true
alert( document.body instanceof EventTarget ); // true
```

Отримавши DOM-вузол, ми можемо дізнатися ім'я його тега з властивостей **nodeName** і **tagName**:

```
alert( document.body.nodeName ); // BODY
alert( document.body.tagName ); // BODY
```

Різниця між tagName і nodeName відображена в назвах властивостей (tag, node):

– властивість tagName є тільки у елементів Element.

– властивість nodeName визначено для будь-яких вузлів Node (для елементів воно дорівнює tagName, для інших типів вузлів (текст, коментар) воно містить рядок з типом вузла.

Властивість **innerHTML** повертає текстовий вміст елемента, включаючи всі пробіли та внутрішні теги HTML. також можна змінювати його.

Приклад (показує вміст document.body, а потім замінює його):

```
<body>
  <p>Text</p>
  <div>DIV</div>
  <script>
    alert( document.body.innerHTML ); // current content
    document.body.innerHTML = 'NEW BODY!'; // content replacement
  </script>
</body>
```

Результат у браузері (рисунок 6.23):

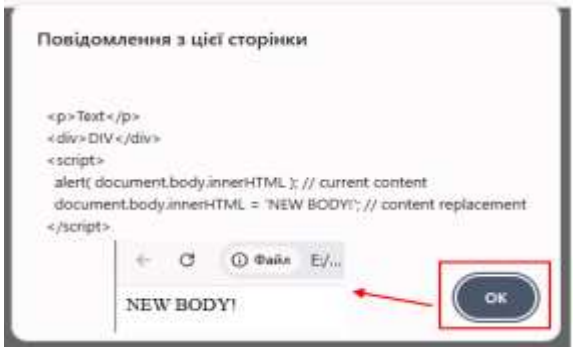


Рисунок 6.23 – Властивість innerHTML

Якщо innerHTML вставляє в документ тег <script> - він стає частиною HTML, але не запускається.

innerHTML + = здійснює перезапис

Можна додати HTML до елементу, використовуючи elem.innerHTML + = "ще html".

chatDiv.innerHTML += " Hello ! ";

chatDiv.innerHTML += "How are you?"; //відбувається не додавання, а перезапис: старий вміст видаляється. На його місце стає нове значення innerHTML (з доданим рядком).

Властивість **outerHTML** містить HTML елемента цілком. Це як innerHTML + сам елемент. Можна замінити елемент на новий HTML. Нове значення HTML передається у вигляді рядка і повністю замінить весь елемент.

Приклад:

```
<div id="elem">Hello <b>World</b></div>
<script>
  alert(elem.outerHTML); // <div id="elem">Hello <b>World</b></div>
</script>
```

Результат у браузері (рисунок 6.24):

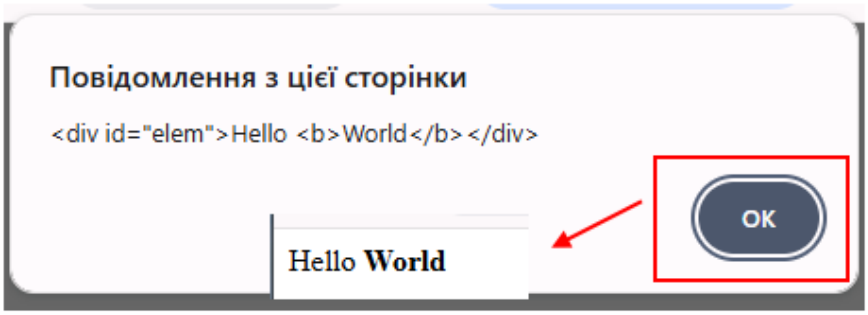


Рисунок 6.24 – Властивість outerHTML

Властивість innerHTML є тільки у вузлів-елементів. У інших типів вузлів, зокрема, у текстових, є свої аналоги: властивості **nodeValue** і **data**.

Приклад:

```
<html>
<body>
  Hello
  <!-- Comments -->
  <script>
    let text = document.body.firstChild;
    alert(text.data); // Hello
    let comment = text.nextSibling;
```

```
    alert(comment.data); // Comments
  </script>
</body>
</html>
```

Результат у браузері (рисунок 6.25):

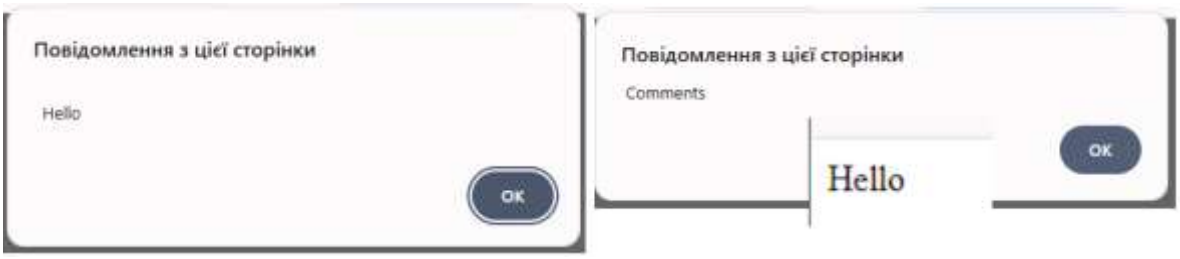


Рисунок 6.25 – Властивість data

Властивість **textContent** надає доступ до тексту всередині елемента за вирахуванням всіх <тегів>.

Приклад:

```
<html>
<body>
<div id="news">
  <h1>Head!</h1>
  <p>Hello!</p>
</div>
<script>
  // Hello!
  alert(news.textContent);
</script>
</body></html>
```

Результат у браузері (рисунок 6.26):

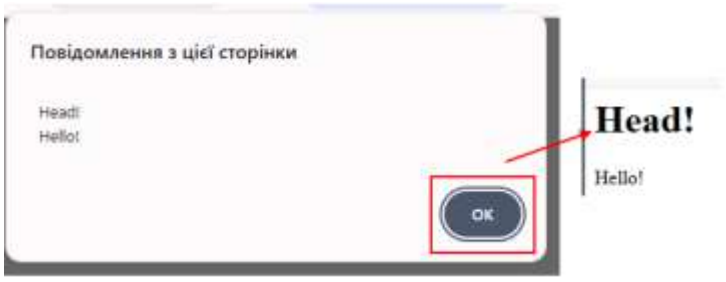


Рисунок 6.26 – Властивість textContent

Атрибут та DOM-властивість **hidden** вказує на те, чи ми бачимо елемент чи ні. Ми можемо використовувати його в HTML або призначати за допомогою JavaScript.

Приклад:

```
<html>
<body>
<div> Both DIV's at the bottom are invisible</div>
<div hidden> With attribute "hidden"</div>
<div id="elem"> With a JavaScript property "hidden"</div>
<script>
  elem.hidden = true;
</script>
</body>
</html>
```

Результат у браузері (рисунок 6.27):

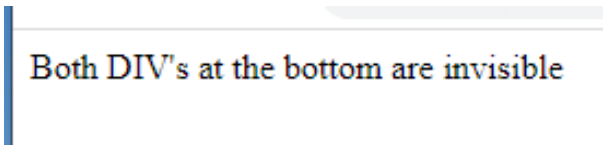


Рисунок 6.27 – Властивість hidden

У DOM-елементів є додаткові властивості, зокрема, залежать від класу:

– value - значення для <input>, <select> і <textarea> (HTMLInputElement, HTMLSelectElement ...).

– href - адреса посилання «href» для <a href="..."> (HTMLAnchorElement).

– id - значення атрибута «id» для всіх елементів (HTMLElement), тощо.

Атрибути і властивості.

Коли браузер завантажує сторінку, він читає HTML і генерує з нього DOM-об'єкти. Для вузлів-елементів більшість стандартних HTML-атрибутів автоматично стають **властивостями DOM-об'єктів**.

<body id = "page"> у DOM-об'єкта буде така властивість body.id = "page".

Але перетворення атрибута у властивість відбувається не один-в-один.

Є вбудовані DOM-властивості, якщо цього замало, можна додати свою властивість. DOM-вузли - це звичайні об'єкти JavaScript, з ними можна працювати, як з об’єктами, можна їх змінювати.

```
document.body.myData = {
  name: 'MyName',
  title: 'MyTitle' };
alert(document.body.myData.title); // MyTitle
```

Можна додати і метод:

```
document.body.sayTagName = function() {
  alert(this.tagName); };
document.body.sayTagName(); // BODY ("this" в цьому методі document.body)
```

Отже, DOM-властивості і методи ведуть себе так само, як і звичайні об'єкти JavaScript:

– їм можна присвоїти будь-яке значення;

– вони чутливі до регістру.

Коли браузер парсить HTML, щоб створити DOM-об'єкти для тегів, він розпізнає стандартні атрибути і створює DOM-властивості для них. Коли у елемента є id або інший стандартний атрибут, створюється відповідна властивість.

Але цього не відбувається, якщо атрибут нестандартний.

```
<body id="test" something="non-standard">
  <script>
    alert(document.body.id); // test
    // нестандартний атрибут не пертворюється у властивість
    alert(document.body.something); // undefined
  </script>
</body>
```

Стандартний атрибут для одного тега може бути нестандартним для іншого. Стандартні атрибути описані в специфікації для відповідного класу елемента.

Таким чином, для нестандартних атрибутів не буде відповідних DOM-властивостей.

Стандартні і нестандартні атрибути доступні за допомогою таких методів:

elem.hasAttribute (name) - перевіряє наявність атрибута.

elem.getAttribute (name) - отримує значення атрибута.

elem.setAttribute (name, value) - встановлює значення атрибута.

elem.removeAttribute (name) - видаляє атрибут.

Ці методи працюють саме з тим, що написано в HTML. Крім цього, отримати всі атрибути елемента можна за допомогою властивості **elem.attributes** : колекція об'єктів, яка належить до вбудованого класу Attr з властивостями name і value.

Синхронізація між атрибутами і властивостями. Коли стандартний атрибут змінюється, відповідна властивість автоматично оновлюється. Це працює і у зворотний бік.

```
<input>
  <script>
    let input = document.querySelector(input);
    input.setAttribute('id', 'id'); // атрибут => властивість
    alert(input.id); // id
    input.id = 'newId';   // властивість => атрибут
    alert(input.getAttribute('id')); // newId
  </script>
```

Але є й винятки, наприклад, input.value синхронізується тільки в один бік - атрибут > значення властивості:

```
<input>
<script>
  let input = document.querySelector('input');
  input.setAttribute('value', 'text'); // атрибут => значення
  alert(input.value); // text
  input.value = 'newValue'; // властивість => атрибут
  alert(input.getAttribute('value')); // text (не оновиться)
</script>
```

DOM-властивості типізовані, не завжди є рядками. Наприклад, властивість input.checked (для чекбоксів) має логічний тип:

```
  <input id="input" type="checkbox" checked> checkbox
  <script>
    alert(input.getAttribute('checked')); // значення атрибута: порожній рядок
    alert(input.checked); // значення властивості: true
  </script>
```

Зміна документа DOM - це можливість створення «живих» сторінок, тобто додавання, заміна, видалення елементів на «льоту», яких нема в HTML-документі.

Приклад (додамо на сторінку повідомлення, яке буде виглядати інакше, ніж alert):

```
<html> <body>
  <style>
   .alert {
     padding: 15px;
     border: 1px solid #d6e9c6;
     border-radius: 4px;
     color: #513c76;
     background-color: #f0e0d8;      }
  </style>
  <div class="alert">
   <strong>Hello!</strong> Message.
  </div>
 </body>  </html>
```

Результат у браузері (рисунок 6.28):

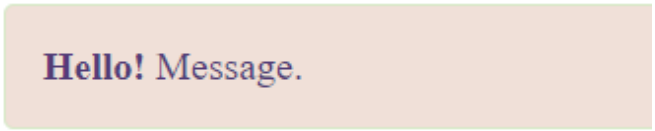


Рисунок 6.28 – Повідомлення

За допомогою JavaScript створимо подібний div.

DOM-вузол можна створити двома методами:

– **document.createElement(tag)** -створює новий елемент із заданим тегом:

let div = document.createElement('div');

– **document.createTextNode(text)** - створює новий текстовий вузол з заданим текстом:

let textNode = document.createTextNode('New node');

У нашому прикладі повідомлення - це div з класом alert і HTML в ньому:

Приклад (створення):

```
<script>
let div = document.createElement('div');
div.className = "alert"; // Властивість className - для вказівки класу css
div.innerHTML = "<strong> Hello! </strong> Message.";
</script>
```

Методи для різних варіантів вставки:

– **node.append (nodes or strings)** - додає вузли або рядки в кінець node;

– **node.prepend nodes or strings)** - вставляє вузли або рядки в початок node;

– **node.before (nodes or strings)** - вставляє вузли або рядки до node;

– **node.after (nodes or strings)** - вставляє вузли або рядки після node;

– **node.replaceWith (nodes or strings)** - замінює node заданими вузлами або рядками.

Якщо ми хочемо вставити HTML саме як html, з усіма тегами та іншим, то є універсальний метод: elem.insertAdjacentHTML (where, html).

Перший параметр - це спеціальне слово, яке вказує, куди по відношенню до elem робити вставку. Значення має бути одним з наступних:

– beforebegin - вставити html безпосередньо перед elem;

– afterbegin - вставити html в початок elem;

– beforeend - вставити html в кінець elem;

– afterend - вставити html безпосередньо після elem.

Другий параметр - це HTML-рядок, який буде вставлений як HTML.

```
<div id="div"></div>
  <script>
    div.insertAdjacentHTML('beforebegin', '<p>111</p>');
    div.insertAdjacentHTML('afterend', '<p>222</p>');
  </script>
```

Призведе до:

```
  <p>111</p>
  <div id="div"></div>
  <p>222</p>
```

Для видалення вузла є метод **node.remove ()**.

Приклад (видалення повідомлення через секундаб наше повідомлення видалялося через секунду):

Приклад (видалення):

```
   <style>
    .alert {
     padding: 15px;
    border: 1px solid #d6e9c6;
    border-radius: 4px;
    color: #513c76;
    background-color: #f0e0d8;       }
    </style>
<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Hello!</strong> Message.";
  document.body.append(div);
  setTimeout(() => div.remove(), 1000); </script>
```

Для клонування вузлів є метод **cloneNode().** Виклик elem.cloneNode (true) створює «глибокий» клон елемента - з усіма атрибутами і дочірніми елементами. Якщо викликати elem.cloneNode (false), тоді клон буде без дочірніх елементів.

Приклад (клонування):

```
<style>
 .alert {
  padding: 15px;
 border: 1px solid #d6e9c6;
 border-radius: 4px;
 color: #513c76;
 background-color: #f0e0d8;       }
 </style>
<div class="alert" id="div">
 <strong>Hello!</strong> Message. </div>
<script>
 let div2 = div.cloneNode(true);
 div2.querySelector('strong').innerHTML = 'Hi!'; // зміна клонованого елемента
 div.after(div2); // клонований елемент після div
</script>
```

Результат у браузері (рисунок 6.29):

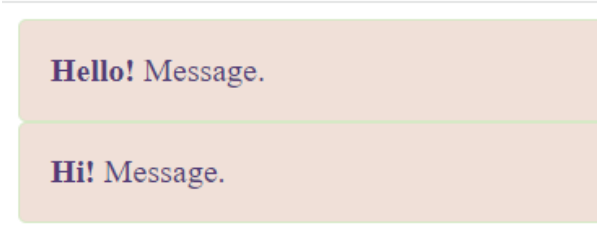


Рисунок 6.29 – Клонування

Є ще один метод додавання вмісту на веб-сторінку: document.write.

Приклад (клонування):

```
<p> Hi </p>
<script>
  document.write('<b>Message</b>');
</script>
<p> Hello </p>
```

Результат у браузері (рисунок 6.30):

Рисунок 6.30 – Метод document.write

Виклик document.write (html) записує html на сторінку тут і зараз. В сучасних скриптах він рідко зустрічається через обмеження: виклик document.write працює тільки під час завантаження сторінки. Якщо викликати його пізніше, то існуючий вміст документа зітреться.

Порядок виконання лабораторної роботи

1) ознайомитися з теоретичними відомостями;
2) виконати завдання;
3) оформити звіт;
4) продемонструвати результат на комп'ютері і захистити роботу

ЛАБОРАТОРНИЙ ПРАКТИКУМ №7 JavaScript. Події. Обробники подій. Спливання. Делегування подій

Мета: навчитись використовувати події та писати обробники подій, застосовувати спливання та делегування подій в реалізації функціоналу.

Завдання

1) Використати одну з подій миші. Написати функцію-обробник. Призначити функцію-обробник події через атрибут і через властивість.

Використати метод addEventListener, призначити одній події різні обробники (написати функції-обробники).

Призначити обробником події об’єкт за допомогою addEventListener, застосувати метод handleEvent, вивести елемент, на якому спрацював обробник, використовуючи event.currentTarget

Видалити об’єкт, використовуючи removeEventListener

2) Створити список або використати існуючий. Реалізувати підсвічування елементів списку при кліку миші. Використати event.target. Обробник onclick застосувати для списку, а не для кожного елементу.

Створити меню (кілька кнопок), додати один обробник для всього меню і атрибути data-* для кожної кнопки, в відповідності з методами, які вони викликають.

Застосувати прийом проєктування «Поведінка» (додавання елементам поведінки behavior за допомогою атрибута data-*.

3) Завантажити проєкт на віддалений репозиторій (на GitHub).

Теоретичні відомості

Подія - це сигнал від браузера про те, що щось відбулось. Всі DOM-вузли подають такі сигнали [8].

Список найчастіше використовуваних DOM-подій:

Події миші:

click - відбувається, коли клікнули на елемент лівою кнопкою миші (на пристроях з сенсорними екранами воно відбувається при торканні);

contextmenu - відбувається, коли клікнули на елемент правою кнопкою миші;

mouseover / mouseout - коли миша наводиться на елемент / полишає елемент;

mousedown / mouseup - коли натиснули / відтиснули кнопку миші на елементі;

mousemove - при русі миші.

Події на елементах управління:

submit - користувач відправив форму <form>;

focus - користувач фокусується на елементі, наприклад натискає на <input>.

Клавіатурні події:

keydown і **keyup** - коли користувач натискає / відтискає клавішу.

Події документа:

DOMContentLoaded - коли HTML завантажений і оброблений, DOM документа повністю побудована і доступна.

CSS events:

transitionend - коли CSS-анімація завершена.

Обробники подій. Події можна призначити обробник, тобто функцію, яка спрацює, як тільки подія відбулася. Саме завдяки обробникам JavaScript-код може реагувати на дії користувача.

Є кілька способів призначити події обробник.

Використання атрибута HTML

Обробник може бути призначений прямо в розмітці, в атрибуті, який називається on <подія>.

Наприклад, щоб призначити обробник події click на елементі input, можна використовувати атрибут onclick, ось так (рисунок 7.1):

<input value="Click me" onclick="alert('Click!')" type="button">

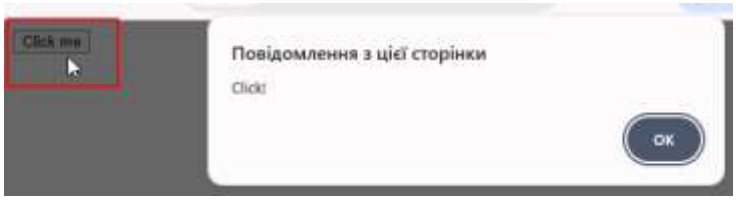


Рисунок 7.1 – Обробник через атрибут

Для вмісту атрибута onclick використовуються одинарні лапки, так як сам атрибут знаходиться в подвійних. Якщо поставити подвійні лапки всередині атрибута: onclick = "alert (" Click! ")", код не буде працювати.

Атрибут HTML-тега - не найзручніше місце для написання великої кількості коду, тому краще створити окрему JavaScript-функцію і викликати її там.

Приклад (при натисканні запускається функція count ():

```
<html>
<body>
  <script>
    function count() {
      for(let i=1; i<=3; i++) {
        alert("Number " + i);        }       }
  </script>
  <input type="button" onclick="count()" value="Counts!">
</body>
</html>
```

Результат у браузері (рисунок 7.2):

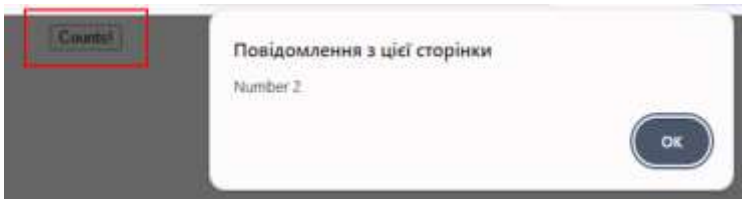


Рисунок 7.2 – Обробник через атрибут (функція)

Атрибут HTML-тега не чутливий до регістру, тому ONCLICK буде працювати так само, як onCLICK і onClick, але, як прийнято атрибути писати в нижньому регістрі: onclick.

Використання властивості DOM-об'єкта

Можна призначати обробник, використовуючи властивість DOM-елемента on <подія>.

Приклад:

```
<html>
<body>
  <input id="elem" type="button" value="Click!">
  <script>
    elem.onclick = function() {
      alert('!!!!!!!!!!!!!!');      };
  </script>
</body>
</html>
```

Результат у браузері (рисунок 7.3):

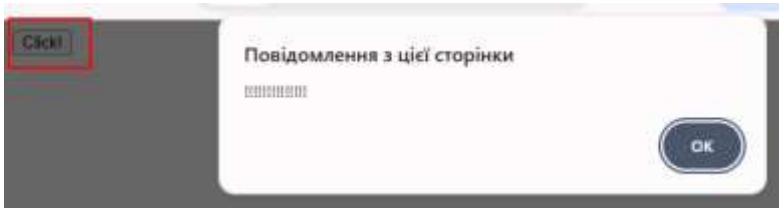


Рисунок 7.3 – Обробник через властивість

Якщо обробник заданий через атрибут, то браузер читає HTML-розмітку, створює нову функцію з вмісту атрибута і записує у властивість.

Обробник завжди зберігається у властивості DOM-об'єкта, а атрибут - лише один із способів його ініціалізації.

Так як у елемента DOM може бути тільки одна властивість з ім'ям onclick, то призначити більше одного обробника так не можна.

Приклад:

```
<html>
<body>
  <input type="button" id="elem" onclick="alert('Old')" value="Click">
  <script>
   elem.onclick = function() { // перезапише існуючий обробник
     alert('New'); // виведеться тільки це
   };
  </script>
</body>
</html>
```

Результат у браузері (рисунок 7.4):

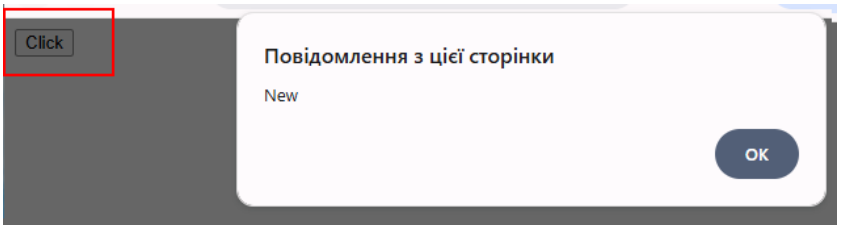


Рисунок 7.4 – Обробник перезапишеться

Обробником можна призначити і вже існуючу функцію:

```
function Hi() {
  alert('Hi!'); }
elem.onclick = Hi;
```

Прибрати обробник можна призначенням elem.onclick = null.

Функція повинна бути присвоєна як Hi, а не Hi ().

button.onclick = Hi;   // правильно

button.onclick = Hi(); // неправильно

Якщо додати дужки, то Hi () - це вже виклик функції, результат якої (рівний undefined, так як функція нічого не повертає) буде присвоєно onclick. Це не буде працювати.

А в розмітці HTML, через атрибут, на відміну від властивості, дужки потрібні:

<input type="button" id="button" onclick="Hi()">

Тобто, при створенні обробника браузером з атрибута, він автоматично створює функцію з тілом зі значення атрибута: Hi().

Розмітка генерує таку властивість:

button.onclick = function() {

 Hi(); // вміст атрибута };

Потрібно використовувати функції, а не рядки. Призначення обробника рядком elem.onclick="alert(1)" також спрацює. Це зроблено з міркувань сумісності, але робити так не рекомендується.

Для обробників не використовується setAttribute, це не буде працювати:

<script>

document.body.setAttribute('onclick', function() { alert(1) }); // при натисканні на body будуть помилки, атрибути завжди рядки і функція стане рядком

</script>

Регістр DOM-властивості має значення. Використовуйте elem.onclick, а не elem.ONCLICK, тому що DOM-властивості чутливі до регістру.

Доступ до елементу через this

Усередині обробника події this посилається на поточний елемент, тобто на той, на якому призначений обробник (рисунок 7.5).

<button onclick="alert(this.innerHTML)">Click</button>

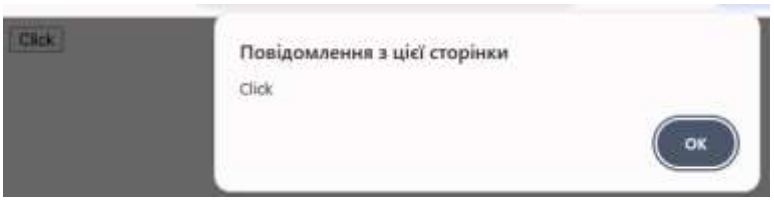


Рисунок 7.5 – Доступ до button через this.innerHTML

Описані способи призначення обробника не дають можливість повісити кілька обробників на одну подію.

Наприклад, одна частина коду при кліку на кнопку, має робити її підсвіченою, а інша - видавати повідомлення.

Для цього потрібно призначити два обробника. Але нова DOM-властивість перезапише попередню:

input.onclick = function() { alert(1); }

// ...

input.onclick = function() { alert(2); } // заміна поп

Є спосіб призначення обробників за допомогою спеціальних методів **addEventListener** і **removeEventListener**.

Додавання обробника:

element.addEventListener(event, handler, [options]);

де:

– event ім'я події, наприклад "click";

– handler посилання на функцію-обробник;

– options Додатковий об'єкт з властивостями (once: якщо true, тоді обробник буде автоматично видалений після виконання; capture: фаза, на якій повинен спрацювати обробник. options може бути false / true, це те ж саме, що {capture: false / true}; passive: якщо true, то вказує, що обробник ніколи не викличе event.preventDefault () – відміна дії браузера).

Видалення обробника:

element.removeEventListener(event, handler, [options]);

Видалення вимагає саме ту ж функцію, якщо функцію обробник не зберіг, її не можна видалити (рисунок 7.6). Немає методу, який дозволяє отримати з елемента обробники подій, призначені через addEventListener.

function handler() { alert(‘WOW!’); }

input.addEventListener("click", handler);

input.removeEventListener("click", handler);

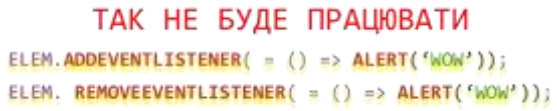


Рисунок 7.6 – Видалення обробника не буде

Метод addEventListener дозволяє додавати кілька обробників на одну подію одного елемента. Можна одночасно призначати обробники і через DOM-властивість і через addEventListener. Але, рекомендується вибрати один спосіб.

Приклад:

```
<html>
<body>
<input id="elem" type="button" value="Click"/>
<script>
  function handler1() {
    alert('Hi!');
  };
  function handler2() {
    alert('Hello!');
  }
  elem.onclick = () => alert("WOW");
```

```
  elem.addEventListener("click", handler1); // Hi!
  elem.addEventListener("click", handler2); // Hello!
</script>
</bdy>
</html>
```

Результат у браузері (рисунок 7.7):

Рисунок 7.7 – Кілька обробників різними способами

Обробники деяких подій можна призначати тільки через addEventListener.

Існують події, які не можна призначити через DOM-властивість, але можна через addEventListener. Наприклад, така подія DOMContentLoaded, яка спрацьовує, коли завершено завантаження і побудова DOM документа.

```
document.onDOMContentLoaded = function() {
  alert("!DOM") }; // не буде працювати
document.addEventListener("DOMContentLoaded", function() {
  alert("DOM");} // буде працювати
```

Об'єкт події

Для обробки події можуть знадобитися деталі того, що сталося. Не просто клік або інша подія, а також, які координати курсора миші, якщо це клавіша, то яка саме і т .д. Коли відбувається подія, браузер створює **об'єкт події**, записує в нього деталі і передає його в якості аргументу функції-обробнику.

Приклад:

```
<html>
<body>
  <input type="button" value="Click me" id="elem">
  <script>
    elem.onclick = function(event) {
      // вивести тип події, елемент та координати кліка
      alert(event.type + " on " + event.currentTarget);
      alert("Coords: " + event.clientX + ":" + event.clientY);
    };
  </script>
</body>
</html>
```

Результат у браузері (рисунок 7.8):

Рисунок 7.8 – Отримання координат миші з об'єкта події

Деякі властивості об'єкта event:

– **event.type** - тип події, в даному випадку "click".

– **event.currentTarget** - елемент, на якому спрацював обробник. Значення - зазвичай таке ж, як і у this, але якщо обробник є функцією-стрілкою або за допомогою bind прив'язаний інший об'єкт в якості this, то ми можемо отримати елемент з event.currentTarget.

– **event.clientX / event.clientY** - координати курсора в момент кліка відносно вікна (для подій миші).

Є також і ряд інших властивостей, в залежності від типу подій.

При призначенні обробника в HTML, теж можна використовувати об'єкт event:

<input type="button" onclick="alert(event.type)" value="Event type">

Це можливо, тому що коли браузер з атрибута створює функцію-обробник, то вона виглядає: **function (event) {alert (event.type)}**. Тобто, її перший аргумент "event", а тіло взято з атрибута.

Можна призначити обробником не тільки функцію, а й об'єкт за допомогою addEventListener. У цьому випадку, коли відбувається подія, викликається метод об'єкта **handleEvent**.

Приклад:

```
<html>
<body>
  <button id="elem">Click me</button>
  <script>
    elem.addEventListener('click', {
      handleEvent(event) {
        alert(event.type + " on " + event.currentTarget);      }    };
  </script>
</body>
</html>
```

Результат у браузері (рисунок 7.9):

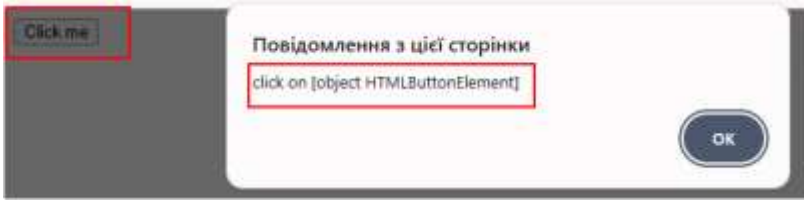


Рисунок 7.9 – Об’єкт, як обробник

Якщо addEventListener отримує об'єкт як обробник, він викликає object.handleEvent (event), коли відбувається подія.

Можна використовувати клас як обробник події.

Приклад:

```
<html>
<body>
 <button id="elem">Click</button>
   <script>
    class Menu {
     handleEvent(event) {
      // mousedown -> onMousedown
      let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);
      this[method](event);
     }
     onMousedown() {
      elem.innerHTML = "MOUSEDOWN";
     }
     onMouseup() {
      elem.innerHTML += "...and MOUSEUP";
     }
    }
    let menu = new Menu();
    elem.addEventListener('mousedown', menu);
    elem.addEventListener('mouseup', menu);
   </script>
</body>
</html>
```

Результат у браузері (рисунок 7.10):

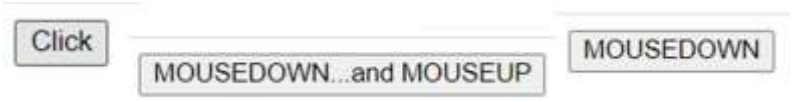


Рисунок 7.10 – Клас, як обробник

Спливання подій

Коли на елементі відбувається подія, обробники спочатку спрацьовують на ньому, потім на його батьку, потім вище і так далі, вгору по ланцюжку пращурів.

Приклад (3 вкладених елемента FORM > DIV > P з обробником на кожному):

```
<html>
<body>
  <style>
   body * {
     margin: 10px;
     border: 1px solid blue;
   }
  </style>
     <form onclick="alert('form')">FORM
   <div onclick="alert('div')">DIV
     <p onclick="alert('p')">P</p>
   </div>
  </form>
</body>
</html>
```

Результат у браузері (рисунок 7.11):

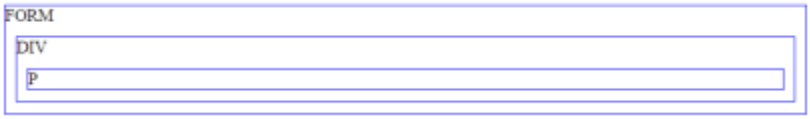


Рисунок 7.11 – Вкладені елементи

Клік по внутрішньому <p> викличе обробник onclick:

– спочатку на самому <p>;

– потім на зовнішньому ;

– потім на зовнішньому <form>;

– і так далі вгору по ланцюжку до самого document.

Тому, якщо клікнути на <p>, то ми побачимо три повідомлення: p > div > form.

Цей процес називається «спливанням», тому що події спливають від внутрішнього елемента вгору через батьків.

Майже всі події спливають. Але, наприклад, подія focus не спливає, і це виняток.

Завжди можна дізнатися, на якому конкретно елементі відбулася подія. Найглибший елемент, який викликає подія, називається цільовим елементом, і він доступний через **event.target**.

Відмінності від this = event.currentTarget:

– **event.target** - це цільової елемент, на якому відбулася подія, в процесі спливання він незмінний;

– **this** - це поточний елемент, до якого дійшло спливання, на ньому зараз виконується обробник.

Наприклад, якщо стоїть тільки один обробник form.onclick, то він перехопить все кліки всередині форми. Де б не був клік всередині - він спливе до елемента <form>, на якому спрацює обробник.

При цьому всередині обробника form.onclick:

– **this = event.currentTarget** завжди буде елемент <form>, так як обробник спрацював на ній;

– **event.target** буде містити посилання на конкретний елемент всередині форми, на якому стався клік.

Приклад:

index.html

```
<html>
 <head>
  <link rel="stylesheet" href="style.css">
 </head>
  <body>
  Клік покаже обидва: і <code>event.target</code>, і <code>this</code> для порівняння:
    <form id="form">FORM
    <div>DIV
     <p>P</p>
    </div>
   </form>
    <script src="script.js"></script>
 </body>
</html>
```

style.css

```
form {
   background-color: green;
   position: relative;
   width: 150px;
   height: 150px;
```

```
    text-align: center;
    cursor: pointer;
  }
div {
    background-color: blue;
    position: absolute;
    top: 25px;
    left: 25px;
    width: 100px;
    height: 100px;
  }
p {
    background-color: red;
    position: absolute;
    top: 25px;
    left: 25px;
    width: 50px;
    height: 50px;
    line-height: 50px;
    margin: 0;
  }

body {
    line-height: 25px;
    font-size: 16px;
  }
```

script.js

```
form.onclick = function(event) {
```

```
    event.target.style.backgroundColor = 'yellow';
    setTimeout(() => {
      alert("target = " + event.target.tagName + ", this=" + this.tagName);
      event.target.style.backgroundColor = ''
    }, 1);
  };
```

Результат у браузері (рисунок 7.12):

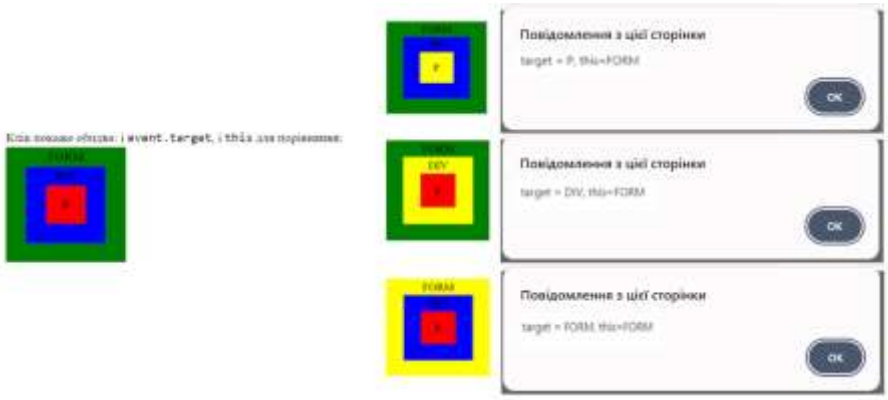


Рисунок 7.12 – Порівняння target і this

Припинення спливання

Спливання йде з «цільового» елемента прямо вгору. За замовчуванням подія буде спливати до елемента <html>, а потім до об'єкта document, а іноді навіть до window, викликаючи всі обробники на своєму шляху.

Але будь-який проміжний обробник може вирішити, що подія повністю оброблена, і зупинити спливання.

Для цього потрібно викликати метод event.stopPropagation() – він припиняє подальшу передачу поточної події:

```
<body onclick="alert(`сюди вспливання не дійде`)">
    <button onclick="event.stopPropagation()">Click</button>
</body>
```

Занурення

Стандарт DOM Events описує 3 фази проходу події:

– фаза занурення (capturing phase) - подія спочатку йде згори донизу;

– фаза мети (target phase) - подія досягло цільового (вихідного) елемента;

– фаза спливання (bubbling stage) - подія починає спливати.

Обробники, додані через on <event> -властивість або через HTML-атрибути, або через **addEventListener (event, handler)** з двома аргументами, нічого не знають про фазу занурення, а працюють тільки на 2-ий і 3-ій фазах.

Щоб зловити подію на стадії занурення, потрібно використовувати третій аргумент **capture**:

elem.addEventListener(..., {capture: true})

elem.addEventListener(..., true)

Існують два варіанти значень опції capture:

– якщо аргумент false (за замовчуванням), то подію буде перехоплено при спливанні;

– якщо аргумент true, то подію буде перехоплено при зануренні.

Делегування подій

Спливання і занурення подій дозволяють реалізувати один з найважливіших прийомів розробки - делегування. Якщо у нас є багато елементів, події на яких потрібно обробляти схожим чином, то замість того, щоб призначати обробник кожній, ми ставимо один обробник на їх спільному батьківському елементі. З нього можна отримати цільовий елемент event.target, зрозуміти на якому саме нащадку відбулася подія і обробити її.

Приклад 1 (підсвічування комірки <td> по кліку):

замість того, щоб призначати обробник onclick для кожної комірки <td> , можна призначити єдиний обробник на елемент <table>. Він буде

використовувати event.target, щоб отримати елемент, на якому відбулася подія, і підсвітити його.

index.html

```
<html>
<body>
  <link type="text/css" rel="stylesheet" href="style.css">
  <table id="my-table">
    <tr>
      <th colspan="3">Head</th>
    </tr>
    <tr>
      <td><strong>A</strong>
        <br>1
        <br>2
      </td>
      <td><strong>B</strong>
        <br>1
        <br>2
      </td>
      <td><strong>C</strong>
        <br>1
        <br>2
      </td>
    </tr>
    <tr>
      <td><strong>D</strong>
        <br>1
        <br>2
```

```
      </td>
      <td><strong>E</strong>
        <br>1
        <br>2
      </td>
      <td><strong>F</strong>
        <br>1
        <br>2
      </td>
    </tr>
</table>
<script src="script.js"></script>
</body>
</html>
```

style.css

```
  table, th, td {
  border: 2px solid blue;
  border-collapse: collapse; }
#my-table th {
  text-align: center;
  font-weight: bold; }
#my-table td {
  width: 100px;
  text-align: center;
  padding-top: 5px;
  padding-bottom: 5px; }
#my-table .light {
```

```
  background: red; }
```

script.js

```
    let selectedTd;
    table.onclick = function(event) {
      let target = event.target;
      while (target != this) {
        if (target.tagName == 'TD') {
          light(target);
          return;
        }
        target = target.parentNode;
      }
    }
    function light(node) {
      if (selectedTd) {
        selectedTd.classList.remove('light');
      }
      selectedTd = node;
      selectedTd.classList.add('light');
}
```

Результат у браузері (рисунок 7.13):

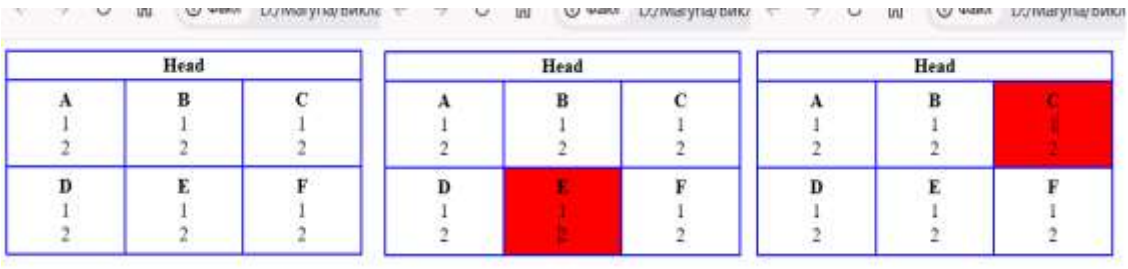


Рисунок 7.13 – Делегуваннння подій

Такому коду немає різниці, скільки клітинок в таблиці. Можемо додавати, видаляти <td> з таблиці динамічно в будь-який час, і підсвічування буде стабільно працювати.

Але, якщо клік може бути не на тезі <td>, а всередині нього, на тегу , то він стане значенням event.target. Всередині обробника table.onclick по event.target потрібно розібратися, був клік всередині <td> чи ні.

Зміни в коді:

```
table.onclick = function(event) {
  let td = event.target.closest('td');
  if (!td) return;
  if (!table.contains(td)) return;
 light(td);
};
```

Метод elem.closest (selector) повертає найближчого батька, відповідного селектору. В даному випадку нам потрібен <td>, що знаходиться вище по дереву від вихідного елемента.

Якщо event.target не міститься всередині елемента <td>, то виклик поверне null, і нічого не станеться.

Якщо таблиці вкладені, event.target може містити елемент <td>, що знаходиться поза поточною таблицею. У таких випадках ми повинні перевірити, чи дійсно це <td> нашої таблиці.

І якщо це так, то підсвічуються його.

Приклад 2 (меню з різними кнопками і об’єкт з відповідними методами):

Можна додати один обробник для всього меню і атрибути data-action для кожної кнопки відповідно до методів, які вони викликають:

```
<button data-action="save">натисніть для збереження</button>
```

Обробник зчитує вміст атрибута і виконує метод.

```
<html>
```

```
<body>
  <div id="menu">
    <button data-action="save">Save</button>
    <button data-action="load">Load</button>
    <button data-action="search">Search</button>
  </div>
  <script>
    class Menu {
      constructor(elem) {
        this._elem = elem;
        elem.onclick = this.onClick.bind(this); // (*)
      }
      save() {
        alert('Save');
      }
      load() {
        alert('Load');
      }
      search() {
        alert('Search');
      }
      onClick(event) {
        let action = event.target.dataset.action;
        if (action) {
          this[action]();
        }
      }
    }
```

```
    new Menu(menu);
  </script>
</body>
</html>
```

Результат у браузері (рисунок 7.14):

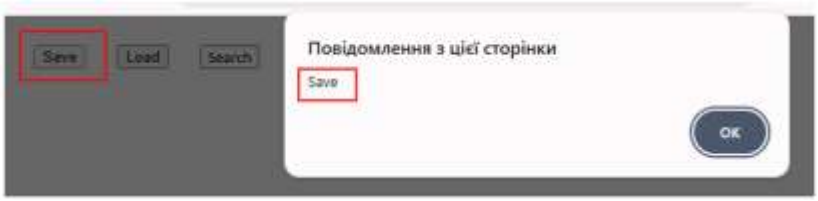


Рисунок 7.14 – Делегування подій

Метод this.onClick в рядку, зазначеному зірочкою (*), прив'язується до контексту поточного об'єкта this. інакше this всередині нього буде посилатися на DOM-елемент (elem), а не на об'єкт Menu, і this [action] буде не тим, що нам потрібно.

Що дає делегування:

– не потрібно писати код, щоб присвоїти обробник кожній кнопці. Досить створити один метод і помістити його в розмітку;

– структура HTML стає по-справжньому гнучкою. Ми можемо додавати / видаляти кнопки в будь-який час;

– атрибути data-action можна використовувати і для стилізації в правилах CSS.

Делегування подій можна використовувати для додавання певної поведінки (**behavior**) елементам декларативно, задаючи обробники за допомогою спеціальних HTML-атрибутів і класів.

Прийом проектування «поведінка» складається з двох частин:

– елементу призначається атрибут, що описує його поведінку.

– за допомогою делегування ставиться обробник на документ, який ловить всі кліки (або інші події) і, якщо елемент має необхідний атрибут, обробляє відповідну подію.

Приклад 3 (Патерн поведінка: «Лічильник»):

```
<html>
<body>
  Лічильник: <input type="button" value="1" data-counter>
  Ще один лічильник: <input type="button" value="2" data-counter>
  <script>
    document.addEventListener('click', function(event) {
      if (event.target.dataset.counter != undefined) {
        event.target.value++;
      }
    });
  </script>
</body>
</html>
```

Результат у браузері (рисунок 7.15):

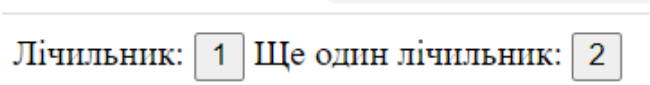


Рисунок 7.15 – Делегування подій

Якщо натиснути на кнопку - значення збільшиться. Елементів з атрибутом data-counter може бути скільки завгодно. Нові можуть додаватися в HTML-код в будь-який момент.

Приклад 4 (Патерн поведінка: «Перемикач»):

```
<html>
<body>
  <button data-toggle-id="subscribe-mail">
```

```
    Показати форму підписки
  </button>
  <form id="subscribe-mail" hidden>
    Ваша пошта: <input type="email">
  </form>
  <script>
    document.addEventListener('click', function(event) {
      let id = event.target.dataset.toggleId;
      if (!id) return;
      let elem = document.getElementById(id);
      elem.hidden = !elem.hidden;
    });
  </script>
</body>
</html>
```

Результат у браузері (рисунок 7.16):

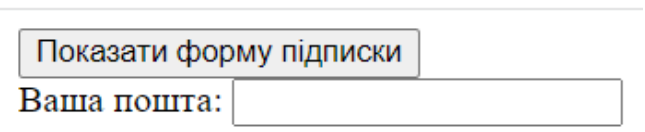


Рисунок 7.16 – Делегування подій

Для того, щоб додати приховування-розкриття будь-якого елементу, навіть не треба знати JavaScript, можна просто написати атрибут data-toggle-id.

Це буває дуже зручно - не потрібно писати JavaScript-код для кожного елемента, який повинен так себе вести. Просто використати поведінку. Обробники на рівні документа зроблять це можливим для елемента в будь-якому місці сторінки. Ми можемо комбінувати кілька варіантів поведінки на одному елементі.

Всі атрибути, назва яких починається з data- валідні, згідно специфікації HTML5, і призначені для прив'язки деякого набору даних користувача до html-елементу. Властивість dataset дозволяє з javascript отримати доступ у режимі читання та запису до атрибутів data-*, встановлених для HTML-елемента.

Порядок виконання лабораторної роботи

1) ознайомитися з теоретичними відомостями;
2) виконати завдання;
3) оформити звіт;
4) продемонструвати результат на комп'ютері і захистити роботу

ЛАБОРАТОРНИЙ ПРАКТИКУМ №8 JavaScript. Події миші

Мета: навчитись використовувати події миші та писати для них обробники подій, реалізувати drag-and-drop за допомогою подій миші.

Завдання

1) Використати події миші: mouseover, mouseout та властивості об’єктів event.target, event.relatedTarget. При русі вказівника миші над елементами, вони повинні змінювати свій стиль.

Реалізувати перетягування елемента (тексту/зображення/файлу) з одного місця в інше, використавши mousedown, mousemove, mouseup.

2) Завантажити проєкт на віддалений репозиторій (на GitHub).

Теоретичні відомості

Події миші бувають не лише через миші, але і емулюються на інших пристроях, зокрема, на мобільних.

Події миші можна розділити на дві категорії: прості і комплексні [8].

Прості події:

– **mousedown / mouseup - к**нопка миші натиснута / відтиснута над елементом;

– **mouseover / mouseout - к**урсор миші з'являється над елементом і йде з нього;

– **mousemove - к**ожен рух миші над елементом генерує цю подію;

– **contextmenu - в**икликається при спробі відкриття контекстного меню, як правило, натисканні правої кнопки миші. Але це не зовсім подія миші, вона може викликатися і спеціальною клавішею клавіатури.

Комплексні події:

– **click** - викликається при mousedown, а потім mouseup над одним і тим же елементом, якщо використовувалася ліва кнопка миші.

– **dblclick - в**икликається подвійним кліком на елементі.

Комплексні події складаються з простих з ними працювати зручно.

Одна дія може викликати кілька подій.

Наприклад, клік мишкою спочатку викликається mousedown, коли кнопка натиснута, потім mouseup і click, коли вона відтиснута.

У разі, коли одна дія ініціює кілька подій, порядок їх виконання фіксований. Тобто обробники подій викликаються в наступному порядку: mousedown → mouseup → click.

У подій, пов’язаних з кліками, є властивість button, яка дозволяє взнати, яку саме кнопку миші клікнули.

Ця властивість не використовується для подій click і contextmenu, оскільки перша відбувається тільки при натисненні лівою кнопкою миші, а друга - правою.

Але якщо ми відстежуємо mousedown і mouseup, то вона нам потрібна, тому що ці події спрацьовують на будь-якій кнопці, і button дозволяє розрізняти натиснуті кнопки.

Можливі значення:

– event. button == 1 - ліва кнопка;

– event. button == 2 - середня кнопка;

– event. button == 3 - права кнопка;

– event. button == 4 – назад;

– event. button == 5 – вперед.

Всі події миші включають в себе інформацію про натиснуті клавіші-модифікатори (shift, alt, ctrl і meta).

Властивості об'єкта події:

– shiftKey: Shift;
– altKey: Alt (або Opt для Mac);
– ctrlKey: Ctrl або Cmd;
– metaKey: Cmd для Mac.

Вони рівні true, якщо під час події була натиснута відповідна клавіша.

Всі події миші мають координати двох видів:

– відносно вікна: clientX і clientY;
– відносно документа: pageX і pageY.

Наприклад, якщо у нас є вікно розміром 500x500, і курсор миші знаходиться в лівому верхньому кутку, то значення clientX і clientY рівні 0. А якщо миша перебуває в центрі вікна, то значення clientX і clientY рівні 250 незалежно від того, в якому місці документа вона знаходиться і до якого місця документ прокручений. У цьому вони схожі на position: fixed.

При наведенні курсора миші на поле введення, побачите clientX / clientY (input знаходиться в iframe, тому координати визначаються відносно iframe) (рисунок 8.1):

<input onmousemove="this.value = event.clientX+':'+event.clientY" value="Point me ">

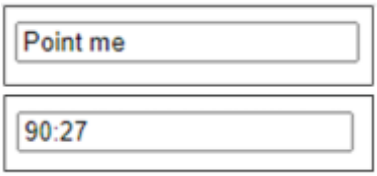


Рисунок 8.1 – Координати clientX і clientY

Координати pageX, pageY відраховуються не від вікна, а від лівого верхнього кута документа.

<input onmousemove="this.value = event.pageX+':'+event.pageY">

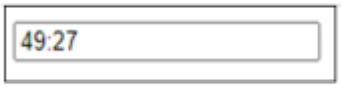


Рисунок 8.2 – Координати pageX і pageY

Подвійний клік миші має специфічну особливість, яка може створювати небажаний результат в деяких інтерфейсах: він виділяє текст.

Наприклад, подвійний клік на текст виділяє його в доповнення до нашого обробник (рисунок 8.3):

Me

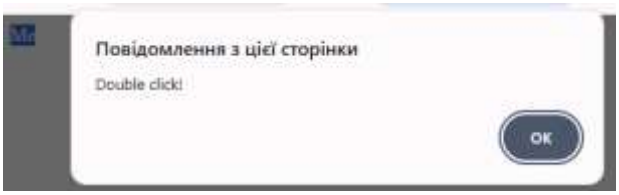


Рисунок 8.3 – Подвійний клік

Якщо затиснути ліву кнопку миші і, не відпускаючи кнопку, провести мишею, то також буде виділення.

Але можна заборонити виділення.

1 спосіб – це використання об’єкта event. Існує метод event.preventDefault().

2 варіант - скасувати дію браузера за замовчуванням, якщо обробник призначено за допомогою on<event> (а не за допомогою addEventListener), return false відмінить типову дію браузера теж.

Наприклад, можна скасувати дію браузера за замовчуванням на події mousedown - це скасує обидва цих виділення - текст не виділяється подвійними кліком і натискання на ньому лівої кнопки не почне виділення (рисунок 8.4):

```
<div ondblclick="alert('Me')" onmousedown="return false">
  "Me", без виділення
</div>
```

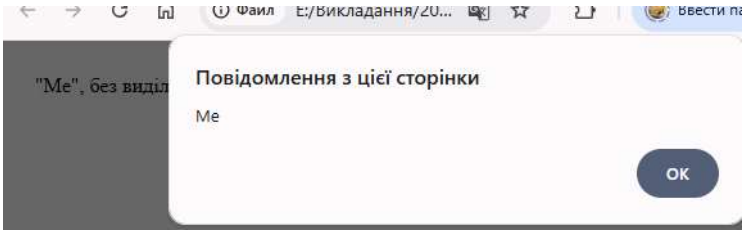


Рисунок 8.4 – Скасування виділення подвійним кліком

Якщо потрібно відключити виділення для захисту вмісту сторінки від копіювання, то можна використовувати подію **oncopy** (рисунок 8.5):

```
<div oncopy="alert('Копіювання заборонено');return false">
  Копіювання заборонено
</div>
```

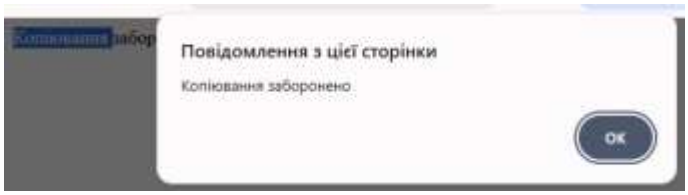


Рисунок 8.4 – Заборона копіювання

Події, що виникають при русі вказівника миші над елементами сторінки mouseover / mouseout.

Подія mouseover відбувається в момент, коли курсор з'являється над елементом, а подія mouseout - в момент, коли курсор йде з елемента. У них є властивість **relatedTarget**. Вона доповнює властивість **target**. Коли миша переходить з одного елемента на інший, то один з них буде target, а інший relatedTarget.

Для події mouseover:

– event.target - це елемент, на який курсор перейшов.

– event.relatedTarget - це елемент, з якого курсор пішов (relatedTarget > target).

Для події mouseout навпаки:

– event.target - це елемент, з якого курсор пішов.

– event.relatedTarget - це елемент, на який курсор перейшов (target > relatedTarget).

Приклад (при русі курсора по цим елементам в текстовому полі відображаються події, які відбуваються):

index.html

```
<html>
<head>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div id="container">
    <div class="child">
    </div>
  </div>
  <textarea id="log">Події!
</textarea>
  <script src="script.js"></script>
</body>
</html>
```

style.css

```
  #container {
  border: 1px solid brown;
  padding: 10px;
  width: 100px;
  margin-bottom: 5px;
  box-sizing: border-box;
```

```
}
#log {
  height: 120px;
  width: 350px;
  display: block;
  box-sizing: border-box;
}
.child {
  display: inline-block;
  width: 70px;
  height: 70px;
  border-radius: 50%;
  margin-right: 20px;
  background: #4405f0;
  border: 5px solid #fd0000;
  position: relative;
}
```

script.js

```
  container.onmouseover = container.onmouseout = handler;
function handler(event) {
  function str(el) {
    if (!el) return "null"
    return el.className || el.tagName;
  }
  log.value += event.type + ':  ' +
    'target=' + str(event.target) +
    ',  relatedTarget=' + str(event.relatedTarget) + "\n";
```

```
  log.scrollTop = log.scrollHeight;
  if (event.type == 'mouseover') {
    event.target.style.background = 'yellow'
  }
  if (event.type == 'mouseout') {
    event.target.style.background = ''
  }
}
```

Результат у браузері (рисунок 8.5):

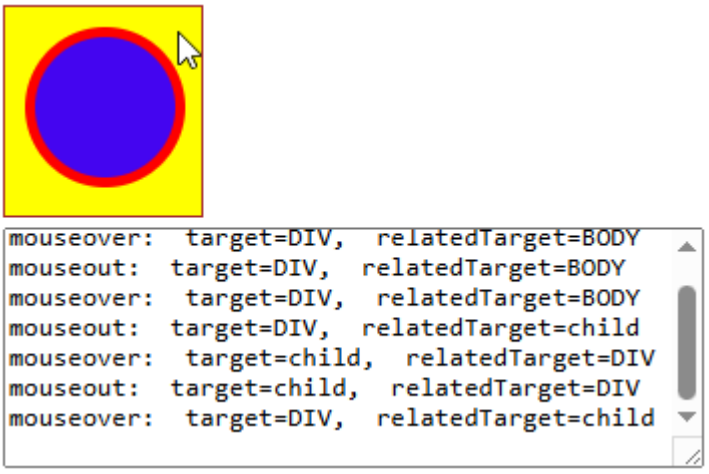


Рисунок 8.5 – Властивості target, relatedTarget

Властивість relatedTarget може бути null. Це означає, що курсор перейшов не з якогось елементу, а з-за меж вікна браузера. Або ж, навпаки, пішов за межі вікна.

Подія mousemove відбувається під час руху миші. але, це не означає, що подія mousemove генерується при проходженні кожного пікселя.

Браузер періодично перевіряє позицію курсора і, помітивши зміни, генерує подію mousemove, тобто, якщо користувач рухає мишкою дуже швидко, то деякі DOM-елементи можуть бути пропущені. Це добре, тому що, якщо проміжних елементів багато, навряд чи потрібно обробляти вхід і вихід для кожного. Також,

можливо, вказівник зайде в середину сторінки з-за меж вікна браузера. У цьому випадку значення relatedTarget буде null, так як курсор прийшов з нізвідки.

Важливою особливістю події mouseout э те, що вона генерується в тому числі, коли курсор переходить з батьківського елемента на його нащадка, який знаходиться глибше, але ми отримаємо mouseout на батьківському елементі (курсор миші може бути тільки над одним елементом в будь-який момент часу - над самим глибоко вкладеним і верхнім за z-index).

Подія mouseover, яка відбувається на нащадку, спливає. Тому, якщо батьківський елемент має обробник mouseover, то він спрацює.

Події mouseenter / mouseleave схожі на mouseover / mouseout. Вони теж генеруються, коли курсор миші переходить на елемент або залишає його. Але є відмінності:

– переходи всередині елемента, на його нащадки і з них, не враховуються;

– події mouseenter / mouseleave не спливають, тобто їх не можна делегувати.

Коли курсор з'являється над елементом - генерується mouseenter, і не має значення, курсор на самому елементі або на його нащадку.

Подія mouseleave відбувається, коли курсор залишає елемент.

Drag'n'Drop з подіями миші

Захоплення елемента мишкою і його перенесення візуально спростять що завгодно: від копіювання та переміщення документів (як в файлових менеджерах), до оформлення замовлення (покласти в кошик).

У стандарті HTML5 є розділ про Drag and Drop - і там є спеціальні події саме для Drag'n'Drop перенесення, такі як dragstart, dragend і так далі.

Вони цікаві тим, що дозволяють легко вирішувати прості завдання. Наприклад, можна перетягнути файл в браузер, так що JS отримає доступ до його вмісту.

Але у них є і обмеження. Наприклад, не можна організувати перенесення тільки по горизонталі або тільки по вертикалі. Також не можна обмежити перенесення всередині заданої зони. Таокж, мобільні пристрої погано їх підтримують.

Алгоритм Drag'n'Drop за допомогою подій миші:

– при mousedown – підготувати елемент до переміщення, якщо необхідно, тобто виконати якісь дії;

– потім при mousemove перетягнути елемент на нові координати шляхом зміни left / top і position: absolute.

– при mouseup - зупинити перенесення елемента і зробити всі дії, пов'язані з закінченням Drag'n'Drop.

Приклад (перетягування смайла):

index.html

```
<html>
<body>
  <p> Перетягніть смайл </p>
<img src="E:\Викладання\2025\ОКР\grag_drop\Smile.png">
</body>
</html>
```

script.js

```
smile.onmousedown = function(event) { // відстежити натискання
  // підготувати до переміщення: розмістити поверх решти вмісту
  smile.style.position = 'absolute';
  smile.style.zIndex = 1000;
```

```
  // перемістимо в body, щоб смайл був не всередині position:relative
  document.body.append(smile);
   // центрування
  function moveAt(pageX, pageY) {
    smile.style.left = pageX - smile.offsetWidth / 2 + 'px';
    smile.style.top = pageY - smile.offsetHeight / 2 + 'px';
  }
 // встановимо абсолютно спозиційований смайл під курсор
    moveAt(event.pageX, event.pageY);
function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }
  // переміщати по екрану
  document.addEventListener('mousemove', onMouseMove);
    // покласти смайл, видалити непотрібні обробники подій
  smile.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    smile.onmouseup = null;
  };
};
```

Результат у браузері (рисунок 8.6):

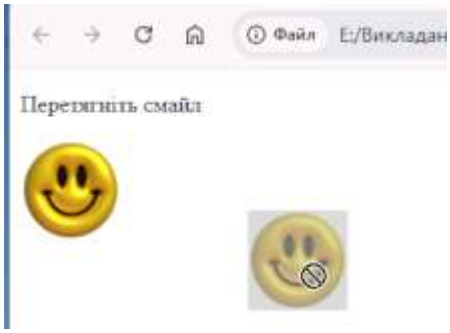


Рисунок 8.6 – Drag'n'Drop

Видно, що при перенесенні смайл роздвоюється і переноситься його клон, тому, що браузер має власний Drag'n'Drop, який автоматично запускається і вступає в конфлікт з нашим. Це відбувається саме для зображень і деяких інших елементів.

Його потрібно відключити:

```
smile.ondragstart = function() {
  return false;
};
```

Тепер такого ефекту не буде.

Також, подія mousemove відстежується на document, а не на smile. З першого погляду здається, що миша завжди над смайлом і обробник mousemove можна повісити на сам смайл, а не на документ.

Подія mousemove виникає хоч і часто, але не для кожного пікселя. Тому через швидкий рух курсор може зістрибнути з смайла і опинитися в середині документа (або за межами вікна).

Ось чому ми повинні відслідковувати mousemove на всьому document, щоб зловити його.

В приуладі смайл позиціонується так, що його центр є під курсором миші:

```
smile.style.left = pageX - smile.offsetWidth / 2 + 'px';
smile.style.top = pageY - smile.offsetHeight / 2 + 'px';
```

Непогано, але є побічні ефекти. Ми, для початку переносу, можемо натиснути мишею на будь-якому місці смайла. Якщо смайл взяти за самий край, то на початку перенесення він різко стрибає, центруючись під курсором миші.

Було б краще, якби початковий зсув курсору щодо елемента зберігався. Де захопили, за ту частину елемента і переносимо.

Оновлений алгоритм:

– коли натискаємо на смайл (mousedown) - запам'ятаємо відстань від курсора до лівого верхнього кута кулі в змінних shiftX / shiftY. Далі будемо утримувати цю відстань у разі перетягування;

Щоб отримати цей зсув, ми можемо відняти координати:

let shiftX = event.clientX - smile.getBoundingClientRect().left;

let shiftY = event.clientY - smile.getBoundingClientRect().top;

– при перенесенні смайла ми позиціонуємо його з тим же зрушенням щодо покажчика миші, ось так:

smile.style.left = event.pageX - shiftX + 'px';

smile.style.top = event.pageY - shiftY + 'px';

Смайл можна покасти де завгодно в межах вікна. Але, ми зазвичай беремо один елемент і перетягуємо в інший.

Тобто, ми беремо draggable елемент і поміщаємо його в інший елемент droppable.

Нам потрібно знати:

– куди користувач поклав елемент в кінці перенесення, щоб обробити його закінчення;

– і над якою потенційною мішенню (елемент, куди можна покласти) він знаходиться в процесі перенесення, щоб підсвітити її.

Можна встановити обробники подій mouseover / mouseup на елемент потенційну мішень перенесення. Але це не спрацює, тому що при переміщенні елемент, який переміщається, завжди знаходиться над іншими елементами. А події миші спрацьовують тільки на верхньому елементі, але не на нижньому.

Те ж саме з перетягуванням елементів. Смайл завжди знаходиться поверх інших елементів, тому події спрацьовують на ньому. Які б обробники ми не ставили на нижні елементи, вони не будуть виконані.

Існує метод document.elementFromPoint (clientX, clientY). Він повертає найглибше вкладений елемент за заданими координатами вікна (або null, якщо зазначені координати знаходяться за межами вікна).

Ми можемо використовувати його, щоб з будь-якого обробника подій миші з'ясувати, над якою ми потенційною ціллю перенесення, ось так:

```
// усередині обробника події миші
smile.hidden = true; // (*) ховаємо елемент, який переноситься
let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
// elemBelow - елемент під смайлем (можлива мішень перенесення)
smile.hidden = false;
```

Нам потрібно заховати смайл перед викликом функції (*). В іншому випадку за цими координатами ми будемо отримувати смайл, адже це і є елемент безпосередньо під курсором: elemBelow = smile. Таким чином, ми ховаємо його і тут же показуємо назад.

Ми можемо використовувати цей код для перевірки того, над яким елементом ми знаходимось в будь-який час. І обробити закінчення перенесення, коли воно станеться.

Приклад (перетягування смайла на квітку):

index.html

```
<html>
<body>
  <p>Перемістіть смайл.</p>
  <img src="E:\Викладання\2025\ОКР\grag_drop\Flover.jpg" id="flover" class="droppable">
  <img src="E:\Викладання\2025\ОКР\grag_drop\Smile.png" id="smile">
<script src="script.js"></script>
</body>
</html>
```

script.js

```
let currentDroppable = null;
   smile.onmousedown = function(event) {
    let shiftX = event.clientX - smile.getBoundingClientRect().left;
    let shiftY = event.clientY - smile.getBoundingClientRect().top;
    smile.style.position = 'absolute';
    smile.style.zIndex = 1000;
    document.body.append(smile);
  moveAt(event.pageX, event.pageY);
    function moveAt(pageX, pageY) {
     smile.style.left = pageX - shiftX + 'px';
     smile.style.top = pageY - shiftY + 'px';
    }
    function onMouseMove(event) {
     moveAt(event.pageX, event.pageY);
     smile.hidden = true;
     let elemBelow = document.elementFromPoint(event.clientX,
event.clientY);
     smile.hidden = false;
     if (!elemBelow) return;
     let droppableBelow = elemBelow.closest('.droppable');
     if (currentDroppable != droppableBelow) {
      if (currentDroppable) {
       leaveDroppable(currentDroppable);
      }
      currentDroppable = droppableBelow;
      if (currentDroppable) {
       enterDroppable(currentDroppable);
```

```
      }
    }
  }
  document.addEventListener('mousemove', onMouseMove);
  smile.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    smile.onmouseup = null;
  };
};
function enterDroppable(elem) {
  elem.style.background = 'pink';
}
function leaveDroppable(elem) {
  elem.style.background = '';
}
smile.ondragstart = function() {
  return false;
};
```

Результат у браузері (рисунок 8.7):

Рисунок 8.7 – Drag'n'Drop

Протягом всього процесу в змінній currentDroppable зберігається поточна потенційна ціль перенесення, над якою ми зараз, можемо її підсвітити або зробити ще якісь дії.

Порядок виконання лабораторної роботи

1) ознайомитися з теоретичними відомостями;
2) виконати завдання;
3) оформити звіт;
4) продемонструвати результат на комп'ютері і захистити роботу

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Visua Studio Code Microsoft Learn. [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/training/modules/install-configure-visual-studio-code/
2. Prem Kumar Ponuthorai, Jon Loeliger. Version Control with Git, 3rd Edition – O'Reilly Media, 2022 – 546 p.
3. Pro Git, 2nd Edition. [Електронний ресурс] – Режим доступу: https://gitscm.com/book/en/v2
4. Jeremy Keith, Jeffrey Zeldman. HTML5 for Web Designers - A Book Apart, 2010 – 87 p
5. Ben Frain. Responsive Web Design with HTML5 and CSS3 - Packt Publishing? 2012, 324 p
6. Довідник по HTML/CSS/JS. [Електронний ресурс] – Режим доступу: https://html-css.co.ua/
7. Jon Duckett, JavaScript and JQuery: Interactive Front-End Web Development 1st Edition - John Wiley and Sons Ltd, 2014 – 640 p
8. The Modern JavaScript Tutorial. [Електронний ресурс] – Режим доступу: https://javascript.info/