

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Електронне видання

**ОСНОВИ КЛІЄНТСЬКОЇ РОЗРОБКИ**

МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ  
РОБІТ ДЛЯ СТУДЕНТІВ

СПЕЦІАЛЬНОСТІ

121, 123, 126

*Затверджено методичною радою ФІОТ*

Київ

КПІ ім. Ігоря Сікорського

2024

Основи клієнтської: Метод. вказівки до виконання лаб. робіт для студентів усіх форм навчання спеціальності 121,123,126 / Уклад.:М.С. – К.: КПІ ім. Ігоря Сікорського, 2024.

*Гриф надано Методичною радою ФІОТ*

*(Протокол від )*

Електронне видання  
**ОСНОВИ КЛІЄНТСЬКОЇ РОЗРОБКИ**

Методичні вказівки до виконання лабораторний робіт для студентів спеціальності  
121, 123, 126

Укладач: Хмелюк Марина Сергіївна, ст. викл. каф. ІСТ, ФІОТ

Відповідальний

редактор О.А. Амонс, канд. техн. наук, доц. каф. ІСТ, ФІОТ

Рецензенти: А.М. Волокита, канд. техн. наук, доц. каф. обчислювальної техніки,  
ФІОТ

*За редакцією укладачів*

## ЗМІСТ

		Стор.
<a href="#"><u>ВСТУП</u></a>	.....	5
<a href="#"><u>Тематики</u></a>	.....	6
<a href="#"><u>Лабораторна робота 1. Системи контролю версій. Git. Проект. Структура проекту.</u></a>	.....	7
<a href="#"><u>Лабораторна робота 2. HTML. Структура документа. Заголовки. Гіперпосилання. Форматування тексту. Кольори. Списки. Зображення. Фон. Таблиці, фрейми</u></a>	.....	37
<a href="#"><u>Лабораторна робота 3. CSS. Внутрішні стилі. Стилi рівня документа. Зовнішні стилі. Оформлення тексту, поля, заповнення, межі. Застосування стилів для таблиць і списків</u></a>	.....	59
<a href="#"><u>Лабораторна робота 4. CSS. Контекстні селектори. Сусідні селектори. Дочірні селектори.</u></a>	.....	92
<a href="#"><u>Лабораторна робота 5. CSS. Блочні елементи. Рядкові елементи. Позиціонування. Псевдокласи. Псевдоелементи</u></a>	.....	99
<a href="#"><u>Лабораторна робота 6. JavaScript. Внутрішні, зовнішні скрипти. Змінні. Умови. Цикли. Функції. DOM. BOM. Браузер: документ(document)</u></a>	.....	120
<a href="#"><u>Лабораторна робота 7. JavaScript. Події. Обробники подій. Спливання. Делегування подій.</u></a>	.....	205
<a href="#"><u>Лабораторна робота 8. JavaScript. Події миші</u></a>	.....	235
<a href="#"><u>СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ</u></a>	.....	254
<a href="#"><u>Титульний аркуш для звіту</u></a>	.....	255

## **ВСТУП**

Методичні вказівки до виконання лабораторних робіт з кредитного модуля *Основи клієнтської розробки* повністю охоплюють навчальну програму та формують у студентів відповідні знання, навички та вміння відповідно до вимог освітньої програми підготовки фахівця за спеціальністю 121, 123, 126.

## **Тематики**

Для виконання лабораторних робіт обрати тематику:

- 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

***Теми в межах групи не повинні повторюватись.***

# Лабораторна робота №1

## 1. Тема

Системи контролю версій. Git. Проєкт. Структура проєкту.

## 2. 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**. 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**).

## 3. Теоретичні відомості

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

VCS надає такі можливості:

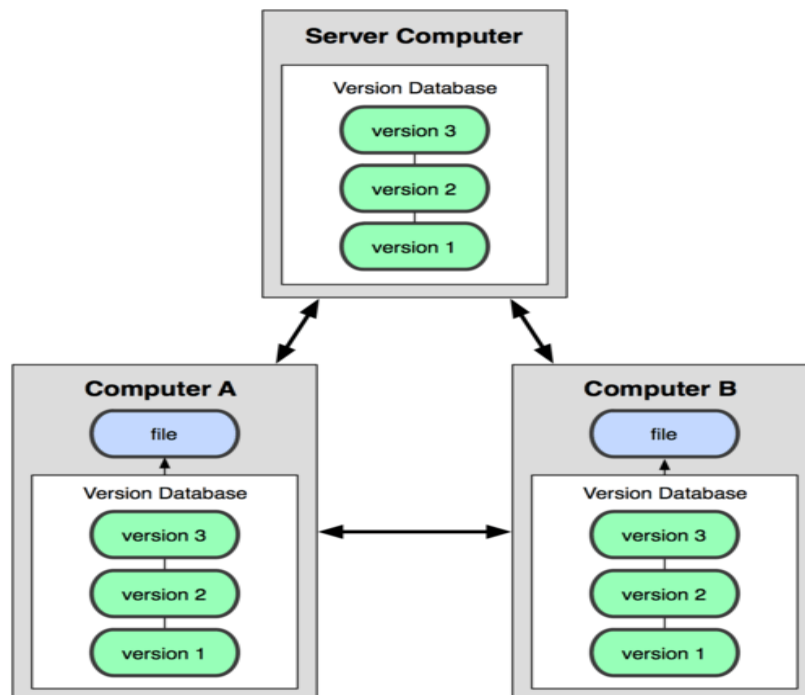
- підтримку зберігання файлів у репозиторії;
- підтримку історії версій файлів у репозиторії;
- знаходження конфліктів при зміні вихідного коду та забезпечення синхронізації при роботі в багатокористувацькому середовищі розробки;
- відстеження змін авторів.

Звичайний цикл роботи розробника з СКВ виглядає так:

- **Оновлення робочої копії.** Розробник виконує операцію оновлення робочої копії з сервера на скільки можливо

- **Модифікація проекту.** Розробник локально модифікує проект, 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.

### Розподілені системи контролю версій



### Розподілені системи контролю версій (Git)

#### *Переваги:*

- 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 проекту.

**Git** – 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, змінено і підготовлено.

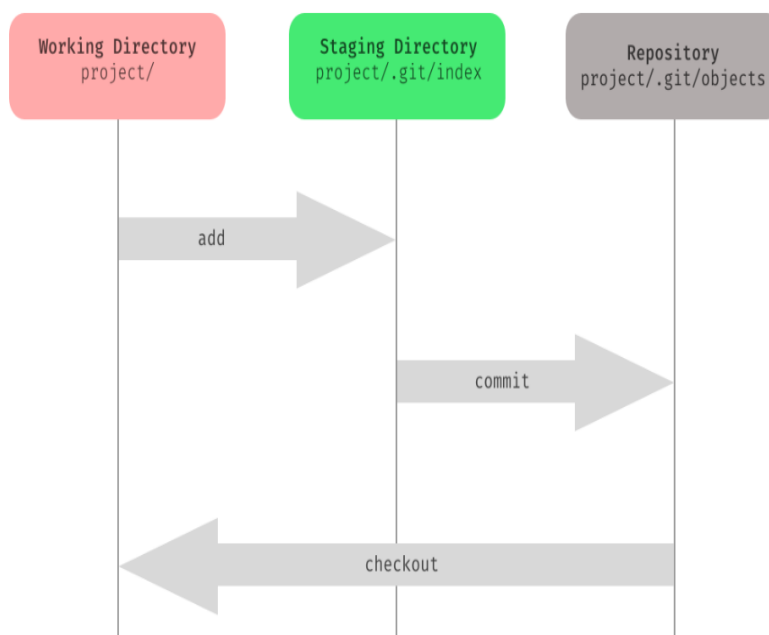
- **Зафіксовано (Підтверджено):** дані безпечно зберігаються локально.
- **Змінено:** ще не підтверджено.
- **Підготовлено:** позначений для переходу до наступного коміту.

Ці стани приводять до трьох розділів:

**Робочий каталог:** це копія версії проекту.

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

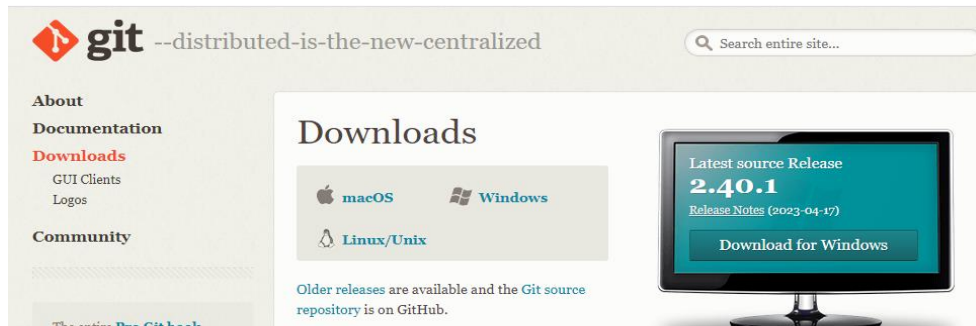
**Каталог Git:** зберігає метадані та об'єкти бази даних для вашого проекту.



## Установка Git

Необхідно встановити Git. можна пройти за цим посиланням на офіційне джерело <https://git-scm.com/downloads>





Найголовніша команда Git help - виводить список усіх команд із коротким описом.

```
e:\Викладання\2022\ОКР_2022\TestGit>git help
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
          [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
          [-p | --paginate] [-P | --no-pager] [--no-replace-objects] [--bare]
          [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
          [--config-env=<name>=<envvar>] <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
  clone      Clone a repository into a new directory
  init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
  add        Add file contents to the index
  mv         Move or rename a file, a directory, or a symlink
  restore    Restore working tree files
  rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
  bisect     Use binary search to find the commit that introduced a bug
  diff       Show changes between commits, commit and working tree, etc
  grep       Print lines matching a pattern
  log        Show commit logs
  show       Show various types of objects
  status     Show the working tree status

grow, mark and tweak your common history
  branch     List, create, or delete branches
  commit     Record changes to the repository
  merge      Join two or more development histories together
  rebase     Reapply commits on top of another base tip
  reset      Reset current HEAD to the specified state
  switch     Switch branches
  tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
  fetch      Download objects and refs from another repository
  pull       Fetch from and integrate with another repository or a local branch
  push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
```

## Структура директорії .git/

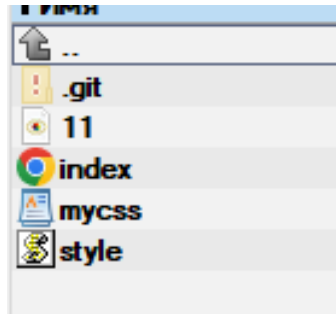
робота з Git буде починається з того, що потрібно проініціалізувати Git директорію у своєму проєкті. Це робиться за допомогою команди:

**git init**

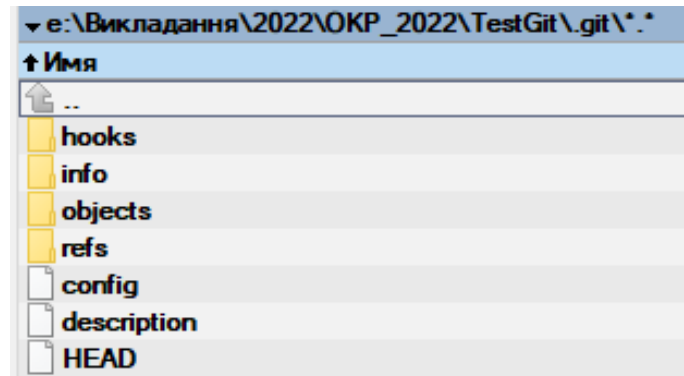
Її необхідно ввести в корені вашого проєкту. Це створить у поточному каталозі новий підкаталог .git

```
e:\Викладання\2022\ОКР_2022\TestGit>git init
Initialized empty Git repository in E:/Викладання/2022/ОКР_2022/TestGit/.git/

e:\Викладання\2022\ОКР_2022\TestGit>
```



з наступним вмістом:



У цій директорії буде вся конфігурація Git та історія проекту. За бажанням можна редагувати ці файли вручну, вносячи необхідні зміни в історію проекту.

### 1. **config**

Цей файл містить налаштування Git репозиторію. тут можна зберігати email та ім'я користувача.

### 2. **description**

Цей файл призначений для GitWeb - це веб-інтерфейс, написаний для перегляду Git-репозиторія використовуючи веб-браузер і містить інформацію про проект (назва проекту та його опис).

### 3. **hooks**

У цьому каталозі Git надає набір скриптів, які можуть автоматично запускатись під час виконання git команд. Наприклад, можна написати скрипт, який редагуватиме повідомлення комміту відповідно до ваших вимог.

### 4. **info**

Каталог info містить файл exclude, в якому можна вказувати будь-які файли, і Git не додаватиме їх у свою історію.

### 5. **refs**

Каталог refs зберігає копію посилань на об'єкти коммітів у локальних і віддалених гілках.

## 6. logs

Каталог logs зберігає історію проекту для всіх гілок у вашому проекті.

## 7. objects

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

## 8. 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).

## 9. head

Файл містить посилання на поточну гілку, в якій ви працюєте

## 10. orig\_head

Щоразу під час злиття до цього файлу потрапляє SHA гілка, з якою проводилося злиття.

## 11. 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)

## 12. merge\_head

Файл зберігає посилання у вигляді SHA на гілки, які брали участь у git merge (злиття)

## 13. commit\_editmsg

Файл містить останнє введення вами повідомлення комміту.

Робота над проектом з використанням СКВ обмежена певними діями:

1. Внести зміни до проекту;
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 проекту);
4. Запушити - `git push` (надіслати результати роботи на віддалений сервер, щоб інші розробники теж мали до них доступ).

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 проекті відстежуються Git, а також які зміни будуть включені до наступного коміту.

```
e:\Викладання\2022\ОКР_2022\TestGit>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    11.JPG
    111.txt
    index.html
    mycss.css
    style.js

nothing added to commit but untracked files present (use "git add" to track)
```

**`git add 111.txt`**

```
e:\Викладання\2022\ОКР_2022\TestGit>git add 111.txt
e:\Викладання\2022\ОКР_2022\TestGit>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   111.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    11.JPG
    index.html
    mycss.css
    style.js
```

Файл додано до індексу. Тепер можна закомітити внесені зміни та додати коментар. **`git commit -m "first commit"`**

```
e:\Викладання\2022\ОКР_2022\TestGit>git commit -m "first commit"
[master (root-commit) 7b86ff1] first commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 111.txt
```

Відредагуємо файл і перевіримо статус

```
e:\Викладання\2022\ОКР_2022\TestGit>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   111.txt

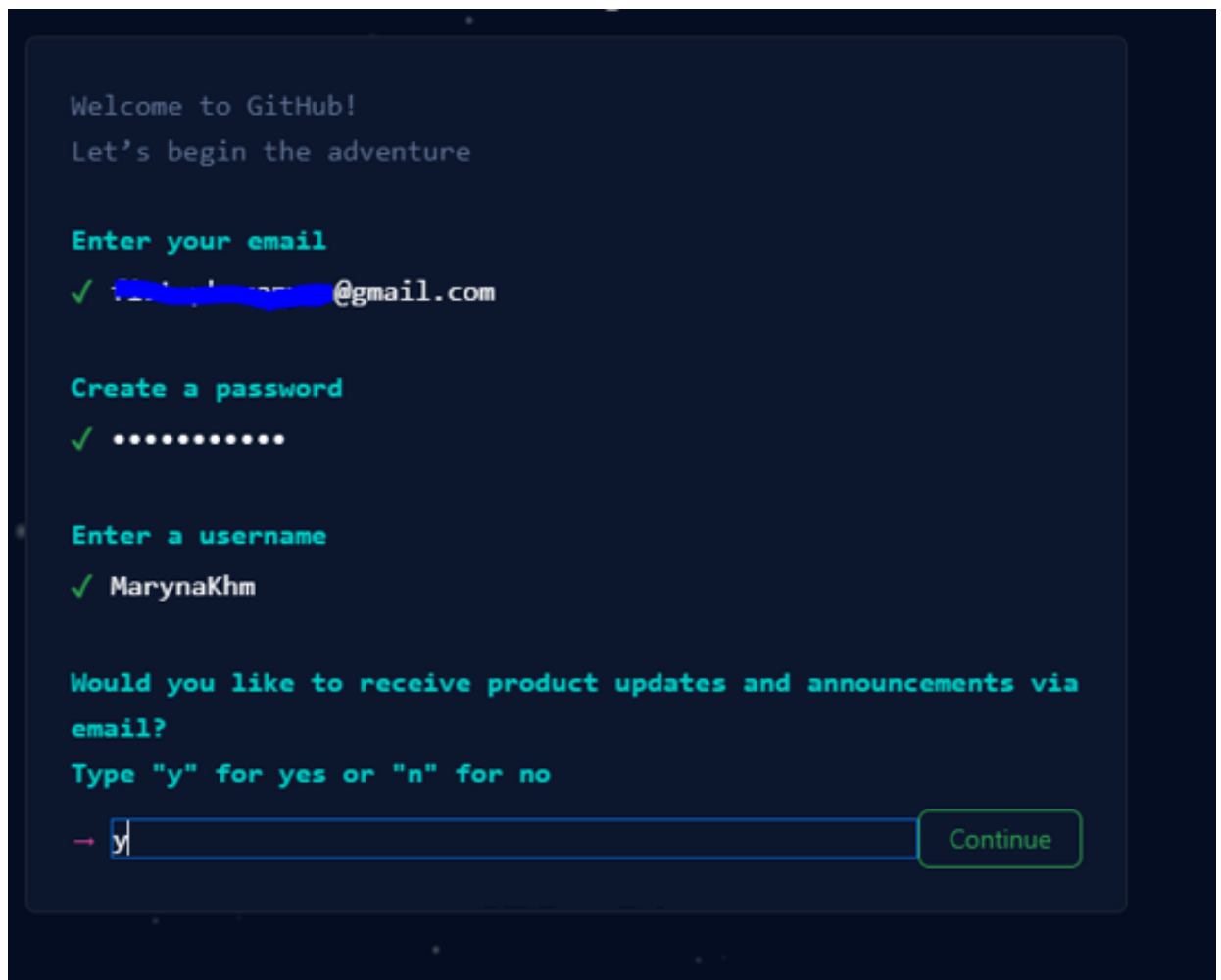
no changes added to commit (use "git add" and/or "git commit -a")
```

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

Ми все записуємо та зберігаємо зміни на своїй машині, тепер нам потрібно відправити версію нашої історії на віддалений сервер.

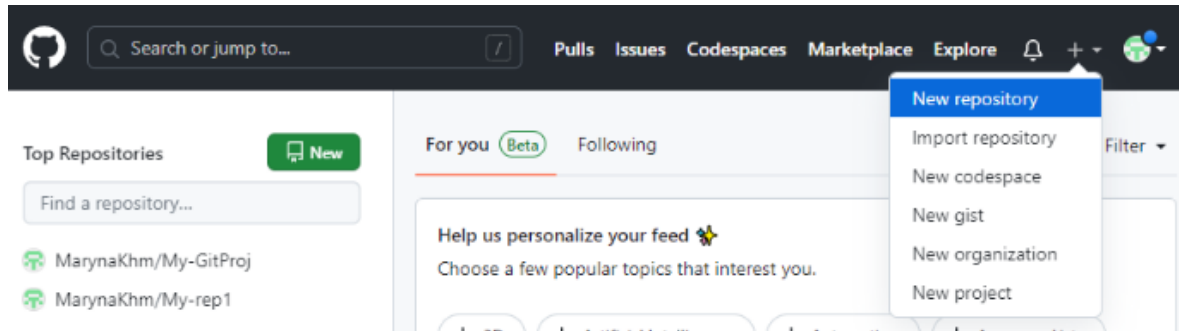
Можна скористатись репозиторієм на **GitHub**.

### Реєстрація на [GitHUB.com](https://github.com)



The image shows a terminal window with a dark blue background and light blue text. It displays the GitHub registration process. At the top, it says "Welcome to GitHub! Let's begin the adventure". Below this, there are three sections for registration: "Enter your email" with a green checkmark and a blue redacted email address followed by "@gmail.com"; "Create a password" with a green checkmark and a series of dots; and "Enter a username" with a green checkmark and the username "MarynaKhm". At the bottom, it asks "Would you like to receive product updates and announcements via email?" and "Type 'y' for yes or 'n' for no". A text input field contains the letter 'y', and a green "Continue" button is visible to its right.

### Створення репозиторія



Ввести коротку назву, видимість сховища, опис (опціонально), натиснути

## Create repository

### Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \* MarynaKhm / Repository name \* TestGit

✓ TestGit is available.

Great repository names are short and memorable. Need inspiration? How about [probable-bassoon](#)?

Description (optional)

- ☒ **Public**  
Anyone on the internet can see this repository. You choose who can commit.
- ☐ **Private**  
You choose who can see and commit to this repository.

Initialize this repository with:

- ☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

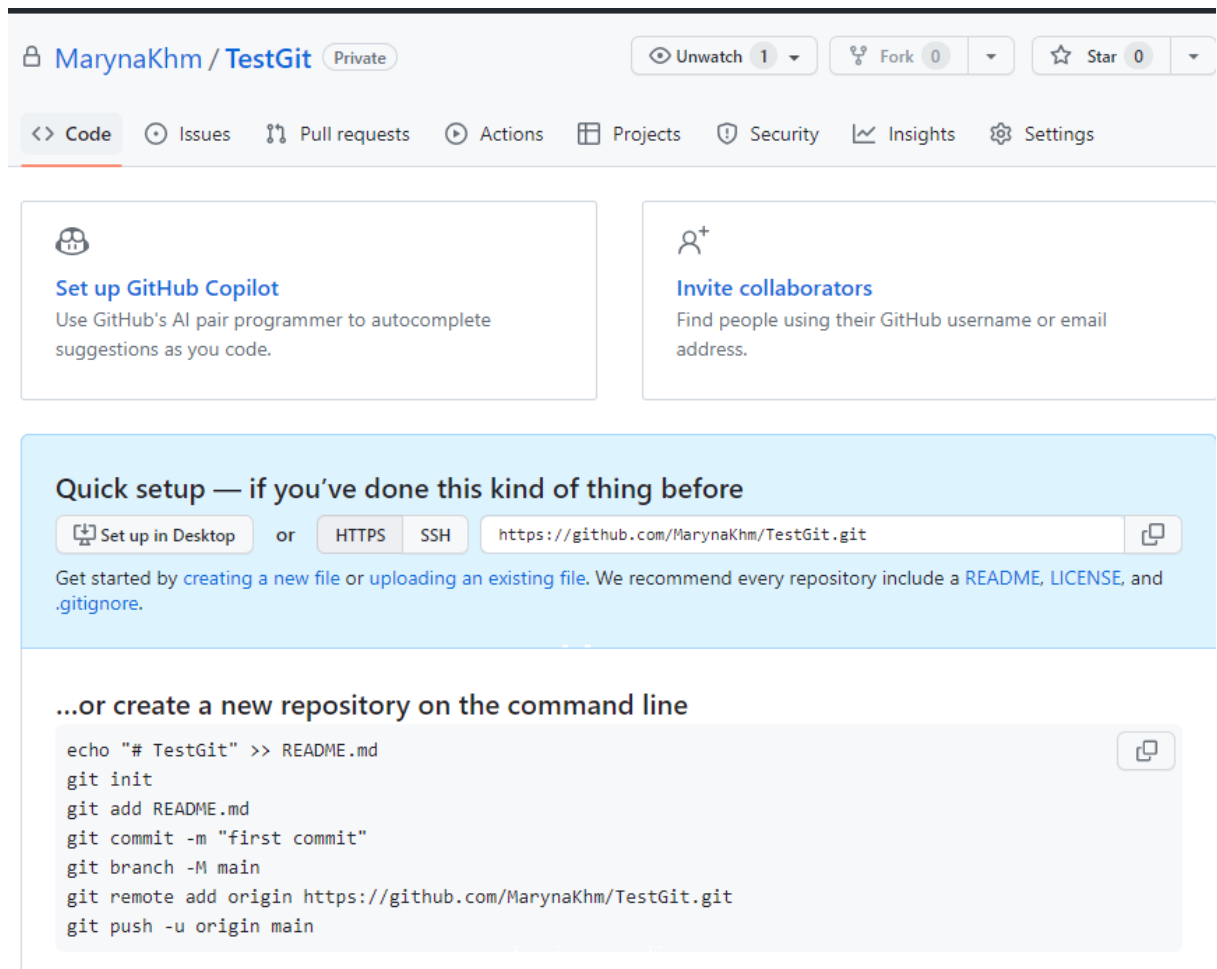
Choose a license

License: None

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

You are creating a public repository in your personal account.

Create repository



Далі необхідно додати віддалений репозиторій у ваш Git:

**git remote add <remote\_name> <remote\_repo\_url>** - команда створює віддалене сховище з посиланням на локальний репозиторій. З цього моменту можна звернутися до видаленого репозиторію через це посилання.

```
e:\Викладання\2022\ОКР_2022\TestGit>git remote add origin https://github.com/MarynaKhm/TestGit.git
```

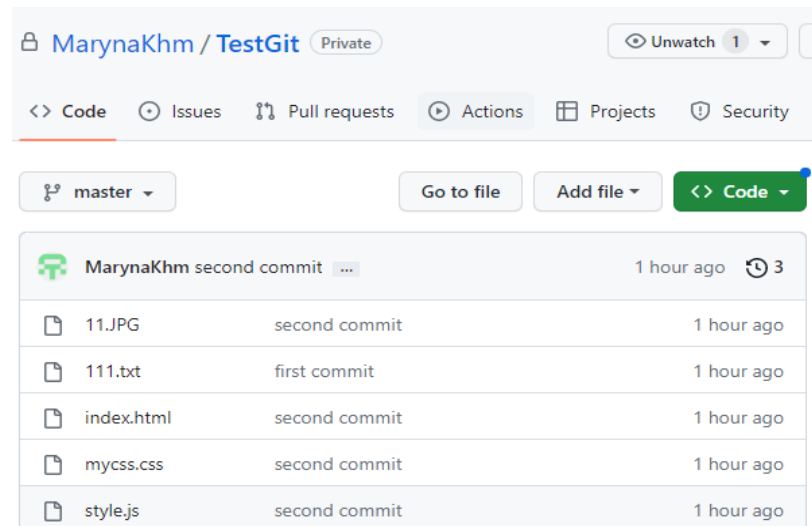
«origin» є коротким іменем для віддаленого репозиторію, на який він буде посилатись (може бути будь-яке ім'я).

відправимо результат нашої роботи в репозиторій.

**git push origin master**

Тепер історія змін вашого проєкту буде зберігатися у віддаленому репозиторії.

```
e:\Викладання\2022\ОКР_2022\TestGit>git push origin master
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 16 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (11/11), 4.71 KiB | 4.71 MiB/s, done.
Total 11 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/MarynaKhm/TestGit.git
* [new branch]      master -> master
```



**Гілка в Git** - це простий переміщений показчик на один з коммітів. За замовчуванням, ім'я основної гілки в Git — master. Як тільки ви почнете створювати комміти, гілка master завжди вказуватиме на останній комміт.

Для перегляду історії в Git є ряд команд:

**git log** - призначена для відображення всієї історії вашого проєкту, Дозволяє дізнатися, які зміни ви внесли раніше.



```
e:\Викладання\2022\ОКР_2022\TestGit>git log
commit 8040a7432e9cbaa431c336eb7a1c94383c54b78a (HEAD -> master, origin/master)
Author: MarynaKhm <[REDACTED]@gmail.com>
Date: Mon May 15 15:01:11 2023 +0300

    second commit

commit aeb73d278af0d956b2b60b55cb0145e48ee2acb8
Author: MarynaKhm <[REDACTED]@gmail.com>
Date: Mon May 15 15:00:13 2023 +0300

    second commit

commit 7b86ff1d720883ea934657db53a96cbbd9edb8b2
Author: MarynaKhm <[REDACTED]@gmail.com>
Date: Mon May 15 14:53:21 2023 +0300

    first commit
```

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

```
e:\Викладання\2022\ОКР_2022\TestGit>git show
commit 8040a7432e9cbaa431c336eb7a1c94383c54b78a (HEAD -> master, origin/master)
Author: MarynaKhm <[REDACTED]@gmail.com>
Date: Mon May 15 15:01:11 2023 +0300

    second commit

diff --git a/11.JPG b/11.JPG
new file mode 100644
index 0000000..b892725
Binary files /dev/null and b/11.JPG differ
```

**git reflog** - виводить упорядкований список комітів, на який вказує HEAD (відображає історію всіх ваших переміщень по проєкту). 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\2022\ОКР_2022\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
```

**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 комміт 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

```
e:\Викладання\2022\ОКР_2022\TestGit>git log --oneline
434c04f (HEAD -> master, origin/master) 3 commit
8040a74 second commit
aeb73d2 second commit
7b86ff1 first commit
```

Зроблено 4 коміти

Відкотимось до 2 коміта **git reset --soft**

```
e:\Викладання\2022\ОКР_2022\TestGit>git reset --soft aeb73d2

e:\Викладання\2022\ОКР_2022\TestGit>git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   11.JPG
    modified:   111.txt

e:\Викладання\2022\ОКР_2022\TestGit>git log --oneline
aeb73d2 (HEAD -> master) second commit
7b86ff1 first commit
```

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

Відкотимось до 2 коміта **git reset --mixed**

```
e:\Викладання\2022\ОКР_2022\TestGit>git reset --mixed aeb73d2
Unstaged changes after reset:
M      111.txt

e:\Викладання\2022\ОКР_2022\TestGit>git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   111.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        11.JPG

no changes added to commit (use "git add" and/or "git commit -a")

e:\Викладання\2022\ОКР_2022\TestGit>git log --oneline
aeb73d2 (HEAD -> master) second commit
7b86ff1 first commit
```

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 **git reset --hard**

```
e:\Викладання\2022\ОКР_2022\TestGit>git reset --hard aeb73d2
HEAD is now at aeb73d2 second commit

e:\Викладання\2022\ОКР_2022\TestGit>git status
On branch master
nothing to commit, working tree clean

e:\Викладання\2022\ОКР_2022\TestGit>git log --oneline
aeb73d2 (HEAD -> master) second commit
7b86ff1 first commit
```

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

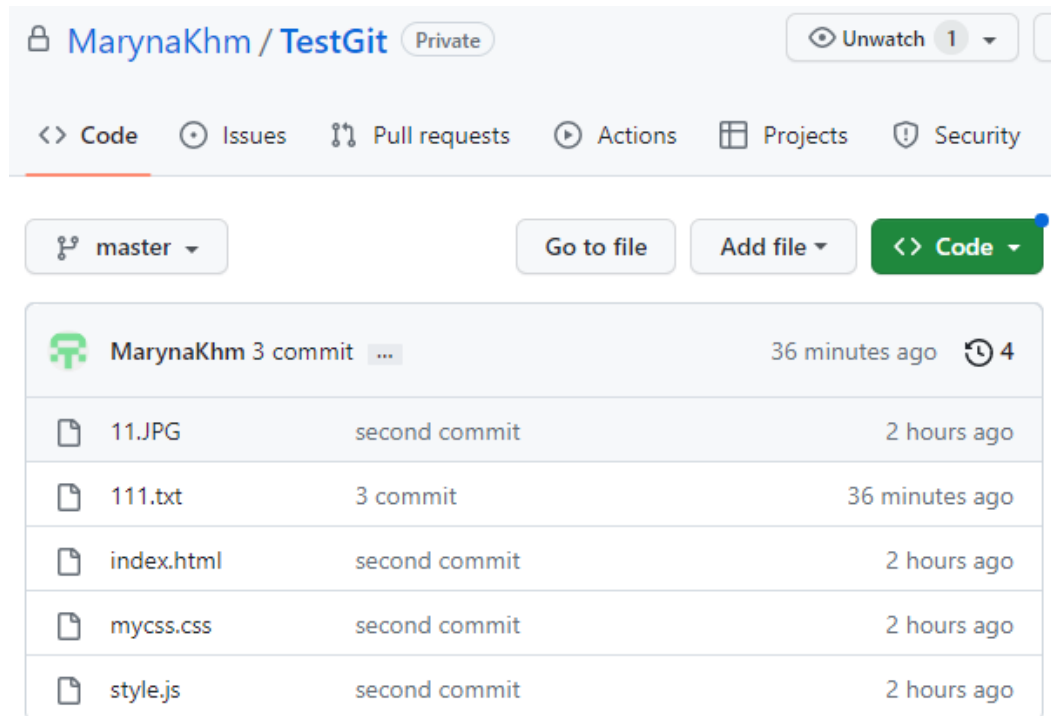
## Гілкування в Git

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)

```
e:\Викладання\2022\ОКР_2022\TestGit>git log --oneline
aeb73d2 (HEAD -> master) second commit
7b86ff1 first commit
```

У віддаленому репозиторії



MarynaKhm / TestGit Private Unwatch 1

Code Issues Pull requests Actions Projects Security

master Go to file Add file Code

MarynaKhm 3 commit ...	36 minutes ago	4
11.JPG	second commit	2 hours ago
111.txt	3 commit	36 minutes ago
index.html	second commit	2 hours ago
mycss.css	second commit	2 hours ago
style.js	second commit	2 hours ago

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

```
e:\Викладання\2022\ОКР_2022\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\2022\ОКР_2022\TestGit>git log --onelin
fatal: unrecognized argument: --onelin

e:\Викладання\2022\ОКР_2022\TestGit>git log --oneline
434c04f (HEAD -> master, origin/master) 3 commit
8040a74 second commit
aeb73d2 second commit
7b86ff1 first commit
```

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

```
e:\Викладання\2022\ОКР_2022\TestGit>git branch
* master
```

Для створення нової гілки, її потрібно створити

**git branch <branch\_name>**

```
e:\Викладання\2022\ОКР_2022\TestGit>git branch new1

e:\Викладання\2022\ОКР_2022\TestGit>git branch
* master
new1
```

\* вказує на поточну гілку, в якій ми працюємо.

Для того, щоб перейти на іншу гілку, є команда

**git checkout <branch\_name>**

```
e:\Викладання\2022\ОКР_2022\TestGit>git checkout new1
Switched to branch 'new1'
```

Зробимо зміни в файлі 111.txt, зробимо коміт

```
e:\Викладання\2022\ОКР_2022\TestGit>git commit -m "5 commit, 1 commit to new1"
On branch new1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   111.txt

no changes added to commit (use "git add" and/or "git commit -a")

e:\Викладання\2022\ОКР_2022\TestGit>git commit -a -m "5 commit, 1 commit to new1"
[new1 f8540d1] 5 commit, 1 commit to new1
1 file changed, 2 insertions(+), 1 deletion(-)

e:\Викладання\2022\ОКР_2022\TestGit>git log --oneline
f8540d1 (HEAD -> new1) 5 commit, 1 commit to new1
434c04f (origin/master, master) 3 commit
8040a74 second commit
aeb73d2 second commit
7b86ff1 first commit
```

Перейдемо в гілку master

### **git checkout master**

Переглянемо історію **git log --oneline**, і переконаємось, що все залишилося без змін.

```
e:\Викладання\2022\ОКР_2022\TestGit>git checkout master
Switched to branch 'master'

e:\Викладання\2022\ОКР_2022\TestGit>git log --oneline
434c04f (HEAD -> master, origin/master) 3 commit
8040a74 second commit
aeb73d2 second commit
7b86ff1 first commit
```

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 ній виконати:

### **git merge new1**

```
e:\Викладання\2022\ОКР_2022\TestGit>git merge new1
Updating 434c04f..f8540d1
Fast-forward
 111.txt | 3 ++-
1 file changed, 2 insertions(+), 1 deletion(-)
```



Тепер непотрібної гілки можна позбутися і видалити її за допомогою команди `git branch -d <branch_name>`.

```
e:\Викладання\2022\ОКР_2022\TestGit>git checkout new1
Switched to branch 'new1'

e:\Викладання\2022\ОКР_2022\TestGit>git branch
  master
* new1

e:\Викладання\2022\ОКР_2022\TestGit>git checkout master
Switched to branch 'master'

e:\Викладання\2022\ОКР_2022\TestGit>git branch
* master
  new1

e:\Викладання\2022\ОКР_2022\TestGit>git branch -d new1
Deleted branch new1 (was f8540d1).

e:\Викладання\2022\ОКР_2022\TestGit>git branch
* master
```

```
e:\Викладання\2022\ОКР_2022\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
```

```
e:\Викладання\2022\ОКР_2022\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
```

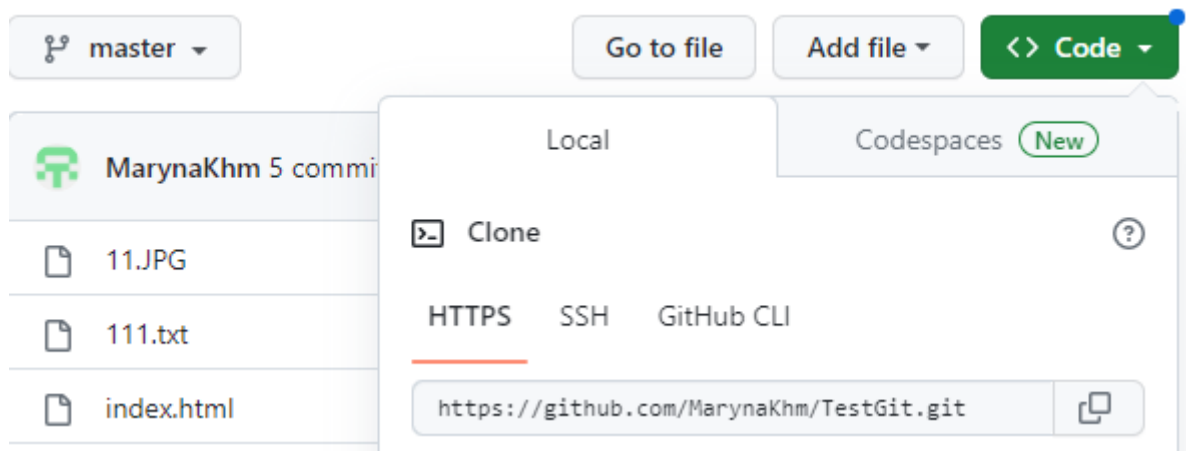


Що потрібно зробити, щоб інша людина отримала всі ваші зміни?

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

Якщо у людини раніше не було проєкту, то їй доведеться його «клонувати» собі:

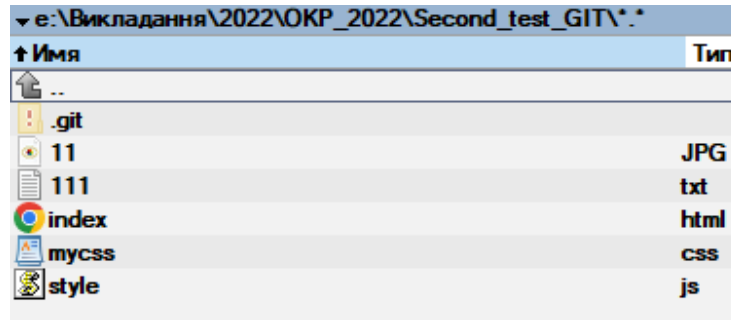
**git clone <URL repo>**



Адресу репозиторію на GitHub можна отримати, натиснувши на зелену кнопку Code.

```
e:\Викладання\2022\ОКР_2022\TestGit>cd..
e:\Викладання\2022\ОКР_2022>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.
```





Имя	Тип
..	
.git	
11	JPG
111	txt
index	html
mycss	css
style	js

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**

**git pull origin master**

```
e:\Викладання\2022\ОКР_2022>cd
e:\Викладання\2022\ОКР_2022>cd e:\Викладання\2022\ОКР_2022\Second_test_GIT\
```

```
e:\Викладання\2022\ОКР_2022\Second_test_GIT>git checkout master
Already on 'master'
Your branch is up to date with 'origin/master'.
```

```
e:\Викладання\2022\ОКР_2022\Second_test_GIT>git pull origin master
From https://github.com/MarynaKhm/TestGit
* branch          master      -> FETCH_HEAD
Already up to date.
```

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

```
<<<<<<< HEAD
```

```
=====
```

```
>>>>>>> master
```

---

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

Ці нові рядки можна розглядати як «роздільники конфлікту».

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>**

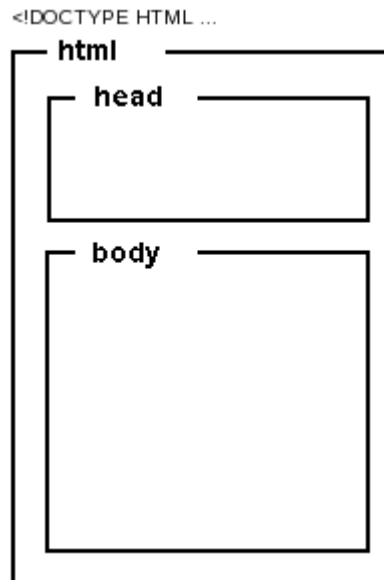
**HTML** - це мова розмітки, який представляє прості правила оформлення і компактний набір структурних і семантичних елементів розмітки (тегів).

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

Не використовуйте кириличні назви файлів і папок - давайте їм англійські назви! Перша сторінка завжди носить назву index.html.

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

Структура html-документа:



HTML-документ складається з тексту, який являє собою інформаційний вміст і спеціальних засобів мови HTML - тегів розмітки, які визначають структуру і зовнішній вигляд документа при його відображенні браузером. Структура HTML-документа досить проста:

1. Опис документа починається з вказівки його типу (секція DOCTYPE).
2. Текст документа полягає в тег `<html>`. Текст документа складається з заголовка і тіла, які виділяються відповідно тегам `<head>` і `<body>`.
  - 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 керуючу розмітку документа (теги), які використовуються браузером.

Наявність секції DOCTYPE дозволяє вказати браузеру, який тип документа йому належить розбирати, тобто, які вимоги потрібно виконувати при обробці гіпертексту.

Приклади DOCTYPE:

- <! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"  
"http://www.w3.org/TR/html4/frameset.dtd">

Гіпертекстовий документ в форматі HTML 4.01, що містить фрейми.

- <! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">

Гіпертекстовий документ в форматі HTML 4.01 із суворим синтаксисом (тобто не використані застарілі і не рекомендовані теги).

- <! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

Гіпертекстовий документ в форматі HTML 4.01 з нестрогим («перехідним») синтаксисом (тобто використані застарілі або які не рекомендовані теги і атрибути).

- <! DOCTYPE HTML> оголошення для документів HTML5.

Стандарт вимагає, щоб секція DOCTYPE була присутня в документі, тому що це дозволяє прискорити і поліпшити обробку гіпертексту. Це досягається за рахунок того, що браузер може не робити припущень про те, як інтерпретувати теги, а звіритися зі стандартним визначенням (файлом .dtd).

Заголовок призначений для розміщення метаінформації, яка описує веб-документ як такий.

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

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

Приклад опису метаданих:

<head>

<meta name = "author" content = "рядок"> - автор веб-документа

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

<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>

Блок <body> містить те, що потрібно показати користувачеві: текст, зображення, впроваджені об'єкти тощо.

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

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

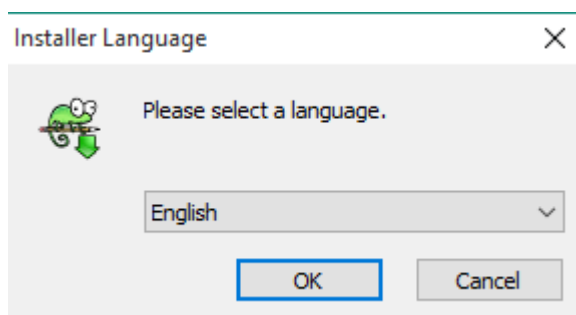
<title> ... </ title> - назва документа (те, що відображається в заголовку вікна браузера)

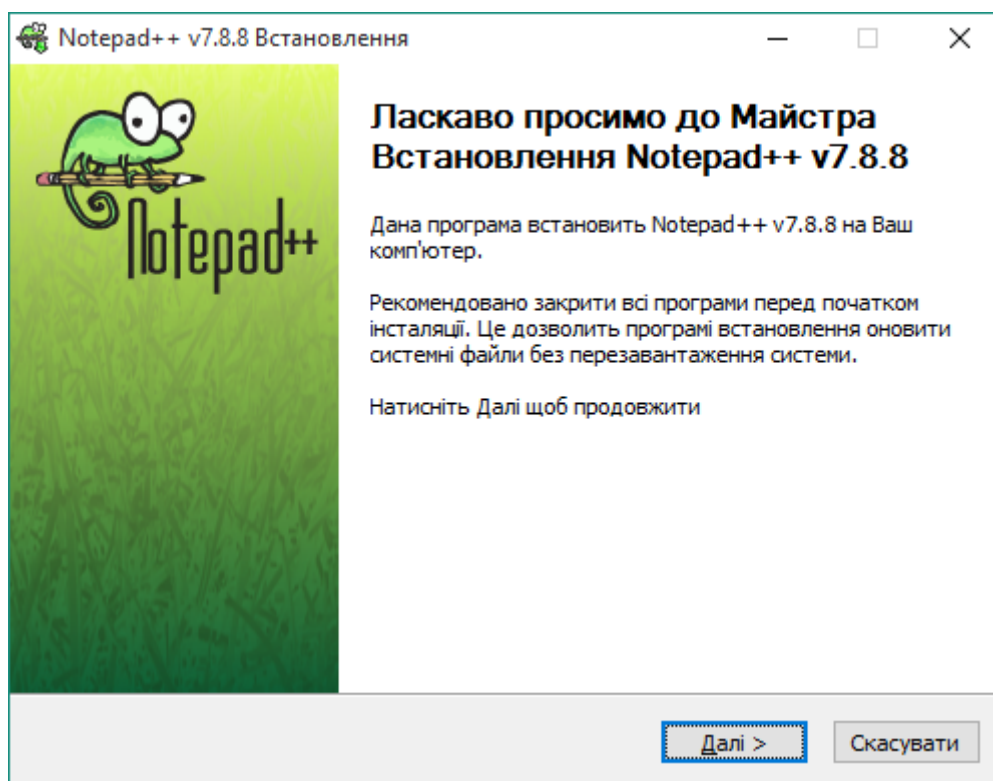
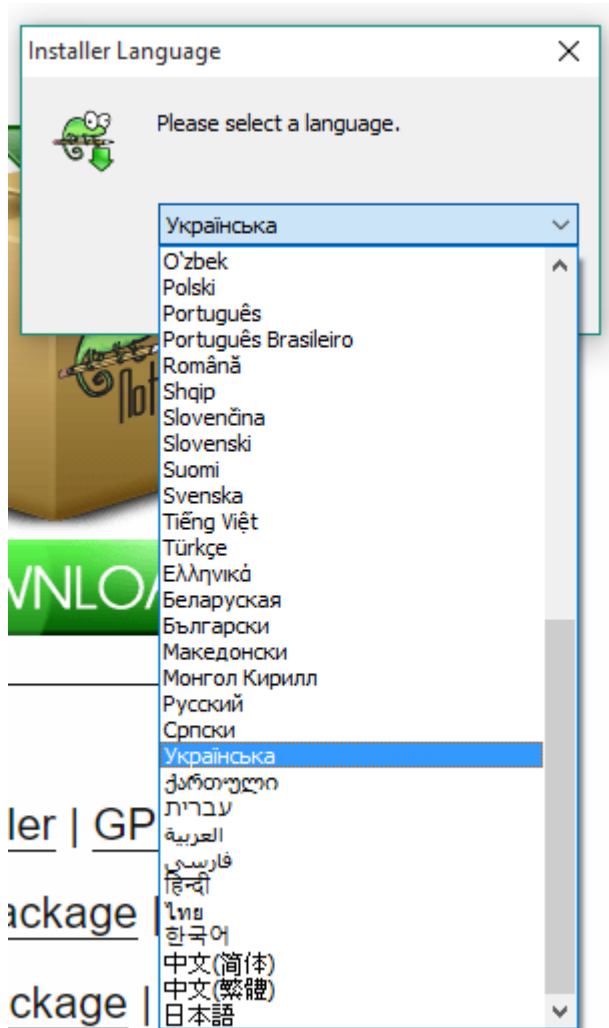
<body> ... </ body> - контейнер тіла документа

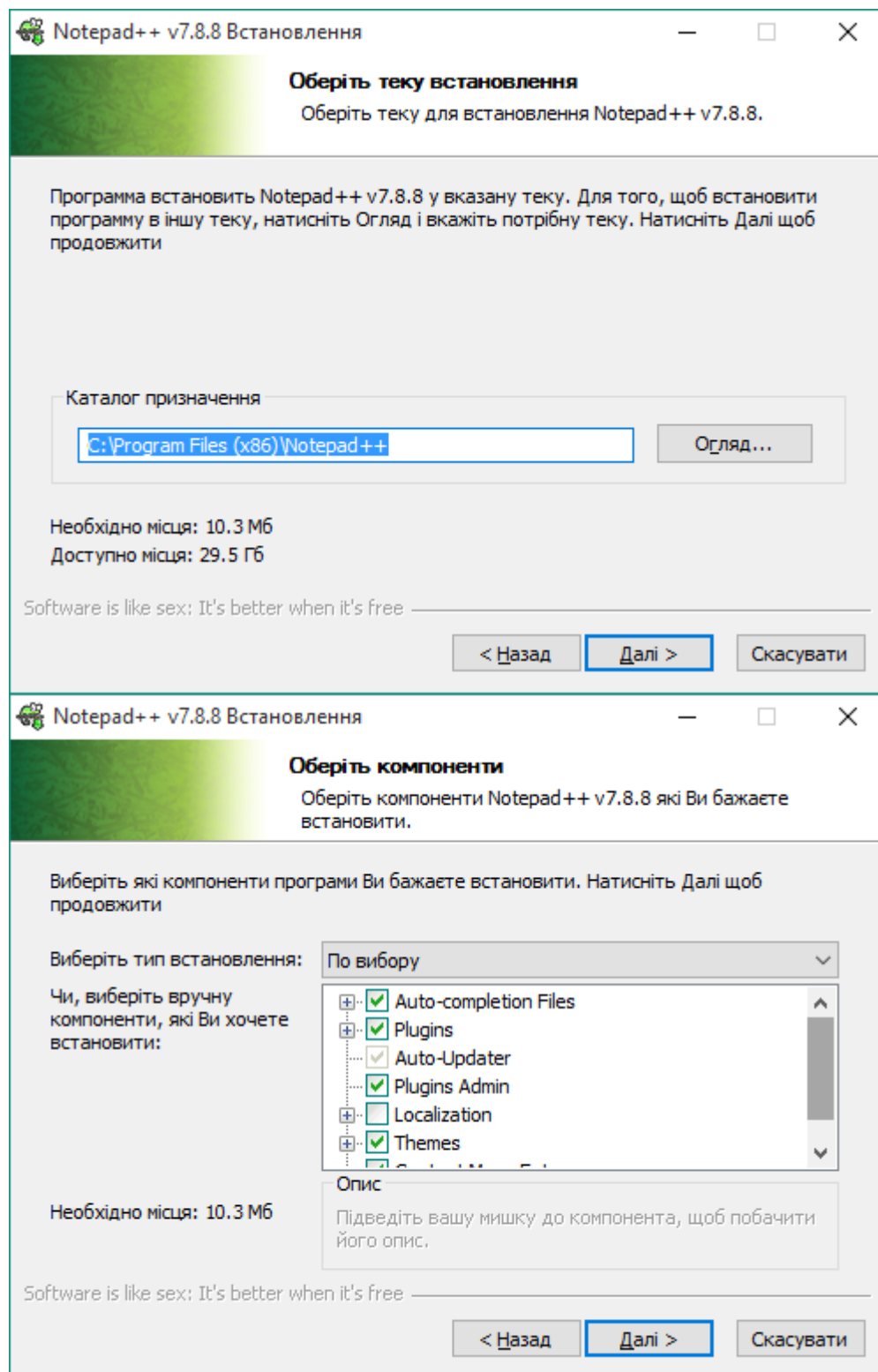
## Середовище розробки

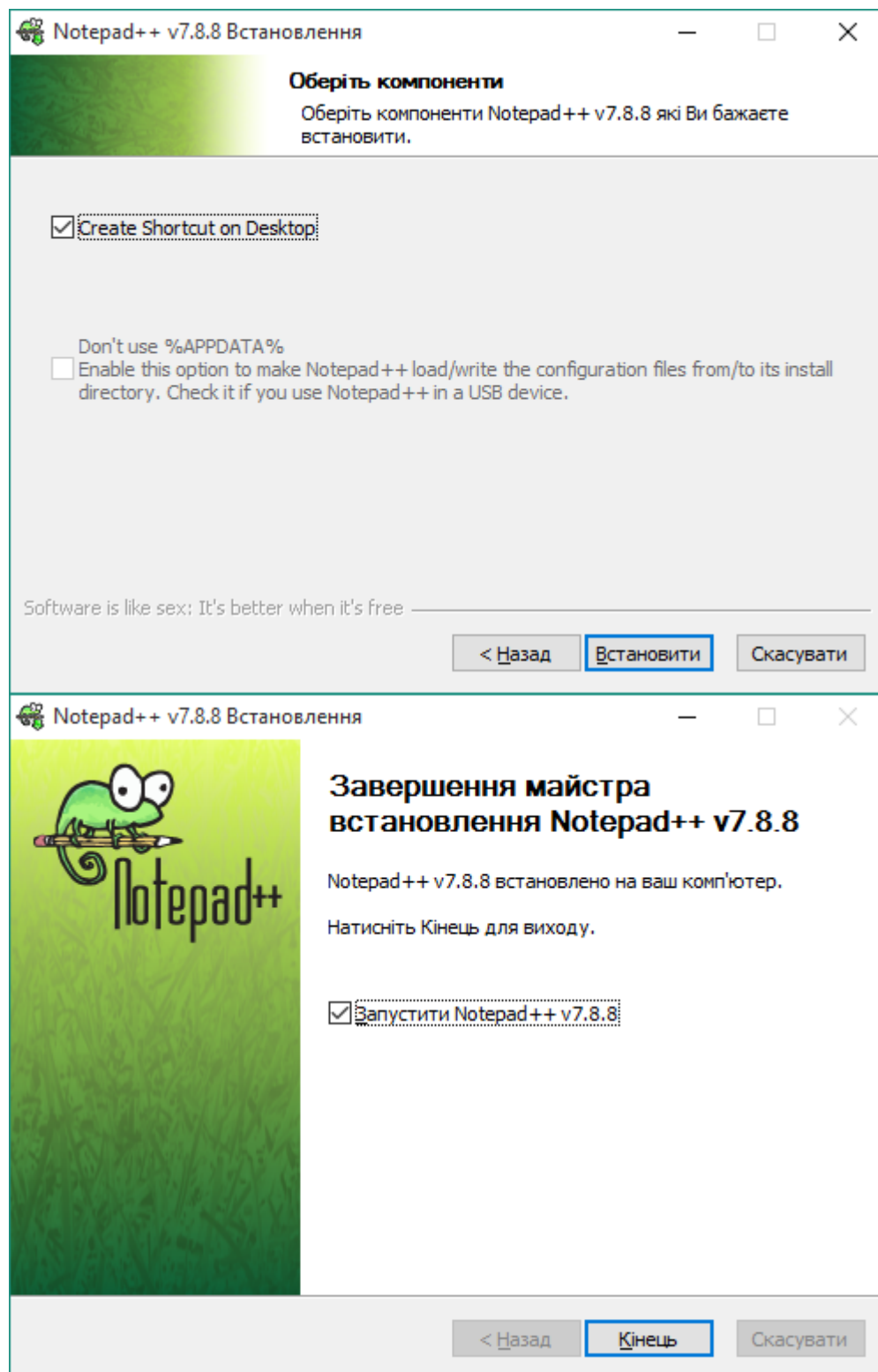
Створити html-документ можна в будь-якому текстовому редакторі, зберегти файл з розширенням .html

Установка та налаштування **Notepad++**











C:\Program Files (x86)\Notepad++\change.log - Notepad++

Файл Редагувати Пошук Вигляд Кодування Мова Налаштування Інструменти Макрос Виконати Плагіни Вікно ?

change.log

```
1 Notepad++ v7.8.8 Enhancements & bug-fixes:
2
3 1. Fix accented characters in ANSI files not found in "find in files" and "replace in files" commands issues.
4 2. Add an option to improve rendering special Unicode characters by using Scintilla's DirectWrite technology.
5 3. Fix URL invisible issue in dark themes.
6 4. Fix the focus not on the opening new file issue.
7 5. Fix Workspace (Project panel), Folder As Workspace and function list keep focus issue after double clicking an item.
8 6. Add Ctrl+Backspace ability to delete word for comboboxes in Find/Replace dialog.
9 7. Add ability to find-all in selected text.
10 8. Fix wrong treatment of backslashes as escape sequences in autocompletion.
11 9. Enhance "Remove Empty Lines" command: Allow scope to be limited by an active selection.
12 10. Fix loading of project & session files by drag & drop not working issue.
13 11. Fix block selection cursor wrong positions after typing TAB.
14 12. Add confirmation prompt to "Replace all in all opened documents" command to avoid PBKAC.
15
16
17 Included plugins:
18
19 1. NppExport v0.2.9
20 2. Converter 4.2.1
21 3. Mime Tool 2.5
22
23
24 Updater (Installer only):
25
26 * WinGUp (for Notepad++) v5.1.1
```

Normal text file length: 1,185 lines: 26 Ln: 1 Col: 1 Sel: 0|0 Windows (CR LF) UTF-8 INS

Налаштування

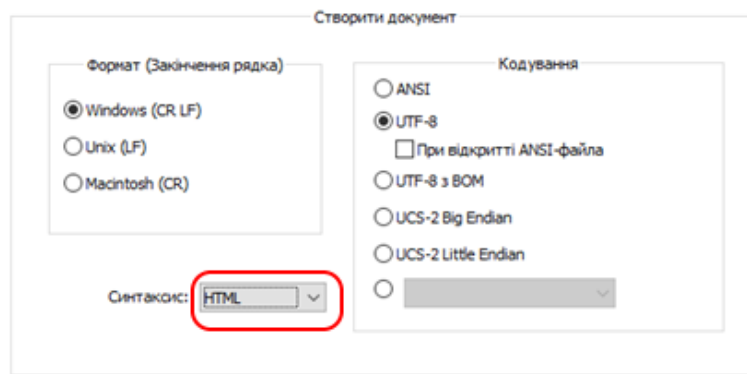
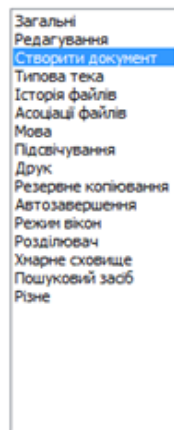
- Загальні
- Редагування
- Створити документ
- Типова тека
- Історія файлів
- Асоціації файлів
- Мова
- Підсвічування
- Друк
- Резервне копіювання
- Автозавершення
- Режим вікон
- Розділювач
- Хмарне сховище
- Пошуковий засіб
- Різне

Створити документ

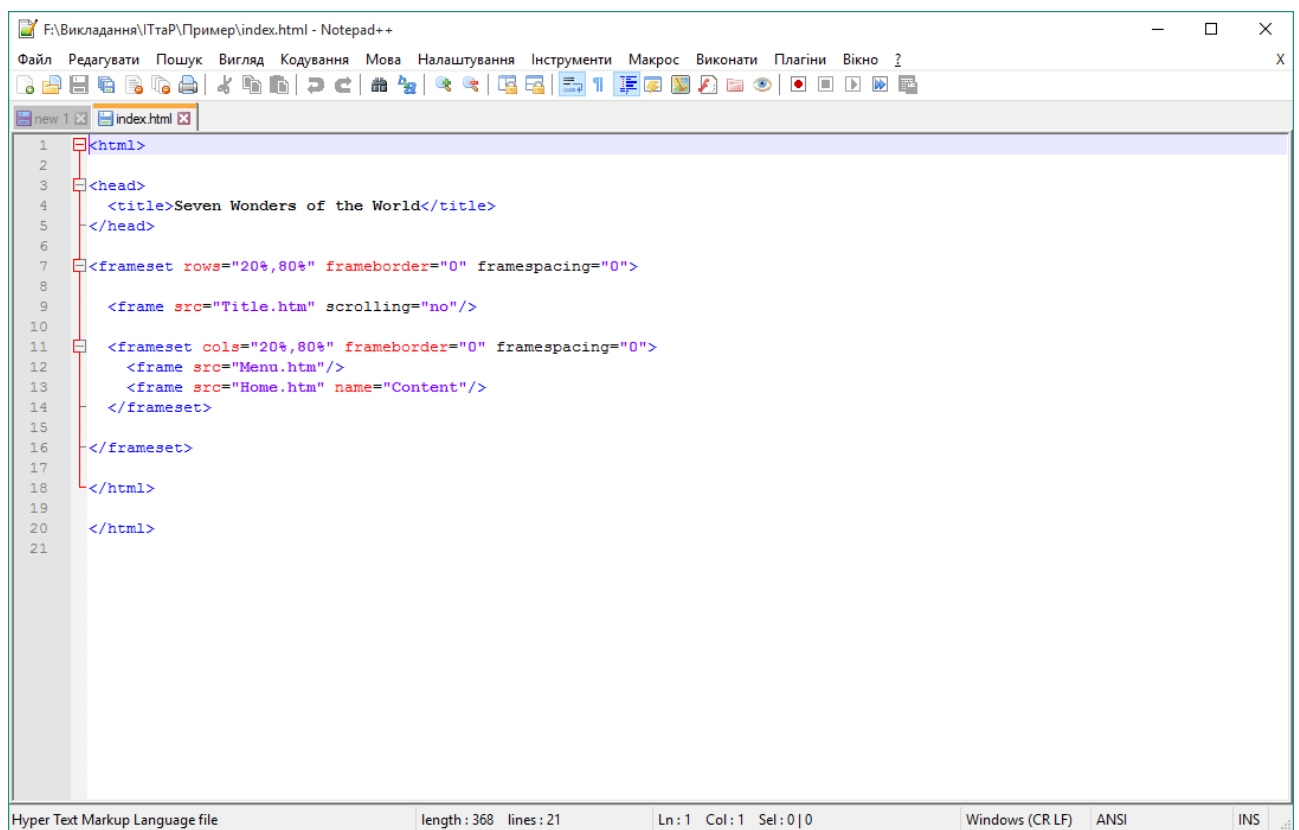
Формат (Закінчення рядка)	Кодування
<input checked="" type="radio"/> Windows (CR LF)	<input type="radio"/> ANSI
<input type="radio"/> Unix (LF)	<input checked="" type="radio"/> UTF-8
<input type="radio"/> Macintosh (CR)	<input type="checkbox"/> Грати відкритті ANSI-файла
	<input type="radio"/> UTF-8 з BOM
	<input type="radio"/> UCS-2 Big Endian
	<input type="radio"/> UCS-2 Little Endian
	<input type="radio"/> <input type="text"/>

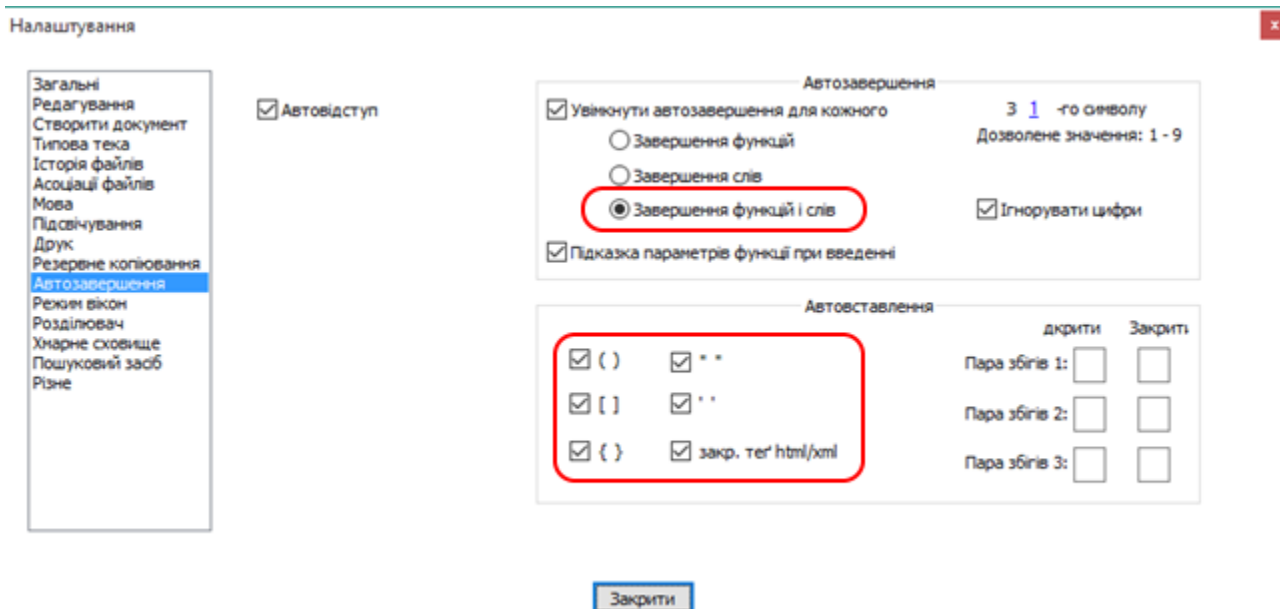
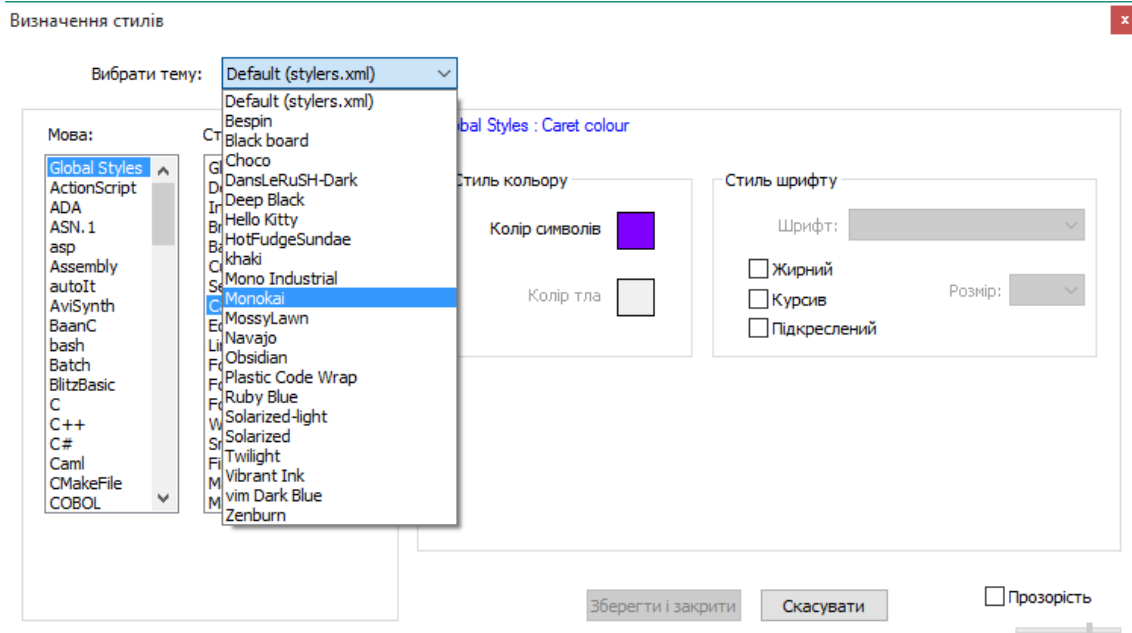
Синтаксис: Normal Text

Закрити



Закрити





#### 4. Порядок виконання лабораторної роботи

- 1) ознайомитися з теоретичними відомостями;
- 2) виконати завдання
- 3) оформити звіт, який включатиме: титульний аркуш, завдання, теоретичні відомості, результати і висновки по роботі;
- 4) продемонструвати результат на комп'ютері і захистити лабораторну роботу

## Лабораторна робота №2

### 1. Тема

HTML. Структура документа. Заголовки. Гіперпосилання. Форматування тексту. Кольори. Списки. Зображення. Фон. Таблиці, фрейми Структура документа. Заголовки. Гіперпосилання.

### 2. Завдання

Використовувати тему, обрану в 1 роботі.

1) В тілі сторінок додати по декілька абзаців тексту (відповідно до тематики) використовуючи тег `<p></p>`. 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, шрифт, розмір, колір, вирівнювання), використовуючи тег `<font></font>` і відповідні атрибути.

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 `<ul></ul>`, маркування `<ol></ol>`, 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 (rowspan, colspan), групування комірок (colgroup, col), використати відступи (cellspacing, cellpadding).

5) Вставити плаваючий фрейм

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

### 3. Теоретичні відомості

#### Теги

Тег (html-тег, тег розмітки) - керуюча символна послідовність, яка задає спосіб відображення гіпертекстової інформації.

HTML-тег складається з імені, за яким може слідувати необов'язковий список атрибутів. Весь тег (разом з атрибутами) полягає в кутові дужки  $\langle \rangle$ :

`<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 коса риска / (`<html> ... </html>`). Кінцеві теги ніколи не містять атрибутів. Деякі теги не мають кінцевого елемента, наприклад тег `<img>`. Регістр символів для тегів не має значення.

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

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

`<p> ... </p>` - основний текст

`<a> ... </a>` - гіперпосилання

`<ol> ... </ol>` - нумерований список

<ul> ... </ ul> - маркований список  
<li> ... </ 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> ... </ em> - виділення (курсивом)  
<strong> ... </ strong> - посилення (напівжирним шрифтом)  
<br> - примусовий розрив рядка

## Атрибути

Атрибути - це пари виду «властивість = значення», уточнюючі уявлення відповідного тега:

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

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

Примітка: Не дивлячись на необов'язковість лапок, їх все ж варто завжди використовувати.

Атрибути можуть бути обов'язковими і необов'язковими. Необов'язкові атрибути можуть бути опущені, тоді для тега застосовується значення цього

атрибута за замовчуванням. Якщо не вказано обов'язковий атрибут, то вміст тега швидше за все буде відображено неправильно.

Короткий список деяких часто використовуваних 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 веб-сторінки) 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

Гіперпосилання - це особливим чином позначений фрагмент веб-сторінки (текст, зображення та ін.), Який пов'язаний з іншим документом. Для вказівки гіперпосилань використовується тег <a>. Гіперпосилання дозволяють переміщатися між пов'язаними веб-сторінками.

Гіперпосилання умовно можна розділити на наступні види:

- Внутрішні - зв'язуючими документи всередині одного і того ж вузла;
- Зовнішні - зв'язуючі Web-сторінку з документами, що не належать даному вузлу;

- Гіперпосилання на поштову адресу;
- Мітки-якоря - дозволяють переходити відвідувачеві на певні розділи документа.

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

`<a name="якір"> Прив'язка до фрагменту тексту </a>`

`<a href="#якір"> Посилання на якір </a>`

`<a href = "адреса посилання"> текст для клацання миші </a>`

`<a href = " адреса посилання "> <IMG src = "посилання на малюнок"> </a>`

У середині 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)

```
<!DOCTYPE HTML>

<html>

  <head>

    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

    <title>Колір посилань</title>

  </head>

  <body link="red" vlink="blue" alink="pink">

    <p><a href="content.html">Вміст сайту</a></p>

  </body>

</html>
```

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

`target="_blank"` тега `<a>`.



<a href=" pag2.html " target="\_blank"> <IMG src=log.gif> </a>

<a href="pag2.html" target="\_blank"> сторінка відкриється в новому вікні </a>

Для вказівки електронної пошти і запуску електронної програми використовується посилання:

<a href =“mailto:vvv@mail.ru”> Іванов Іван</a>

<a href="#new"> Нові надходження </a> - перехід до рядка тієї ж сторінки з позначкою тегом <a name="new">

<a href="pag2.htm#new1"> примітки </a> - перехід на сторінку сайту pag2 до рядка з позначкою тегом

<a name="new1">

<p> подробиці читайте <a href="pag2.htm"> друга сторінка </a> </ p> - посилання на іншу сторінку того ж сайту

<p> <a href="pag2.htm"> IMG src = log.gif alt = "про нашу фірму" </a> </p> - посилання на іншу сторінку того ж сайту, але посиланням є малюнок

<a href="myfile.exe" title=" файл 10 мегабайт"> Завантажити програму </a> - Посилання з підказкою title

<a Href="http://home.ifmo.ru/index.html"> тест </a> - зовнішнє посилання

Посилання можуть бути абсолютними і відносними.

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

<a href="http://example.com/page.html"> Абсолютне посилання </a>

<a href="http://example.com/images/figure1.gif"> Посилання на сторінку в каталозі </a>

*Відносні* посилання, навпаки, використовують для переходу на внутрішні сторінки сайту. Для них потрібно вказувати шлях щодо посилання:

<a href="/index.html"> Посилання на сторінку в кореневому каталозі </a>

`<a href="page.html#seg1">` Посилання на фрагмент сторінки в поточному каталозі `</a>`

`<a href="images/figure1.gif">` Посилання на сторінку в підкаталозі поточного каталогу `</a>`

`<a href="/docs/manual.html">` Посилання на сторінку в підкаталозі кореневого каталогу `</a>`

`<a href="../files/index.html">` Посилання на сторінку в вищележачому каталозі `</a>`.

### Колір в HTML

Колір в HTML може бути заданий ключовими словами - назвами квітів на англійській мові:

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)

Black = "#000000"	Green = "#008000"
Silver = "#C0C0C0"	Lime = "#00FF00"
Gray = "#808080"	Olive = "#808000"
White = "#FFFFFF"	Yellow = "#FFFF00"
Maroon = "#800000"	Navy = "#000080"
Red = "#FF0000"	Blue = "#0000FF"
Purple = "#800080"	Teal = "#008080"
Fuchsia = "#FF00FF"	Aqua = "#00FFFF"

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> - формувати текст як підрядковий індекс

<sup> ... </ sup> - формувати текст як надрядковий індекс

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

<font> ... </ font> - встановлює розмір, колір і гарнітуру тексту

### Приклад:

HTML-код:  $101_{\text{2}} = 5$

У браузері:  $101_2 = 5$

HTML-код:  $2^8 = 256$

У браузері:  $2^8 = 256$

Всі ці характеристики задаються за допомогою відповідних атрибутів в тезі управління шрифтом

<font> текст </font>

Атрибути:

color = "колір" - задає колір тексту

face = "шрифт" - визначає гарнітуру тексту; значенням атрибута може бути список шрифтів, перерахованих через кому - в цьому випадку вибирається перший доступний шрифт

size = "1-7" - встановлює розмір шрифту (від 1 до 7)

### Приклад:

HTML-код:

`<font face = "Tahoma" size = "2" color = "gray"> текст </font>`

У браузері: текст

`<p> ... </ p>` - задає початок і кінець параграфа

Атрибут:

`align = "..."` - визначає режим вирівнювання тексту

`left` - по лівому краю (за замовчуванням)

`center` - по центру

`right` - по правому краю

`justify` - по ширині

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

`<br>` - перенесення рядка

Тег `<hr>` - виводить горизонтальну розділову лінію

Атрибути:

`align = "..."` - визначає режим вирівнювання лінії

`left` - по лівому краю

`center` - по центру (за замовчуванням)

`right` - по правому краю

`noshade` - використовувати суцільну лінію замість об'ємної

`size = "N"` - товщина лінії в пікселях

`width = "N"` - ширина лінії в пікселях або відсотках по відношенню до ширини екрану.

### Робота зі списками

У HTML є можливість створювати нумеровані і маркіровані списки.

`<Ol> ... </ ol>` - створює нумерований список елементів

Атрибути:

`start = "N"` - почати нумерацію з числа N

type = "..." - визначає формат нумерації

l - арабські цифри (за замовчуванням)

A - великі літери (A, B, C)

a - малі літери (a, b, c)

I - прописні римські цифри (I, II, III)

i - рядкові римські цифри (i, ii, iii)

<ul> ... </ul> - створює маркований список елементів

Атрибут:

type = "..." - визначає формат маркера

disk - диск (за замовчуванням)

circle - окружність

square - квадрат

<li> ... </li> - задає елемент списку в нумерованому або маркованому списку

Атрибути:

type = "..." - формат номера або маркера (див. опис <ol> і <ul>)

value = "N" - задає номер елемента списку

Приклад:

<ol>

<li> арабські цифри (за замовчуванням) </li>

<li type = "A"> прописні букви </li>

<li type = "a"> малі літери </li>

<li type = "I"> прописні римські цифри </li>

<li type = "i"> рядкові римські цифри </li>

</ol>

<ul>

<li> диск (за замовчуванням) </li>

<li type = "circle"> окружність </li>

<li type = "square"> квадрат </li>

</ul>

`<dl> ... </dl>` - створює список визначень

`<dt>` створює термін

`<dd>` задає визначення цього терміна.

Приклад:

```
<dl>
```

```
  <dt>Термін 1</dt>
```

```
  <dd>Визначення терміна 1</dd>
```

```
  <dt>Термін 2</dt>
```

```
  <dd> Визначення терміна 2</dd>
```

```
</dl>
```

### Зображення

Вставка зображень на сторінці здійснюється непарним тегом `<img>`.

Обов'язковий атрибут `src` вказує абсолютний або відносний URL зображення.

Стандартними форматами зображень є GIF, PNG и JPEG.

Уникайте використання інших форматів зображень (наприклад, BMP або TIFF), тому що вони можуть не підтримуватися окремими типами браузерів.

Атрибути:

`align = "..."` - визначає режим вирівнювання зображення щодо 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

`alt = "..."` - визначає текст, що описує зображення для браузерів без підтримки графіки (або з відключеною графікою), пошукових машин і т.п.

`border = "N"` - встановлює товщину рамки навколо зображень, рівній N пікселів, 0 - для відключення рамки

`height = "N"` - висота зображення в пікселях або відсотках

`width = "N"` - ширина зображення в пікселях або відсотках

Браузер визначає розмір зображення автоматично. Для прискорення завантаження рекомендується вказувати розмір зображення атрибутами height і width, щоб браузер не обчислював цей розмір автоматично після завантаження зображення. Також цими атрибутами можна розтягнути / стиснути зображення по горизонталі / вертикалі, але таке масштабування призведе до втрати якості.

Зображення може бути зроблено посиланням, шляхом приміщення всередину тега <a>. В цьому випадку навколо зображення автоматично з'являється рамка. Товщина рамки задається атрибутом border. Зазвичай рамку прибирають, вказуючи border = "0" в тезі <img>.

#### Приклади:

HTML-сторінка знаходиться в папці site, а зображення picture.jpg знаходиться в папці site / images /.

```
<img src = "images / picture.jpg" alt = "фотографія">
```

Зображення знаходиться на іншому сайті в Інтернет

```
<img src = "http://example.com/pics/tree.gif" alt = "дерево">
```

```
<a href="log.gif " target="_blank" width = "200 " hight = "200 "></a> картинка  
відкриється в новому вікні з розмірами 200-200</a>
```

#### Фонове зображення сторінки

Можна задавати адресу фонового зображення для сторінки в атрибуті background тега <body>. Фонове зображення відображається в натуральну величину. Якщо розмір зображення менше розміру вікна браузера, то малюнок повторюється по горизонталі вправо і по вертикалі вниз.

```
<body background = "bg1.jpg"> </ body>
```

**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.

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

Всі інші елементи таблиці - текст, малюнки, списки - повинні бути вкладеними в нього. Допускається також вкладення таблиць одна в іншу, тобто вмістом комірки може бути інша таблиця.

Теги `<tr> </tr>` і `<td> </td>` - описують рядки і стовпці (елементи таблиці).

Тег `<th> </th>` - описує заголовки в першому рядку таблиці.

Тег `<caption> </caption>` - описує заголовок таблиці.

TR = table row

TD = table data

TH = table header

Приклад:

```
<Table>
```

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

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

```
</ Table>
```

Кількість тегів `<tr> ... </tr>` визначає кількість рядків. У кожному тезі рядки повинно бути одне і те ж число тегів `<td> ... </td>`, яке дорівнює кількості стовпців, інакше таблиця відобразиться неправильно. Можна створювати вкладені таблиці: вкладати таблицю в комірку іншої таблиці.

`<table> ... </table>` - визначає початок і кінець коду таблиці, містить в собі теги рядків і комірок.

Атрибути:

`align = "..."` - визначає режим вирівнювання таблиці щодо тексту в рядку

`left` - по лівому краю

`right` - по правому краю



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

Приклад:

```
<table border = "1">  
<tr align = "center" valign = "top">  
<td width = 120 height = 100> по центру, по верхній межі </td>  
<td align = "right" valign = "middle" width = 200> за правій межі, по середині  
</td>  
</tr>  
</table>
```

`background = "URL"` - задає фоновий малюнок у таблиці

`bgcolor = "колір"` - колір фону таблиці

`border = "N"` - встановлює товщину меж таблиці, рівну N пікселів (0 для відключення)

`bordercolor = "колір"` - колір рамки

`bordercolorlight` = колір - колір рамки зліва і зверху

`bordercolordark` = колір - колір рамки праворуч і знизу

`title = "Текст"` - підказка

`width` = число - ширина таблиці у відсотках або пікселях

```
<table width = "500">
```

```
<table width = "100%">
```

`frame = "..."` - описує параметри зовнішньої рамки.

`box` - відображає всі частини рамки навколо таблиці

`void` - видаляє всі рамки навколо таблиці

`above` - рамка тільки зверху

`below` - рамка тільки знизу

`lhs` - рамка тільки зліва

`rhs` - рамка тільки справа

`vsides` - рамка тільки зліва і справа

hsides - рамка тільки зверху і знизу

cellpadding = "N" - розмір поля навколо вмісту кожної комірки

Приклад:

*cellpadding = "0" cellpadding = "15"*

cellspacing = "N" - розмір вільного простору між комірками

Приклад:

*cellspacing = "0" cellspacing = "15"*

<tr> ... </tr> - визначає рядок елементів таблиці

Атрибути:

align = "..." - визначає режим вирівнювання вмісту комірок рядка

left - по лівому краю

center - по центру

right - по правому краю

justify - по ширині

background = "URL" - URL зображення, яке заповнить фон комірок рядка

bgcolor = "колір" - колір фону комірок рядка

valign = "..." - визначає режим вирівнювання вмісту комірок рядка по вертикалі

top - по верхньому краю

middle - по середині (за замовчуванням)

bottom - по нижньому краю

<td> ... </td> - визначає комірку даних таблиці

атрибути:

align = "..." - визначає режим вирівнювання вмісту комірки

left - по лівому краю

center - по центру

right - по правому краю

background = "URL" - URL зображення, яке заповнить фон комірки

bgcolor = "колір" - колір фону комірки

valign = "..." - визначає режим вирівнювання вмісту комірки по вертикалі

top - по верхньому краю

middle - по середині (за замовчуванням)

bottom - по нижньому краю

height = "N" - висота комірки в пікселях

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

<col> и <colgroup>- задає ширину і інші характеристики однієї або декількох стовпців таблиці.

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

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

Атрибути для <col> і <colgroup>:

align - Встановлює вирівнювання вмісту колонки по краю.

span - Кількість колонок, до яких слід застосовувати атрибути.

valign - Задає вертикальне вирівнювання вмісту колонки.

width - Ширина колонок.

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:

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

Рядки HTML таблиці можна розділити на три семантичні секції: header, body і footer (заголовок, тіло і нижній колонтитул)

<thead> - означає заголовок таблиці і містить теги <th> ... </th> замість <td> ... </td>

<tbody> означає колекцію рядків таблиці, які містять дані

<tfoot> означає нижній колонтитул таблиці, але описується до тега <tbody>

<table>

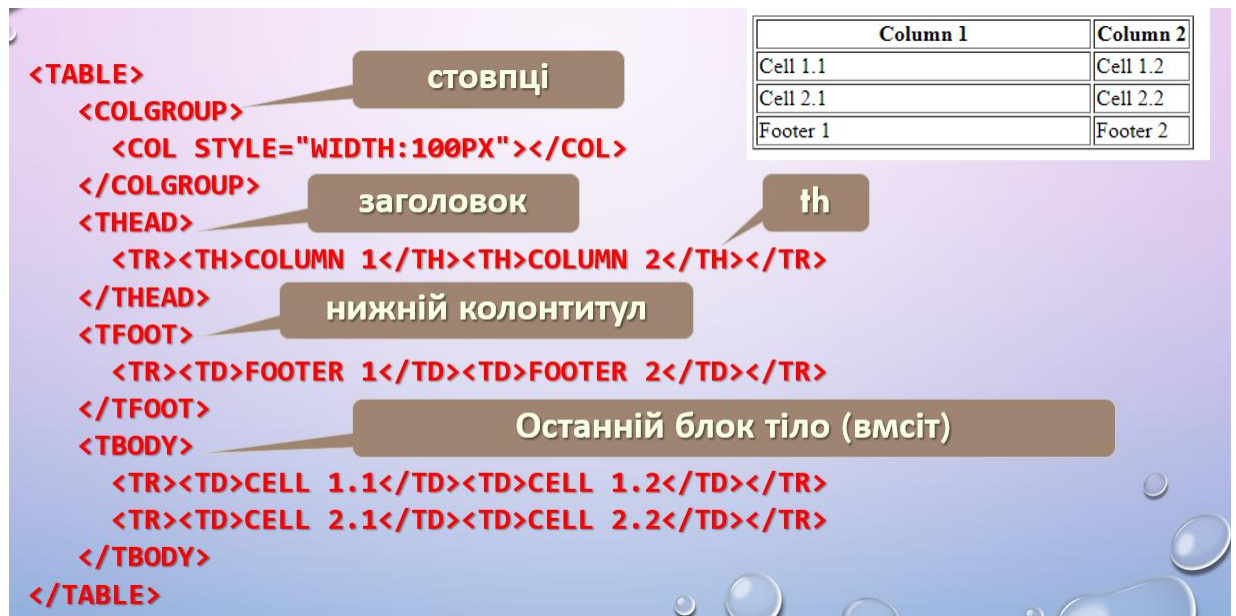
<thead> <tr> <th> ... </th> </tr> </thead>

<tfoot> <tr> <td> ... </td> </tr> </tfoot>

<tbody> <tr> <td> ... </td> </tr> </tbody>

</table>

Приклад:



### Об'єднання комірок

`colspan = "N"` - розтягує комірку на N стовпців вліво

#### Приклад:

```

<table cellpadding = "15" border = "1">
  <tr> <td colspan = "2"> 1 </td> </tr>
  <tr> <td> 2 </td> <td> 3 </td> </tr>
</table>
  
```

`rowspan = "N"` - розтягує комірку на N рядків вниз

#### Приклад:

```

<table cellpadding = "15" border = "1">
  <tr> <td rowspan = "2"> 1 </td> <td> 2 </td> </tr>
  <tr> <td> 3 </td> </tr>
</table>
  
```

### Ширина таблиці

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

```

<table border = "1">
  
```

```
<tr> <td> ширина таблиці не задана! </td> </tr>  
</table>
```

Максимальна ширина таблиці в такому випадку дорівнює ширині вікна. Якщо ж ширина задана атрибутом width, то браузер розставляє переноси слів в тексті комірок таким чином, щоб дотримати заданий розмір.

```
<table width = "100" border = "1">  
<tr> <td> якщо задати атрибут width, текст починає переноситися за  
словами </td> </tr>  
</table>
```

### Додавання заголовка таблиці

Заголовок таблиці можна створити за допомогою відомих вам тегів <h1> - <h6> Але оскільки ширина таблиці може відрізнятись від ширини вікна оглядача, вирівняти текстовий заголовок щодо таблиці може виявитися досить складно Тому для створення заголовків краще використовувати тег <CAPTION>, який створює заголовок безпосередньо в таблиці.

```
<table border = 10 width = 100%> <caption> назва таблиці </caption>
```

### Плаваючі фрейми

Плаваючий фрейм, або лінійний фрейм, з'являється як окреме, плаваюче вікно для виведення інших документів. Він отримав свою назву з того факту, що може з'являтися вбудованим в нормальний потік елементів сторінки або може розміщуватися 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 в цьому лінійному фреймі.

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

```
<iframe
```

```
src = "url"  
name = "framename"  
frameborder = "1 / 0"  
scrolling = "auto / yes / no"
```

*Виключені:*

```
width = "n / n%"  
height = "n / n%"  
align = "left / right"  
align = "top / middle / bottom"  
vspace = "n"  
hspace = "n"  
marginwidth = "n"  
marginheight = "n"
```

>

</Iframe>

Атрибут `src` визначає сторінку для початкового завантаження у фрейм. Атрибут `name` привласнює кадру ім'я в якості покажчика для посилань. Фрейм не обов'язково з'єднувати з посиланнями. Можна просто вивести всередині фрейму один зовнішній документ, і в цьому випадку треба визначити в атрибуті `src` фрейма без імені тільки URL.

За замовчуванням навколо фрейма виводяться кордони. Можна відключити кордони за допомогою атрибута `frameborder = "0"`. Якщо контент сторінки, що завантажується у фрейм, більше фрейму, то автоматично виводяться панелі прокрутки.

Можна відключити панелі прокрутки за допомогою атрибута `scrolling = "no"` або постійно виводити панелі прокрутки за допомогою `scrolling = "yes"`.

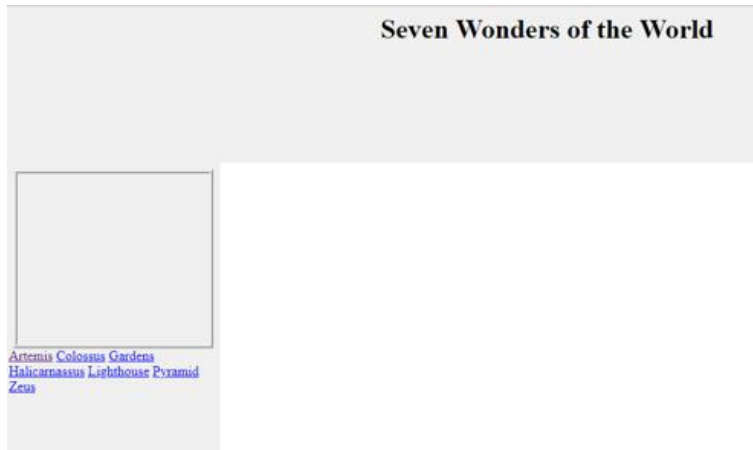
Решта атрибутів - width, height, align, vspace, hspace, marginwidth, і marginheight - краще ставити за допомогою таблиць стилів. Вони повинні вважатися виключеними атрибутами.

Відзначимо, що закриває тег `</iframe>` є обов'язковим, навіть якщо він нічого не замикає. На сторінці Web можна визначити будь-яку кількість плаваючих фреймів.

Наступний код використовується для виведення і активації плаваючого фрейма. Властивості таблиці стилів замінюють більшість атрибутів фрейму. Відзначимо, що в тезі `<iframe>` атрибут src не заданий. Тому фрейм відкривається 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>
<div>
<a href="Artemis.htm" target="TheFrame">Artemis</a>
<a href="Colossus.htm" target="TheFrame">Colossus</a>
<a href="Gardens.htm" target="TheFrame">Gardens</a>
<a href="Halicarnassus.htm" target="TheFrame">Halicarnassus</a>
<a href="Lighthouse.htm" target="TheFrame">Lighthouse</a>
<a href="Pyramid.htm" target="TheFrame">Pyramid</a>
<a href="Zeus.htm" target="TheFrame">Zeus</a>
</div>
```





#### **4. Порядок виконання лабораторної роботи**

- 1) ознайомитися з теоретичними відомостями;
- 2) виконати завдання
- 3) оформити звіт, який включатиме: титульний аркуш, завдання, теоретичні відомості, результати і висновки по роботі;
- 4) продемонструвати результат на комп'ютері і захистити лабораторну роботу

## Лабораторна робота №3

### 1. Тема

CSS. Внутрішні стилі. Стилі рівня документу. Зовнішні стилі. Оформлення тексту, поля, заповнення, межі. Застосування стилів для таблиць і списків

### 2. Завдання

1)

Створіть зовнішній CSS файл. Підключіть його до всіх сторінок. Використовуючи селектори (класи, ідентифікатори, унікальний ідентифікатор) налаштуйте стиль шрифту (розмір, колір, стиль, міжрядковий інтервал, вирівнювання) для заголовка (HN), для тіла (BODY), посилань, задайте для тега BODY фон властивістю background-color.

Застосуйте стиль рівня документу для перевизначення стилю для посилань.

Застосуйте внутрішній стиль до абзацу.

Використайте оголошення !important.

2)

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

Списки (див. попередні л/р).

Таблиці (див. попередні л/р).

Шрифти (див. попередні л/р).

3)

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

### 3. Теоретичні відомості

CSS (Cascading Style Sheets - каскадні таблиці стилів - технологія управління зовнішнім виглядом елементів (тегів) веб-сторінки. 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».

До переваг використання CSS відносяться:

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

#### Відносини між множинними вкладеними елементами

В html - документі елементи (теги) можуть перебувати в рамках інших елементів. Відносини між вкладеними елементами можуть бути батьківськими, дочірніми і братніми (в ряді літератури також зустрічається назва сестринські). Пояснимо ці та інші терміни, пов'язані з структурі html - документа:

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

Батьківський елемент - елемент, що містить в собі розглянутий елемент. У записі виду `<p> <strong> ... </ strong> </ p>`, елемент `<p>` є батьківським по відношенню до `<strong>`.

Пращур - елемент на кілька рівнів вище і містить в собі розглянутий елемент. Тобто в запису виду `<body> ... <p> <strong> ... </ strong> </ p> ... </ body>`, `<body>` є предком `strong`.

Дочірній елемент - елемент, що знаходиться усередині розглянутого документа. У записі виду `<p> <strong> ... </ strong> </ p>`, елемент `<strong>` є дочірнім по відношенню до `<p>`.

Нащадок - елемент, що знаходиться всередині елемента, що розглядається і знаходиться на кілька рівнів нижче. У записі виду `<body> ... <p> <strong> ... </ strong> </ p> ... </ body>`, `<strong>` є нащадком `<body>`.

Братський елемент - елемент, який має загальний батьківський елемент з даним. Тобто в запису `<p> <strong> ... </strong> ... <img ...> </p>`, елементи `<img>` і `<strong>` 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: значення3 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 прогалини. Можна додавати коментарі, укладаючи їх між `/ * і * /`.

### Селектори

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

В якості селектора можна використовувати:

- **Назву тега - тоді стиль застосується до всіх таким тегам.**

Приклад:

*A {font-size: 12pt; text-decoration: none}*

*TABLE {border: black solid 1px}*

Перший рядок цього CSS-коду задає всіх посиланнях 12-й розмір шрифту і прибирає підкреслення. На другій сходинці вказується, що у всіх таблиць межа буде чорного кольору, суцільний (solid) і шириною 1 піксель.

- **Кілька тегів через кому** - тоді стиль застосовується для всіх перерахованих тегів.

Приклад:

*H1, H2, H3, H4, H5, H6 {color: red} / \* робимо все заголовки червоними \* /*

- **Кілька тегів через пробіл:**

*TABLE A {font-size: 120%}*

Правило відноситься до всіх тегів А, вкладених в тег TABLE. Розмір шрифту збільшиться на 20% від базового.

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

Приклад:

*CSS*

*#supersize {font-size: 200%}*

*HTML*

*<a href="http://htmlbook.ru" id="supersize"> Довідник*

*HTML i CSS </a>*

Не можна вносити в документ кілька елементів з однаковим id!

- **Класи**

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.ім'я\_класа {...}*

Правила, зазначені після такого селектора, будуть діяти тільки на теги з атрибутом class = "ім'я\_класу":

<ТЕГ class = "ім'я\_класу"> ... </ ТЕГ>

Можна не вказувати ім'я тега, тоді правила будуть застосовуватися до всіх тегам з відповідним значенням атрибута class.

Приклад:

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

```
.class1 {text-decoration: underline; font-size: 80% }
```

```
A.class1 {text-decoration: none;}
```

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

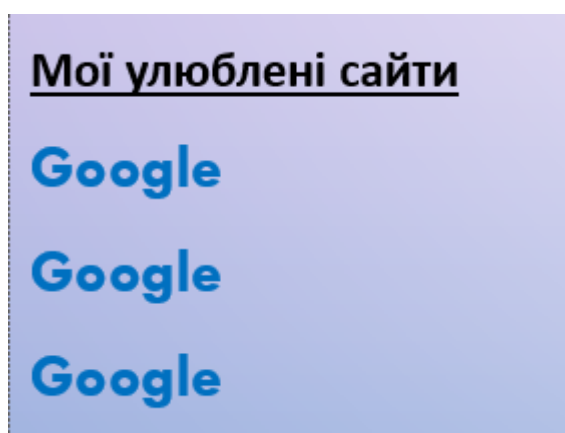
```
<H1 class = "class1"> Мої улюблені сайти </ h1>
```

```
<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>
```

У браузері буде відображатися:



Можна вказувати для одного елемента кілька класів через пробіл.

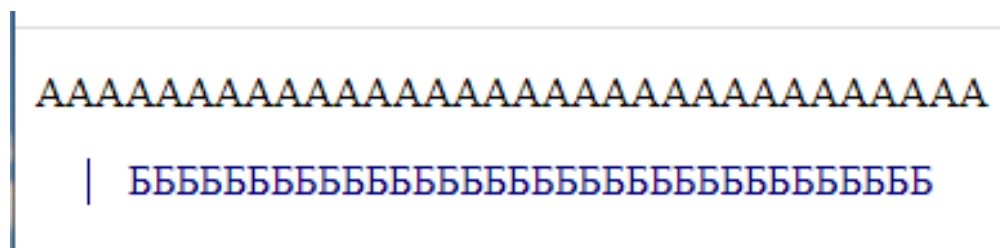
Приклад 2:

```

<html>
<head>
<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; /* Кордон ліворуч від тексту */
    padding-left: 15px; /* Відстань від лінії до тексту */
  }
</style>
</head>
<body>
<p>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</p>
<p class="cite">BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB</p>
</body>
</html>

```

Результат 2:

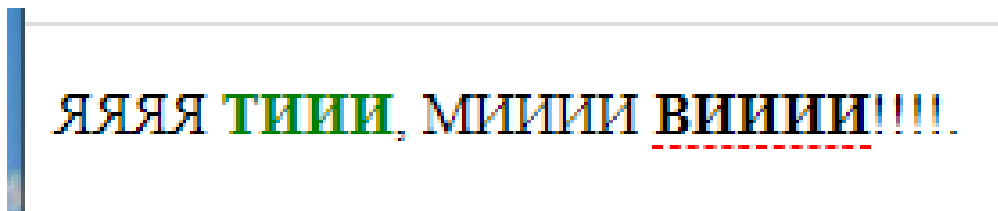


Перший абзац вирівняний по ширині з текстом чорного кольору (цей колір задається браузером за замовчуванням), а наступний, до якого застосовано клас з ім'ям cite - відображається синім кольором і з лінією зліва.

Можна, також, використовувати класи і без вказівки тега. Синтаксис в цьому випадку буде наступний.

```
<html>
<head>
  <meta charset="utf-8">
  <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">ТІІІІ</span>, МІІІІІІ
    <b class="term">ВІІІІІІ</b>!!!!.
  </p>
</body>
</html>
```

Результат:



### Універсальний селектор

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

Символ \* - універсальний селектор. правила застосуються до всіх елементів документа.



Синтаксис \* {Опис правил стилю}

У деяких випадках вказувати універсальний селектор не обов'язково. Так, 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>
```

Приклад:

```
* {margin: 0; } прибере відступи у всіх елементів на сторінці.
```

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

### **Зовнішні стилі.**

Зберігаються в окремому файлі .css. підключаються тегом

```
<Link rel = "stylesheet" type = "text / css" href = "адреса_стиля">
```

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 кожен сторінку

окремо. Якщо на сайті кілька десятків або сотень сторінок це стає дуже трудомістким завданням.

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

Реалізуємо наш попередній приклад.

Створимо файл style.css:

```
.class1 {text-decoration: underline; font-size: 80%}
```

```
A.class1 {text-decoration: none;}
```

Тепер 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>
```



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

Зверніть увагу: за допомогою CSS можна відключити у посилань підкреслення. Засобами HTML цього зробити неможливо. CSS значно розширює можливості оформлення сторінки.

Другий важливий момент: використання CSS дозволяє розділити оформлення та вміст документа. У нашому прикладі правила оформлення містяться в файлі style.css, а зміст - в links.html.

Такий поділ істотно спрощує редагування сайту в подальшому. Рекомендується для оформлення використовувати тільки засоби CSS, відмовитися від використання таких тегів, як <font>, <s>, <u>, <center>, атрибутів align, border, color, height, width і т.д.

## Стили рівня документа

Застосовуються до всього документа, записуються всередині тега <style> ... </ style>, який вкладається в тег <head> ... </ head> в документі HTML.

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

Додамо в наш приклад тег <style>:

```
<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>
```

## My favorite Sites

Google 1  
Google 2  
Google 3

### Внутрішні стилі

Використовуються, коли потрібно вказати стилі конкретного єдиний елемент. Внутрішній стиль записується в атрибуті style і застосовується тільки до вмісту цього тега. Внутрішній стиль має більш високий пріоритет, ніж зовнішні стилі і стиль рівня документа. Переважно не використовувати такий спосіб завдання стилю, тому що він не відповідає принципу поділу змісту і оформлення.

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

### 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 ніяк не з внутрішніми стилями. Загальний синтаксис наступний.

*@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. У наступному прикладі

показано, як можна імпортувати стиль із зовнішнього файлу в таблицю глобальних стилів.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8">
    <title>Импорт</title>
    <style>
      @import url("style/header.css");
      H1 {
        font-size: 120%;
        font-family: Arial, Helvetica, sans-serif;
        color: green;
      }
    </style>
  </head>
  <body>
    <h1>Заголовок 1</h1>
    <h2>Заголовок 2</h2>
  </body>
</html>
```

В даному прикладі показано підключення файлу header.css, який розташований в папці style.

### **Оголошення! Important**

Якщо ви зіткнулися з екстремим випадком і вам необхідно підвищити значимість якої-небудь властивості, можна додати до нього оголошення! Important:

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

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

Також! Important перекриває внутрішні стилі. Занадто часте застосування! Important не вітається багатьма розробниками. 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 зовнішнього та внутрішнього таблиці стилів, від механізму наслідування, від батьківських елементів, від класів і ID, від селектора тега, від атрибута style і т.д. Оскільки в цих випадках часто відбувається конфлікт стилів, була створена система пріоритетів: в кінцевому підсумку застосовується той стиль, який виходить від джерела з більш високим пріоритетом.

Які джерела є більш значущими, а які - менше? Розібратися в цьому допоможе ця таблиця, де вказана вага (значимість) кожного селектора. Чим більше вага, тим вище пріоритет:

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

Коли селектор складається з декількох інших селекторів, необхідно порахувати їх загальна вага. Ось як обчислюється пріоритет: за кожен селектор додається 1 в відповідному полі. В інших комірках стоять нулі. Щоб отримати загальну вагу, необхідно «склеїти» все числа в комірках.

Селектор	ID	Класс	Тег	Общий вес
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

Якщо трапилося так, що два селектора мають однакову вагу, то пріоритет віддається тому стилю, який знаходиться нижче в коді. Якщо для одного елемента визначити тип і в зовнішній, і у внутрішній таблицях, то пріоритет віддається стилю в тій таблиці, яка знаходиться нижче в коді.

Приклад: у внутрішній таблиці стилів заданий червоний колір для тегів <p>, а у зовнішній - зелений колір для цих же тегів. У HTML-документі ви насамперед 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.

Наприклад:

- стиль, вказаний в атрибуті style, перекриває стиль, вказаний в тезі <style> або зовнішньому файлі CSS:

```
<html>
<head>
<style>
A {color: red; text-decoration: none}
</style>
</head>
<body>
<a href=http://intuit.ru style="color:
green">INTUIT</a>
</body>
</html>
```

У браузері посилання буде неподчеркнутой, зеленого кольору.

- селектор ID (#) має більший пріоритет, ніж селектор класу (.), А той, у свою чергу, - більший, ніж звичайний селектор тега:

```
<html>
<head>
<style>
A {color: red; text-decoration: none; font-size: 120%}
.links {color: blue; text-decoration: underline}

```

```
#greenlink {color: green}
</style>
</head>
<body>
<a href="http://INT.ua" class="links"
id="greenlink">INT.ua</a>
</body>
</html>
```

У браузері посилання буде зеленою і підкресленою, 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

#### **Розміри**

Розміри в CSS можна задавати в різних одиницях виміру:

- `em` - поточна висота шрифту
- `pt` - пункти (друкарський одиниця виміру шрифту)
- `px` - піксель
- `%` - відсоток

Набагато рідше використовується вказівка розмірів в міліметрах (`mm`), сантиметрах (`cm`) і дюймах (`in`).

Одиниця виміру записується відразу за значенням без пробілу:

```
TABLE {font-size: 12pt}
```

#### **Колір**



В CSS колір задається як і в HTML - шістнадцятирічними цифрами: по 2 на кожен базовий колір (червоний, зелений, синій). Також можна використовувати стандартні назви квітів англійською).

Паприклад:

```
A.content {color: black}
```

```
A.menu {color: # 3300AA}
```

Допускається скорочувати шістнадцядкове представлення до 3 цифр: запис # 3300AA можна замінити на # 30A.

Рідше використовується конструкція `rgb (...)`, яка дозволяє задавати червону, зелену і синю компоненти в десятковому або відсотком вигляді:

```
A.content {color: rgb (0%, 0%, 0%)}
```

```
A.menu {color: rgb (51,0,170)}
```

## **URL**

URL задаються конструкцією `url (...)`. Наприклад, наступний CSS-код додає фонове зображення для сторінки:

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

## **Шрифти**

Шрифт - набір накреслень букв і знаків. У комп'ютері шрифт являє собою файл, в якому описано, як повинні відображатися на моніторі або принтері різні символи: літери, цифри, знаки пунктуації та ін.

Часто шрифти містять тільки накреслення для латинського алфавіту і не мають, наприклад, підтримки кирилиці. Існують Unicode-шрифти, які містять символи для всіх мов. Основні формати файлів шрифтів: TTF - TrueType і його розширення OTF - OpenType.

### Типи шрифтів:

**serif** - шрифти із зарубками (антіквенніе), наприклад: Times New Roman, Georgia.

**sans-serif** - рубані шрифти (шрифти без зарубок або гротески), типові представники - Arial, Impact, Tahoma, Verdana;

**cursive** - курсивні шрифти: Comic Sans MS;

**fantasy** - декоративні шрифти, наприклад: Curlz MT.

**monospace** - моноширинних шрифти, ширина кожного символу однакова. Приклади: Courier New, Lucida Console.

Зарубками називають елементи на кінцях штрихів букв. Порівняємо букву шрифту Times New Roman і букву шрифту Arial.



Пунктирними лініями обведені зарубки.

Використання шрифтів із зарубками полегшує читання тексту з паперу, тому такі шрифти зазвичай використовують для набору основного тексту в книгах. Для web-сайтів основний текст частіше набирають шрифтом без зарубок: Arial, Tahoma, Trebuchet MS, Verdana.

### **Текст**

CSS дозволяє управляти властивостями шрифту і тексту.

**font-family** - задає накреслення шрифту. 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

Приклад:

*font-family: Georgia, 'Times New Roman', serif*

Якщо на комп'ютері користувача встановлено шрифт Georgia, то буде використовуватися він, якщо немає - то Times New Roman. Якщо ж і Times New Roman відсутній, то браузер буде використовувати шрифт із зарубками, який встановлений на комп'ютері.

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

Приклад:

*font-size: 12pt*

або

*font-size: 150%*

**font-style** - задає зображення тексту: **normal** (звичайний), **italic** (курсивний) або **oblique** (похилий). Курсив є спеціальною зміненою версією шрифту, що імітує рукописний текст з нахилом вправо. Похиле накреслення виходить зі звичайного нахилом букв.

Зазвичай браузер не може відобразити похилий нарис і замінює його курсивним.

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

**color** - задає колір тексту (див. пункт «Кольори» цієї лекції).

Наприклад, задамо червоний колір для всіх заголовків:

*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 CSS.

Приклад:

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

У програмуванні прийнято відокремлювати цілу частину числа від дробової точкою, як в англійській мові.

**text-decoration** - задає оформлення тексту.

Варіанти: **line-through** (перекреслений), **overline** (лінія над текстом), **underline** (підкреслення), **none** (відключення ефектів).

Наприклад, відключимо підкреслення у посилань:

*A {text-decoration: none}*

**text-align** - вирівнювання тексту в блоці: left (по лівому краю), center (по центру), right (по правому краю) або justify (по ширині).

Приклад:

*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.

Приклад:

*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, наприклад:

*H1 {font: bold 14pt / 1.5 sans-serif}*

### 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).

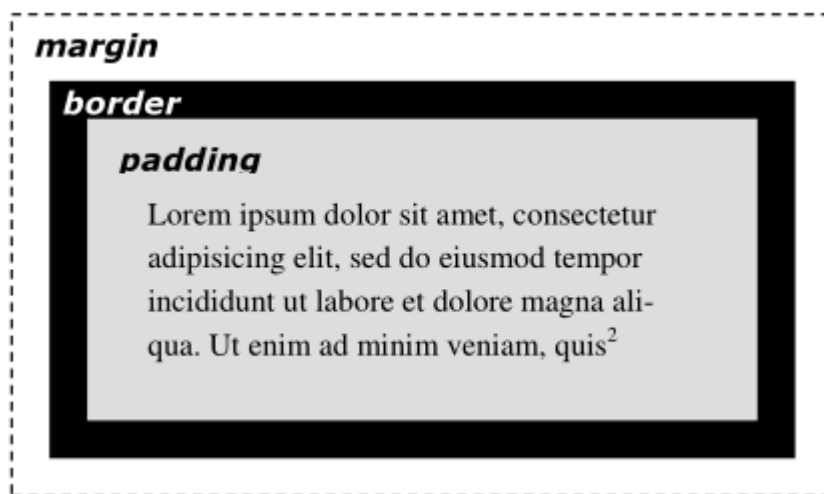


Рисунок Бокс (box) елемента.

Ширина полів і заповнення 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.

Приклад:

*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 буде задавати ширину верхнього і нижнього поля, а друге - лівого і правого. Якщо вказати три значення, то перше буде присвоюватися верхньому полю, друге - лівому і правому, а третє - нижньому. Нарешті, при вказівці чотирьох значень, вони по черзі будуть вказувати верхнє, праве, нижнє і лівє поля.

**padding-top, padding-right, padding-bottom, padding-left** - встановлюють ширину заповнення 1 зверху, праворуч, знизу і 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 і заповнення.

Для кордонів можна задати товщину, колір і стиль:

**border-width** - товщина кордону;

**border-color** - колір кордону (за замовчуванням - чорний);

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

На рис. представлені всі види кордонів, border-width встановлений в 5 пікселів.

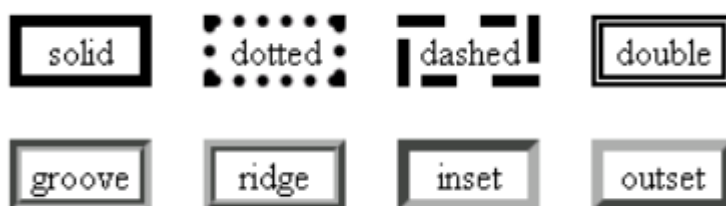


Рисунок Види кордонів.

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

Наприклад:

```
<P style = "border: solid 1px green"> Текст </P>
```

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

Приклад:

```
<html>
<head>
<title>приклад</title>
<style>
H3 {
border-top: 2px dashed black;
border-bottom: 2px dashed black;
border-left: 0;
border-right: 0;
}
</style>
<body>
<h3>Header</h3>
</body>
</html>
```

У браузері:



Рисунок властивості меж вказані окремо

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 і стилю кожної кордону, наприклад: **border-top-width**, **border-right-color**, **border-bottom-style** і ін.

У попередньому прикладі межа розтягнулася по всій ширині вікна браузера. Це сталося через те, що багато HTML елементи за замовчуванням займають 100% ширини елемента, в які вони вкладені. Для визначення розміру в CSS існують

властивості **width** і **height**. Найчастіше ширину і висоту задають у пікселях (px) або у відсотках (%) від ширини батьківського елемента.

Розглянемо приклад:

```
<html>
<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">11111111</p>
<p id="text2">222222222</p>
<p id="text3">3333333333333</p>
</body>
</html>
```

Результат:

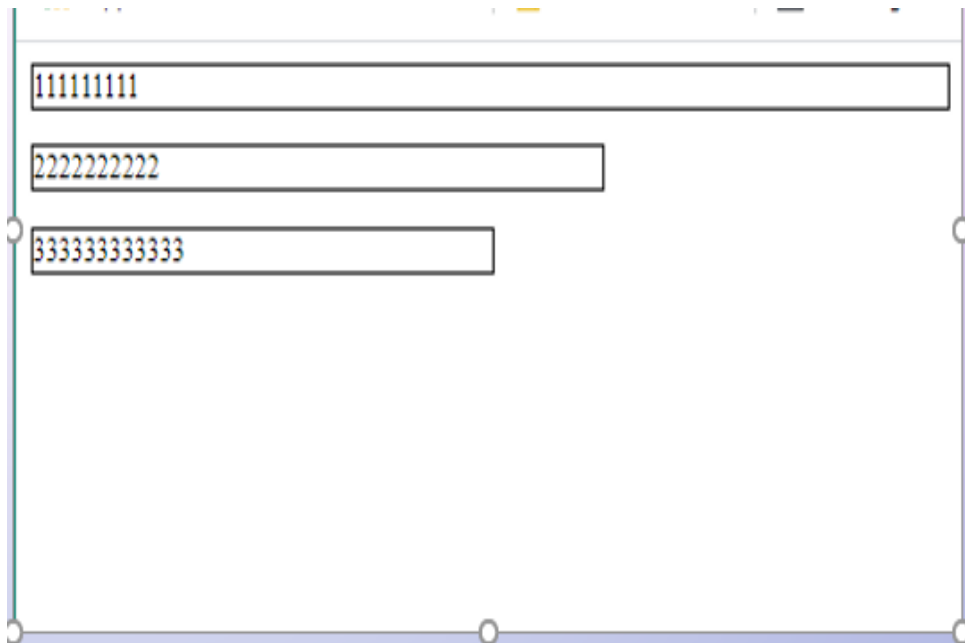


Рисунок Відображення прикладу в браузері



Розміри першого абзацу не вказані в стилі, ширина першого абзацу задана абсолютно в пікселях, а третього - щодо ширини вікна.

Якщо ширина або висота не задані, вони автоматично обчислюються браузером, виходячи з розмірів вмісту: для першого абзацу браузер встановив ширину, рівну ширині вікна (100%). У другому і третьому абзаці ширина задана, але не задана висота, тому браузер сам підібрав її так, щоб весь текст помістився в елемент.

Тепер, якщо користувач змінить розмір вікна, пропорційно зміниться ширина тих елементів, де вона була задана в процентах.

Зменшимо розмір вікна браузера. У першого і третього абзацу зменшиться ширина, а висота збільшиться, щоб вмістити весь текст. Розміри другого абзацу залишаться незмінними, з'являться смуги прокручування.

Поведінка браузерів різниться, якщо для елемента задані і ширина, і висота, а вміст не вміщається в ці розміри. Internet Explorer збільшить розміри елемента. Браузери, повністю підтримують стандарт CSS, такі як Firefox, відобразять вміст поверх блоку.

Результат:

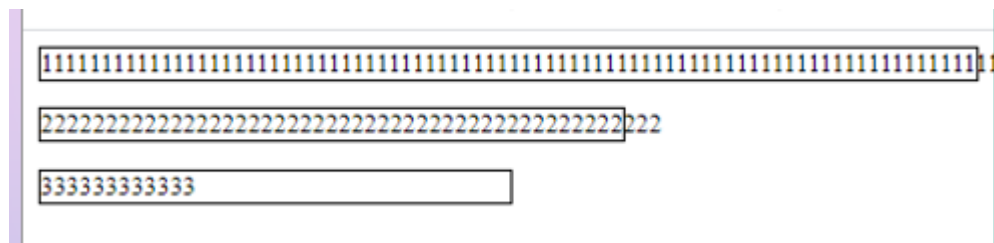


Рисунок Відображення прикладу в браузері при зменшенні ширини вікна

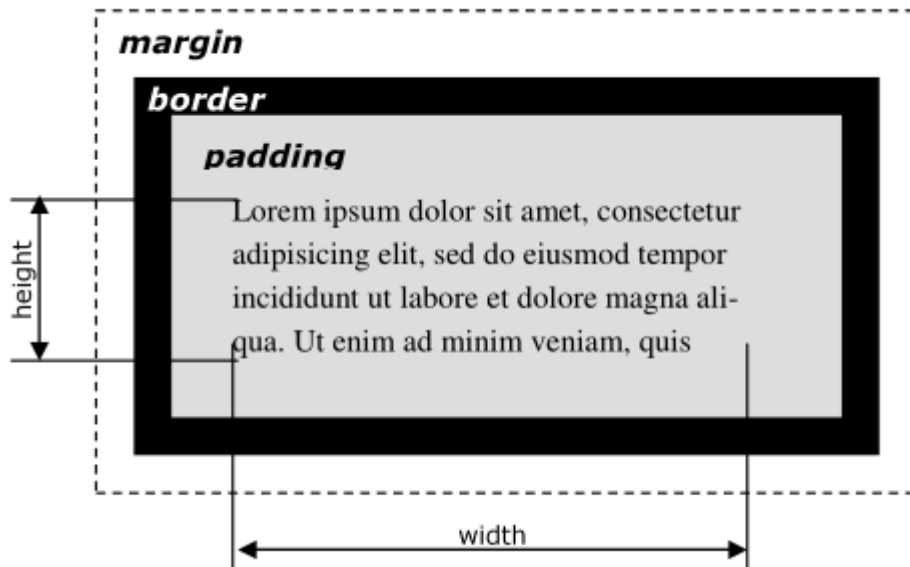
Можна ставити мінімальні і максимальні розміри властивостями **min-width**, **min-height** і **max-width**, **max-height**. Ці властивості не підтримує браузер Internet Explorer версії 6 і нижче.

Загальні розміри елемента складаються так:

*Ширина* = *width* + *padding* + *border* + *margin*

*Висота* = *height* + *padding* + *border* + *margin*

Тобто **width** і **height** задають тільки розміри вмісту, не включаючи поля, заповнення та кордон! Див. рис. нижче.



### CSS-властивості: фон, оформлення таблиць

#### **Фон**

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

**background-color** - встановлює колір фону.

Приклад:

```
TD.head {background-color: #ffff00}
```

**background-image** - встановлює в якості фону зображення:

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

**background-attachment** - задає поведінку фонового зображення при прокручуванні. За замовчуванням задається значення **scroll** - фон прокручується разом з вмістом. Значення **fixed** робить фон нерухомим.

**background-position** - початкове положення фонового зображення по горизонталі (left, center, right) і вертикалі (top, center, bottom). Замість ключових слів можна вказувати відстань у пікселях або відсотках.

**background-repeat** - вказує, в якому напрямку повинно розмножуватися фонове зображення:

**repeat** - по горизонталі і вертикалі (за замовчуванням);

**repeat-x** - тільки по горизонталі;

**repeat-y** - тільки по вертикалі;

**no-repeat** - відключити повторення.

Приклад:

Використовуючи зображення одного вагона, складемо в тлі поїзд.



Рисунок Фонове зображення.

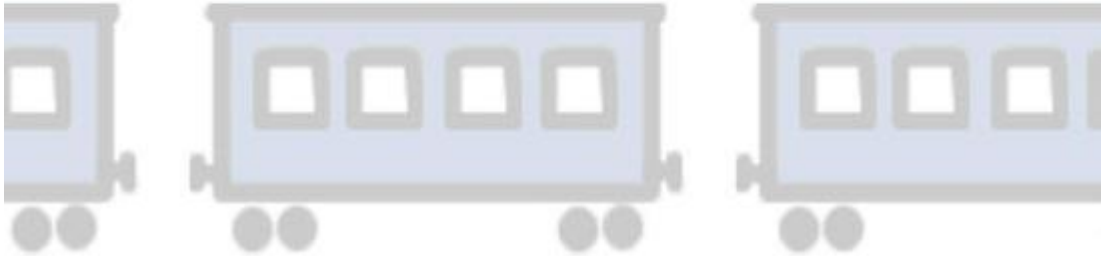
CSS код:

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

У браузері:

---

## Header



## Таблиці

Властивості CSS можуть застосовуватися до таблиць, їх рядках і комірках для завдання властивостей тексту та шрифту, управління фоном, полями, межами, розмірами і т.п.

Створимо таблицю і застосуємо до неї CSS-стилі. У таблицю внесемо дані про популярність різних браузерів. 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>Safari</th>
<th>Opera</th>
</tr>
<tr>
<td>2010</td>
<td>61.43%</td>
```

```

<td>24.40%</td>
<td>4.55%</td>
<td>2.37%</td>
</tr>
<tr>
<td>2009</td>
<td>69.13%</td>
<td>22.67%</td>
<td>3.58%</td>
<td>2.18%</td>
</tr>
<tr>
<td>2008</td>
<td>77.83%</td>
<td>16.86%</td>
<td>2.65%</td>
<td>1.84%</td>
</tr>
<tr>
<td>2007</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 буде виглядати так:

---

<b>Yaer/Browser</b>	<b>IE</b>	<b>Firefox</b>	<b>Safari</b>	<b>Opera</b>
2010	61.43%	24.40%	4.55%	2.37%
2009	69.13%	22.67%	3.58%	2.18%
2008	77.83%	16.86%	2.65%	1.84%
2007	79.38%	14.35%	4.70%	0.50%

Рисунок Відображення таблиці за замовчуванням

За замовчуванням вміст заголовків комірок відображається жирним шрифтом з вирівнюванням по центру.

Додамо в тег <head> ... </ head> тег <style> ... </ style>, а до тегу <table> ... </ table> атрибут id = "browser\_stats". Запишемо CSS-правила для таблиці. Для заголовків комірок встановимо сірий фон і відступ вмісту від кордонів (padding) в половину висоти рядка, для комірок з даними - вирівнювання по правому краю і padding три десятих від висоти рядка. Навколо таблиці задамо подвійну рамку, а для комірок - звичайну одинарну.

Код:

```
<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>
```

У браузері:

Year/Browser	IE	Firefox	Safari	Opera
2010	61.43%	24.40%	4.55%	2.37%
2009	69.13%	22.67%	3.58%	2.18%
2008	77.83%	16.86%	2.65%	1.84%
2007	79.38%	14.35%	4.70%	0.50%

Рисунок Відображення таблиці з заданими CSS-стилями

Видно істотний недолік: у кожній комірці з'явилася власна рамка. Щоб цього не відбувалося, необхідно вказати в правилах для таблиці властивість `border-collapse` із значенням `collapse`.

Результат:

Year/Browser	IE	Firefox	Safari	Opera
2010	61.43%	24.40%	4.55%	2.37%
2009	69.13%	22.67%	3.58%	2.18%
2008	77.83%	16.86%	2.65%	1.84%
2007	79.38%	14.35%	4.70%	0.50%

Рисунок Ефект злиття кордонів сусідніх комірок

Тепер застосуємо до тієї ж таблиці інше форматування. Розділимо таблицю двома лініями на 3 частини: назви браузерів, роки і процентні дані. Назви браузерів і процентні частки вирівнюємо по центру, роки - по правому краю. Задамо однакову ширину для стовпців з інформацією по браузерам.

Yaer/Browser	IE	Firefox	Safari	Opera
2010	61.43%	24.40%	4.55%	2.37%
2009	69.13%	22.67%	3.58%	2.18%
2008	77.83%	16.86%	2.65%	1.84%
2007	79.38%	14.35%	4.70%	0.50%

Рисунок Оформлення таблиці з двома розділовими лініями

Щоб застосувати правила CSS до лівої колонки (роки), нам доведеться додати новий клас `lc` і прописати атрибут `class = "lc"` в усі комірки лівої колонки.

Горизонтальна лінія створюється шляхом зазначення властивості `border-bottom` для комірок TH, вертикальна - `border-left` для комірок класу `lc`.

Код-сторінки:

```
<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 id="browser_stats">
<tr>
<th class="lc">Year/Browser
</th>
<th>IE</th>
<th>Firefox</th>
<th>Safari</th>
<th>Opera</th>
</tr>

```



```

<tr>
<td class="lc">2010</td>
<td>61.43%</td>
<td>24.40%</td>
<td>4.55%</td>
<td>2.37%</td>
</tr>
<tr>
<td class="lc">2009</td>
<td>69.13%</td>
<td>22.67%</td>
<td>3.58%</td>
<td>2.18%</td> </tr>
<tr>
<td class="lc">2008</td>
<td>77.83%</td>
<td>16.86%</td>
<td>2.65%</td>
<td>1.84%</td> </tr>
<tr>
<td class="lc">2007</td>
<td>79.38%</td>
<td>14.35%</td>
<td>4.70%</td>
<td>0.50%</td>
</tr> </table> </body> </html>

```

#### 4. Порядок виконання лабораторної роботи

1) ознайомитися з теоретичними відомостями;

2) виконати завдання

3) оформити звіт, який включатиме: титульний аркуш, завдання, теоретичні відомості, результати і висновки по роботі;

4) продемонструвати результат на комп'ютері і захистити лабораторну роботу

## Лабораторна робота №4

### 1. Тема

CSS. Контекстні селектори. Сусідні селектори. Дочірні селектори. Блочні елементи.

### 2. Завдання

1)

Застосувати контекстні, сусідні, дочірні селектори в рамках тематики

2)

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

### 3. Теоретичні відомості

#### Сусідні селектори

`<p>Llllll<b>Ppppppp</b> dddddd, <i>cccccc</i>aaaaaa  
<tt>eeeeee</tt>.</p>`

Сусідніми тут є теги `<b>` і `<i>`, а також `<i>` і `<tt>`. При цьому `<b>` і `<tt>` до сусідніх елементів не належать через те, що між ними розташований контейнер `<i>`.

Для управління стилем сусідніх елементів використовується символ плюса (+), який встановлюється між двома селекторами. Загальний синтаксис наступний.

*Селектор 1 + селектор 2 {Опис правил стилю}*

Прогалини 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 слід відразу після нього.

Приклад:

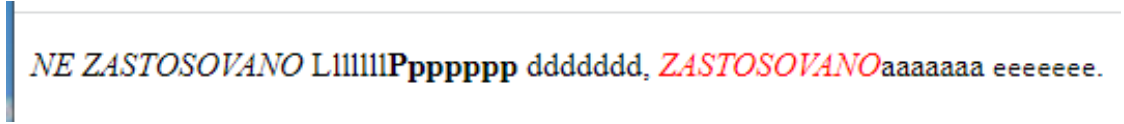
```
<style>
B + I {
color: red; /*Червоний колір тексту */
}
</style>
```

```

</head>
<body>
  <p><i>NE      ZASTOSOVANO</i>      Llllll<b>Ppppppp</b>      ddddddd,
<i>ZASTOSOVANO</i>aaaaaaa <tt>eeeeeee</tt>.</p>
</body>

```

Результат:



В даному прикладі відбувається зміна кольору тексту для вмісту контейнера `<i>`, коли він розташовується відразу після контейнера `<b>`. У першому абзаці така ситуація реалізована, тому слово «consectetur» в браузері відображається червоним кольором. У другому абзаці, хоча і присутній тег `<i>`, але по сусідству ніякого тега `<b>` немає, так що стиль до цього контейнеру не застосовується.

### Контекстні селектори

При створенні веб-сторінки 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 `<b>` тільки коли він розташовується всередині контейнера `<p>`. Таким чином можна одночасно встановити стиль для окремого тега, а також для тега, який знаходиться всередині іншого.

Контекстний селектор складається з простих селектор розділених пропуском. 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>*

Використання контекстних селекторів продемонстровано в наступному прикладі.

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:



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; /* Темно-синій колір посилань */
}
</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:

```



Ukranian | English | German

Link

В даному прикладі використовується два типи посилань. Перше посилання, стиль якої задається за допомогою селектора A, буде діяти на всій сторінці, а стиль другої посилання (.menu A) застосовується тільки до посилань всередині елемента з класом menu.

При такому підході легко керувати стилем однакових елементів, на зразок зображень і посилань, оформлення яких має відрізнятися в різних областях веб-сторінки.

### Дочірні селектори

Дочірнім називається елемент, який безпосередньо розташовується всередині батьківського елемента.

Приклад:

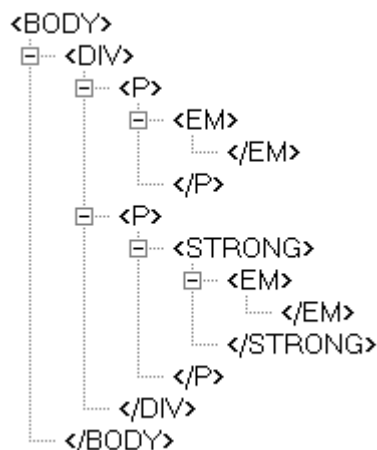
```
<!DOCTYPE HTML>
```

```

<html>
  <head>
    <meta charset="windows-1251">
  <title>LABEL</title>
  </head>
  <body>
    <div class="main">
      <p><em>11111111111111</em></p>
      <p><strong><em>222222222222</em></strong>,
3333333333333333</p>
    </div>
  </body>
</html>

```

В даному прикладі застосовується кілька контейнерів, які в коді розташовуються один в іншому. Найбільш наочно це видно на дереві елементів, так називається структура відносин тегів документа між собою.



Дерево елементів для прикладу

На малюнку в зручному вигляді представлена вкладеність елементів і їх ієрархія. Тут дочірнім елементом по відношенню до тегу `<div>` виступає тег `<p>`. Разом з тим тег `<strong>` не є дочірнім для тега `<div>`, оскільки він розташований в контейнері `<p>`.

Повернемося тепер до селекторам. Дочірнім селектором вважається такою, що в дереві елементів знаходиться прямо всередині батьківського елементу. Синтаксис застосування таких селекторів наступний.

*Селектор 1 > селектор 2 {Опис правил стилю}*

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

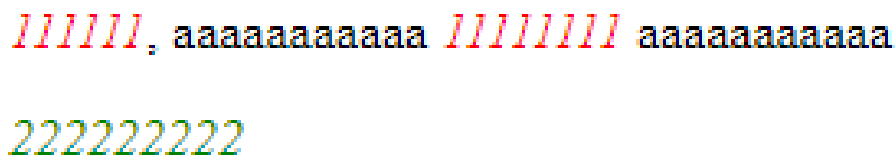
Якщо знову звернутися до нашого прикладу, то стиль виду `P > EM {color: red}` буде встановлений для першого абзацу документа, оскільки тег `<em>` знаходиться всередині контейнера `<p>`, та не дасть ніякого результату для другого абзацу. А все через те, що тег `<em>` в другому абзаці розташований в контейнері `<strong>`, тому порушується умова вкладеності.

За своєю логікою дочірні селектори схожі на селектори контекстні. Різниця між ними така. Стиль до дочірнього селектору застосовується тільки в тому випадку, коли він є прямим нащадком, іншими словами, безпосередньо розташовується всередині батьківського елемента. Для контекстного селектора ж допустимо будь-який рівень вкладеності. Щоб стало зрозуміло, про що йде мова, розберемо наступний код:

```
<head>
  <meta charset="windows-1251">
  <title>Дочірні селектори</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>, aaaaaaaaaa <i>11111111</i>
    aaaaaaaaaa</p>
    <i>22222222</i>
  </div>
</body>
```

Результат:





*lllllll, aaaaaaaaaa llllllll aaaaaaaaaa*  
*222222222*

Колір тексту, заданий за допомогою дочірнього селектора

На тег `<i>` в прикладі діють одночасно два правила: контекстний селектор (тег `<i>` розташований всередині `<div>`) і дочірній селектор (тег `<i>` є дочірнім по відношенню до `<p>`). При цьому правила є рівносильними, оскільки всі умови для них виконуються і не суперечать один одному. У подібних випадках застосовується стиль, який розташований в коді нижче, тому курсивний текст відображається червоним кольором. Варто поміняти правила місцями і поставити `DIV I` нижче, як колір тексту зміниться з червоного на зелений.

Зауважимо, що 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, а також простий і наочний код.

Найзручніше застосовувати зазначені селектори для елементів, які мають ієрархічною структурою - сюди відносяться, наприклад, таблиці і різні списки.

#### 4. Порядок виконання лабораторної роботи

- 1) ознайомитися з теоретичними відомостями;
- 2) виконати завдання
- 3) оформити звіт, який включатиме: титульний аркуш, завдання, теоретичні відомості, результати і висновки по роботі;
- 4) продемонструвати результат на комп'ютері і захистити лабораторну роботу

## Лабораторна робота №5

### 1. Тема

Блочні елементи. Рядкові елементи. Позиціонування. Псевдокласи. Псевдоелементи.

### 2. Завдання

1)

Застосувати блочні елементи `<div> ... </div>`, рядкові елементи `<span> ... </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**).

### 3. Теоретичні відомості

#### Теги DIV і SPAN

До сих пір ми застосовували стилі CSS до тегам, які вже мають заздалегідь задану функцію: таблиць, заголовків, параграфів і т.д. Але іноді потрібно застосувати стилі до фрагменту вмісту, не віднесеного до окремих тегів. Наприклад, виділити фоном кілька слів в тексті.

Теги `<div> ... </div>` і `<span> ... </span>` використовуються там, де не підходить жоден інший тег. Самі по собі вони не визначають ніякого форматування, але зручні для прив'язки до них стилів. При цьому **DIV** є блоковим елементом, а **SPAN** - рядковим.

Основна відмінність між блоковими і малими елементами полягає в наступному: рядкові елементи йдуть один за одним в рядку тексту, а блочні -

розташовуються один за іншим. До рядковим елементів відносяться такі теги, як `<a>`, `<img>`, `<input>`, `<select>`, `<span>`, `<sub>`, `<sup>` і ін.

До блоковим: `<div>`, `<form>`, `<h1>` ... `<h6>`, `<ol>`, `<p>`, `<table>`, `<ul>` і деякі інші. Розглянь відмінність на прикладі. Для тега `<span>` вказано стильове правило, яке задає колір фону.

HTML-код:

```
<span style="background-color: #eeeeee">Riadlovi elementi</span>
<sub>roztashovuiutsia v riadku</sub>

<sup>I idut odin za odnim</sup>
```

У браузері:



Рисунок - Поведінка рядкових елементів.

Розглянемо приклад для блокових тегів:

```
<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:

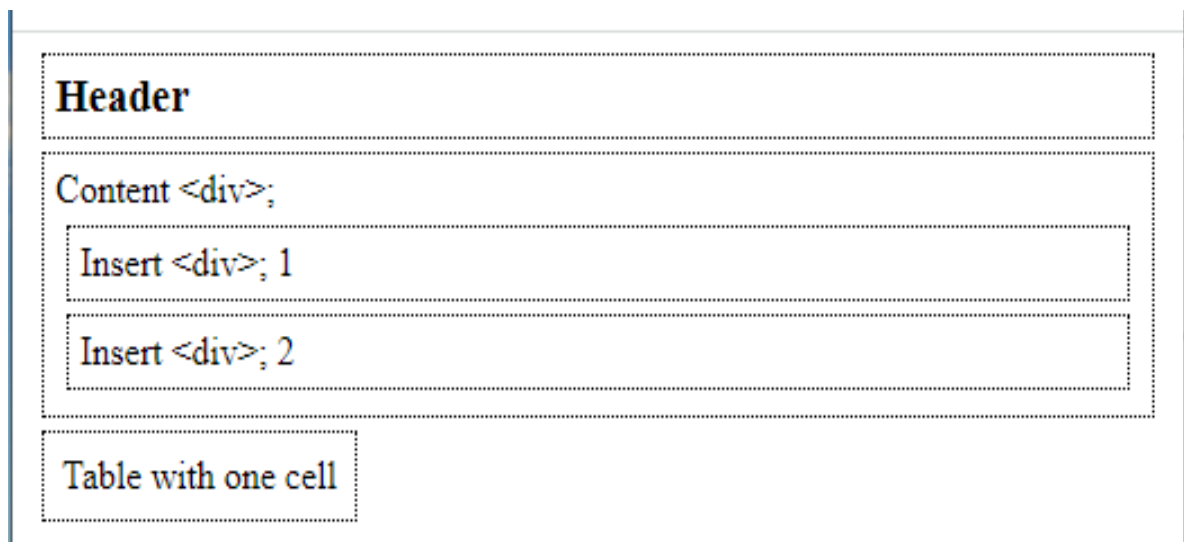


Рисунок Поведінка блокових елементів.

Блочні елементи розташовуються один під одним, багато займають всю можливу ширину. Блочні елементи можуть включати в себе малі і інші блочні. Але малі елементи не можуть містити блочні!

Ще однією відмінністю є те, що для малих елементів не працюють такі властивості, як **margin-top**, **margin-bottom**, **padding-top** і **padding-bottom**. Винятком є теги **<img>**, **<input>**, **<textarea>** і **<select>** - для них можна задавати відступи **padding-top** і **padding-bottom**.

### Псевдокласи

Ми розглядали раніше способи прив'язки правил оформлення CSS до елементів документа HTML: за назвою тега, по імені класу, по ID і т.п.

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 псевдокласу:

**:link** - посилання, які не відвідувалися користувачем;

**:visited** - відвідані посилання;

**:active** - активна (натиснута) посилання;

**:hover** - посилання, на яку наведений курсор.

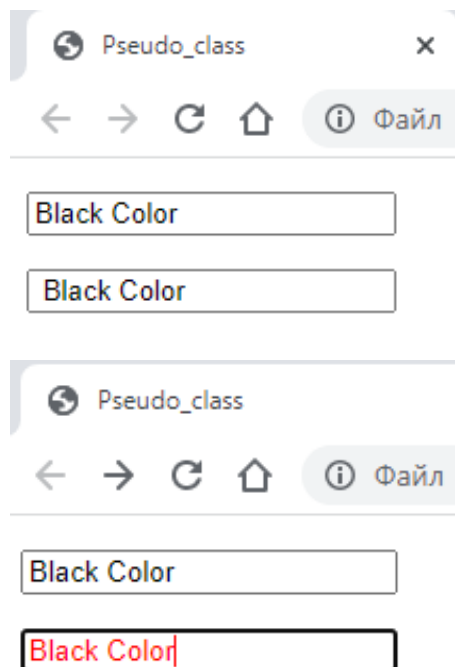
**:focus** - застосовується до елемента при отриманні ним фокусу.

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

Приклад:

```
<!DOCTYPE HTML>
<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:



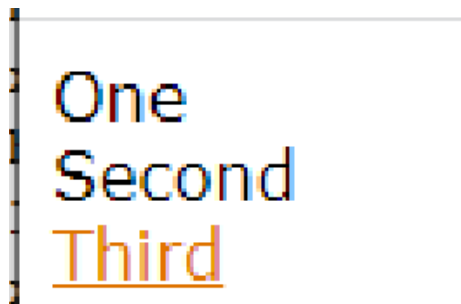
Приклад 2:

```

<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:



### Псевдокласи, що мають 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:

```

<html>
<head>
<meta charset="utf-8">
<title>Pseudo_class</title>
<style type="text/css">
B:first-child {

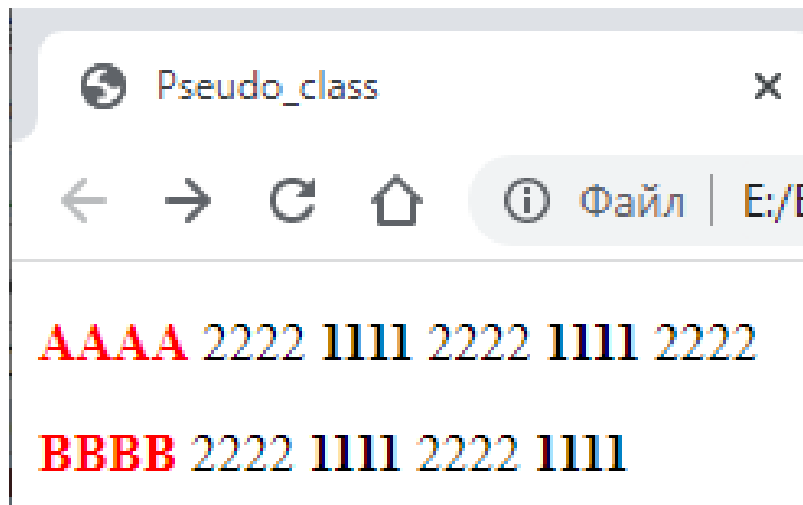
```

```

    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:



В даному прикладі псевдоклас **:first-child** додається до селектора **B** і встановлює для нього червоний колір тексту. Хоча контейнер **<b>** зустрічається в першому абзаці три рази, червоним кольором буде виділено лише перша згадка, т.б. текст «Lorem ipsum». В інших випадках вміст контейнера **<b>** відображається чорним кольором. З наступним абзацом все починається знову, оскільки батьківський елемент помінявся. Тому фраза «Ut wisis enim» також буде виділена червоним кольором.

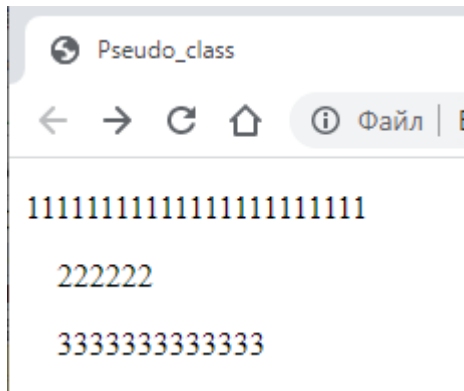
Псевдоклас **:first-child** найзручніше використовувати в тих випадках, коли потрібно задати різний стиль для першого і інших однотипних елементів. Наприклад, в деяких випадках новий рядок для першого абзацу тексту не встановлюють, а для інших абзаців додають відступ першого рядка. З цією метою застосовують властивість **text-indent** з потрібним значенням відступу. Але щоб

змінити стиль першого абзацу і прибрати для нього відступ буде потрібно скористатися псевдоклас **:first-child**

Приклад

```
<html>
<head>
  <meta charset="utf-8">
  <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>11111111111111111111111111</p>
  <p>222222</p>
  <p>333333333333333</p>
</body>
</html>
```

Результат



### Псевдокласи, що задають мову тексту

Для документів, одночасно містять тексти на декількох мовах має значення дотримання правил синтаксису, характерні для тієї чи іншої мови. За допомогою псевдокласів можна змінювати стиль оформлення закордонних текстів, а також деякі настройки.

**:lang**



Визначає мову, який використовується в документі або його фрагменті. У кодї HTML мова встановлюється через атрибут lang, він зазвичай додається до тегу <html>. За допомогою псевдокласу :lang можна задавати певні настройки, характерні для різних мов, наприклад, вид лапок в цитатах. Синтаксис наступний.

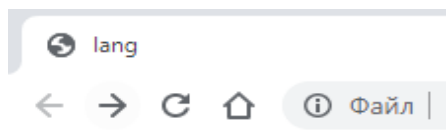
Елемент: lang (мова) {...}

В якості мови можуть виступати наступні значення: ua - український, ru - російська; en - англійська; de - німецький; fr - французький; it - італійський.

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:



FR: «Devis».

DE: „Zitate“.

EN: “Quotes”.

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>`.

### Псевдоелементи

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

Синтаксис використання псевдоелементів наступний.

#### **Селектор: Псевдоелемент {Опис правил стилю}**

Спочатку слід ім'я селектора, потім пишеться двокрапка, після якого йде ім'я псевдоелемента. Кожен псевдоелемент може застосовуватися тільки до одного селектору, якщо потрібно встановити відразу декілька псевдоелементів для одного селектора, правила стилю повинні додаватися до них окремо, як показано нижче.

**.foo:first-letter {color: red}**

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

Псевдоелементи не можуть застосовуватися до внутрішніх стилям, тільки до таблиці пов'язаних або глобальних стилів.

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. У прикладі показано використання псевдоелемента: `after` для додавання тексту в кінець абзацу.

Приклад:

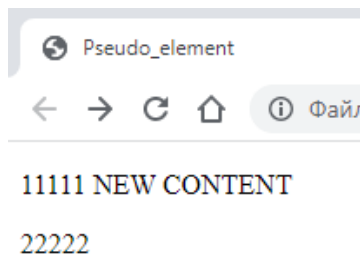
```
<html>
<head>
  <meta charset="utf-8">
  <title>Pseudo_element</title>
```

```

<style>
P.new:after {
  content: " NEW CONTENT";
}
</style>
</head>
<body>
<p class="new">11111</p>
<p>22222</p>
</body>
</html>

```

Результат:



В даному прикладі до вмісту абзацу з класом new додається додаткове слово, яке виступає значенням властивості content.

Псевдоелементи: after і: before, а також стильова властивість content не підтримуються браузером Internet Explorer до сьомої версії включно.

### **: before**

За своєю дією: before аналогічний псевдоелементу: after, але вставляє контент до вмісту елементу. У наступному прикладі показано додавання маркерів свого типу до елементів списку за допомогою приховування стандартних маркерів і застосування псевдоелемента: before.

Приклад

```

<html>
<head>
<title>Pseudo_element</title>
<style>
UL {
  padding-left: 0;

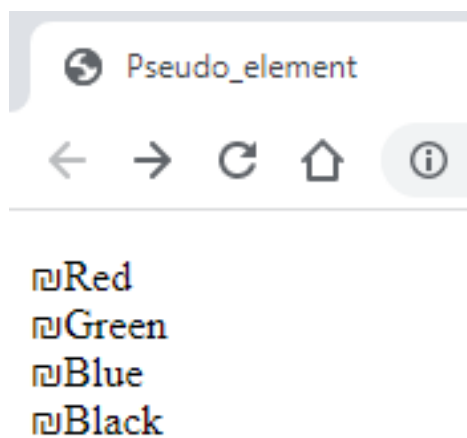
```

```

    list-style-type: none; /* Hiding list markers */ }
LI:before {
    content: "\20aa "; /* Marker: symbol UNICODE */ }
</style>
</head>
<body>
<ul>
    <li>Red</li>
    <li>Green</li>
    <li>Blue</li>
    <li>Black</li>
</ul>
</body>
</html>

```

Результат:



В даному прикладі псевдоелемент: `before` встановлюється для селектора `LI`, що визначає елементи списку. Додавання бажаних символів відбувається шляхом завдання значення властивості `content`. Зверніть увагу, що в якості аргументу не обов'язково виступає текст, можуть застосовуватися також символи Unicode.

І **`:after`** і **`:before`** дають результат тільки для тих елементів, у які містять дані, тому додавання до селектору `img` або `input` нічого не виведе.

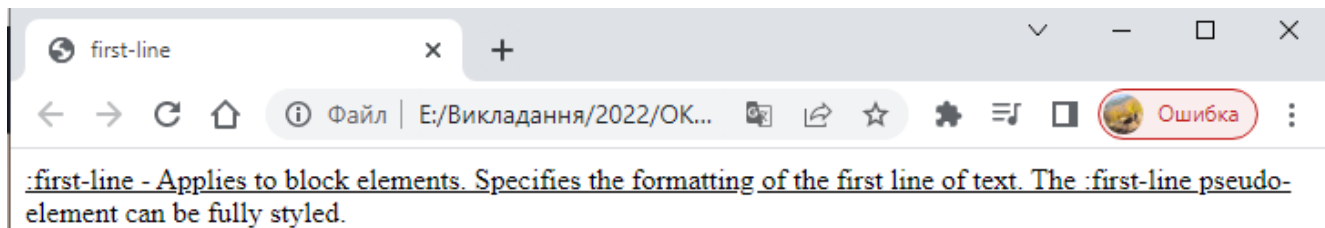
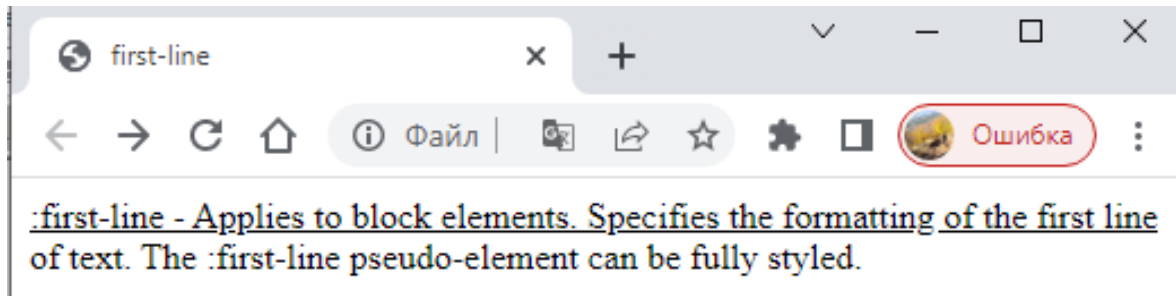
**`:first-line`**

Застосовується для блочних елементів. Задає форматування першого рядка тексту.

До псевдоеlementу: **first-line** можуть застосовуватися не всі стильові властивості. Допустимо використовувати властивості, що відносяться до шрифту, зміни колір тексту і фону, а також: **clear, line-height, letter-spacing, text-decoration, text-transform, vertical-align і word-spacing.**

Приклад:

```
<html>
<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>
</html>
```



### : first-letter

Дозволяє задати форматування першої літери тексту. Для прикладу створимо «буквицу» - початкову літеру тексту збільшеного розміру:

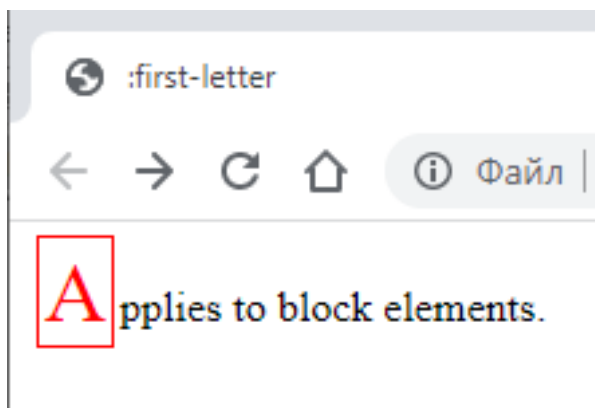
Приклад:

```
<html>
```

```

<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>
</html>

```



## CSS-властивості: позиціонування

### Установка координат елемента

За допомогою CSS можна точно задати положення елемента на сторінці.

Режимом позиціонування управляє властивість **position**:

**position** - встановлює, яким чином обчислюється положення елемента в площині екрану. Існує чотири режими.

**position: static** - режим за замовчуванням, елементи відображаються як зазвичай - в порядку проходження в коді за правилами HTML.

**position: relative** - задає відносне вільне позиціонування.

Значення атрибутів **top**, **right**, **bottom**, і **left** при цьому задають зміщення координат елемента сторінки від точки, в якій він був відображений.

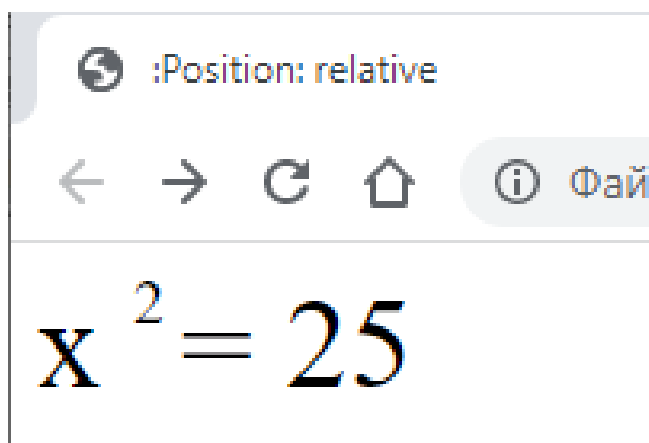
Наприклад, створимо CSS-заміну тегу `<sup> ... </sup>`.

HTML-код:

```
<span style = "font-size: 30pt">
```

```
<span style = "font-size: 50%; position: relative; top: - 1em;"> 2 </span> = 25  
</span>
```

Щоб помістити цифру «8» в верхній індекс, зменшуємо її розмір в половину і направляємо вгору на висоту рядка (1 em). Властивість `top` вказує відстань від початкового положення відносно верхньої межі документа. Для того, щоб підняти «8» наверх, ми вказуємо від'ємне значення `top`. У цьому прикладі можна замість властивості `top: -1em` написати `bottom: 1em`.



При розробці сайтів таким способом користуватися не рекомендується. Для перетворення в верхній індекс краще використовувати спеціально призначений атрибут `vertical-align` із значенням `sub` для нижнього індексу або `super` для верхнього **position: absolute** - задає абсолютне вільне позиціонування.

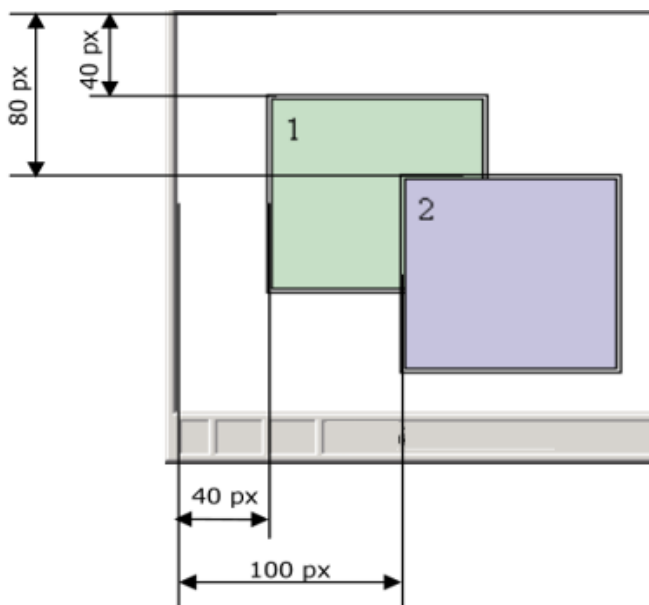
Значення атрибутів **top**, **right**, **bottom** і **left** і при цьому задають абсолютні координати елемента сторінки щодо батька. Створимо два контейнери `DIV` і скористаємося `position: absolute` для вказівки їх координат.

Для блоків задається відступ від верхнього і лівого краю властивостями `top` і `left`. Так як другий блок оголошений в HTML-коді пізніше, він перекриває перший блок на сторінці.

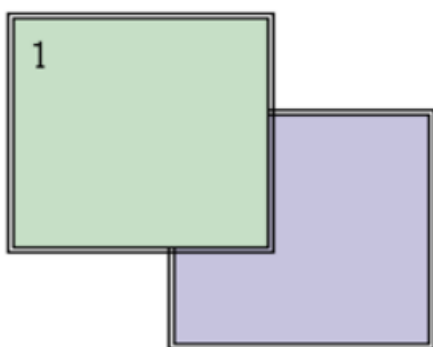
### Приклад:

```
<html>
<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>
</html>
```





Для управління порядком накладення елементів один на одного необхідно використовувати властивість **z-index**. Значним z-index є позитивне або негативне число, що задає «висоту», на якій розташований елемент. Елементи з великим z-index накладаються зверху елементів з меншим z-index. Щоб в попередньому прикладі перший блок виявився вищим другого, необхідно для першого блоку задати z-index, наприклад, рівним двом, а для другого - одиниці.



**position: fixed** - фіксує елемент щодо вікна. Елемент залишається на місці навіть при прокручуванні сторінки. На жаль, режим fixed не працює в браузері Internet Explorer версії 6 і нижче, тому поки застосовувати його не рекомендується.

### 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).

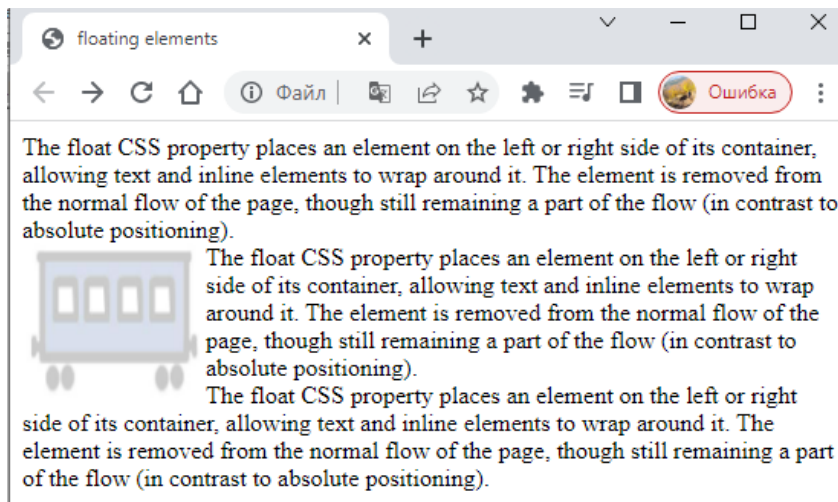
Плаваючий елемент буде прагнути до лівої чи правої сторони батьківського елемента, а з інших сторін він може обтікати текстом або

При цьому потрібно пам'ятати, що властивість float не працює одночасно із завданням позиціонування, розглянутим у першій частині лекції.

Наочно робота float видно на прикладі:

```
<html>
<head>
<title> floating elements</title>
<style>
DIV#floating{
float: left;
}
</style>
</head>
<body>
The float<br>
<div id="floating"></div>
The float<br>
The float<br>
</body>
</html>
```

Контейнер DIV із зображенням прагне до лівого краю документа, а з інших трьох сторін він обтекається текстом

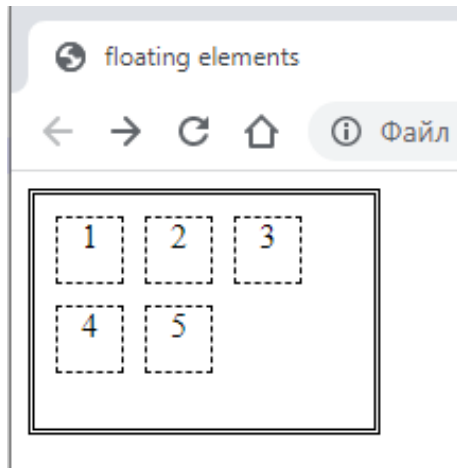


Створимо приклад з декількома плаваючими блоками. Задамо основний контейнер з фіксованою шириною, а в нього помістимо п'ять плаваючих блоків з вирівнюванням по лівому краю.

Приклад:

```
<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>

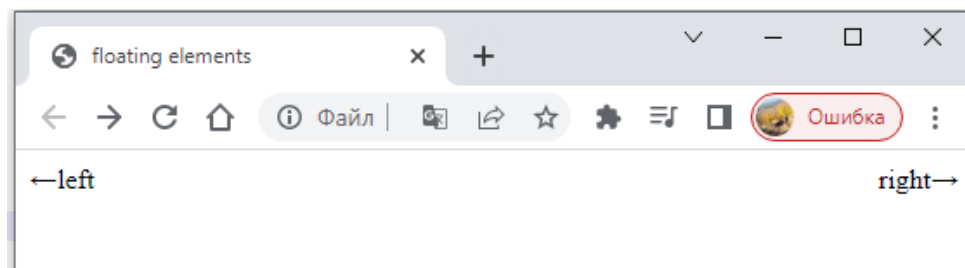


Перший блок вирівнюється по лівому краю батьківського контейнера. Другий блок теж прагне до лівого краю, але так як місце вже зайнято першим блоком, другий блок стає (обтікає) праворуч від першого. Аналогічно надходить третій блок. Четвертий блок вже не може встати праворуч від третього, тому він поміщається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

Можна одночасно використовувати блоки з вирівнюванням по лівому і правому краю.

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

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



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

Розглянемо приклад:

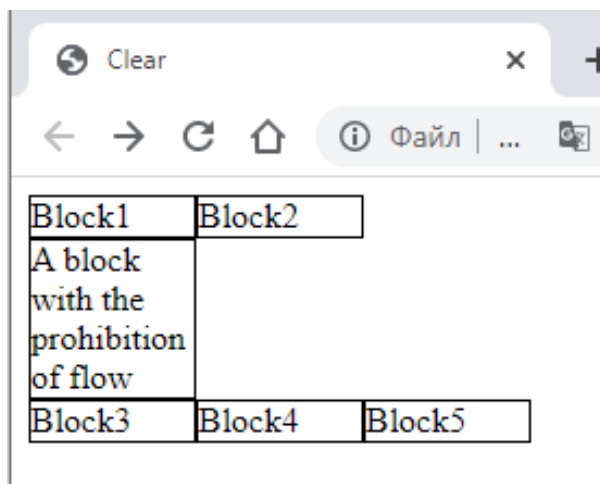
<html>

```

<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>
</html>

```

Результат:



При створенні сайтів плаваючі елементи, властивості float і clear часто використовуються для створення «каркаса» сторінок сайту.

#### 4. Порядок виконання лабораторної роботи

- 1) ознайомитися з теоретичними відомостями;
- 2) виконати завдання
- 3) оформити звіт, який включатиме: титульний аркуш, завдання, теоретичні відомості, результати і висновки по роботі;
- 4) продемонструвати результат на комп'ютері і захистити лабораторну роботу.

## Лабораторна робота №6

### 1. Тема

JavaScript. Внутрішні, зовнішні скрипти. Змінні. Умови. Цикли. Функції. DOM. BOM. Браузер: документ(document)

### 2. 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`.

2)

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

За допомогою об'єкта `location` перенаправити браузер на іншу сторінку.

Використовуючи метод `getElementById`

Використати метод `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**).

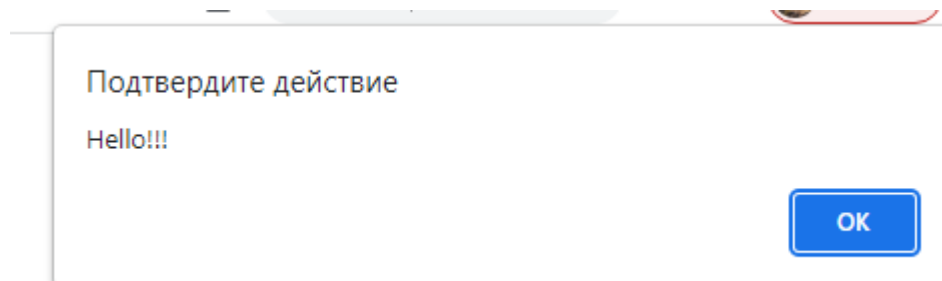
### 3. Теоретичні відомості

Тег «script»

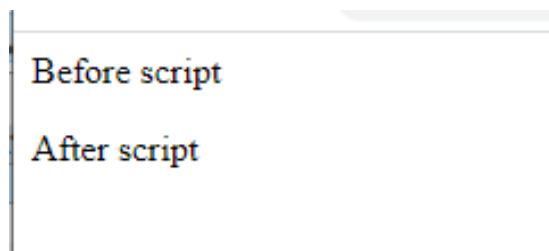
Програми на JavaScript можуть бути вставлені в будь-яке місце HTML-документа за допомогою тега `<script>`.

Для прикладу:

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







Тег `<script>` містить JavaScript-код, який автоматично виконується, коли браузер его обробляє.

### Сучасна розмітка

Тег `<script>` має кілька атрибутів, які рідко використовуються, але все ще можуть зустрітися:

Атрибут `type`: `<script type = ...>`

Старий стандарт HTML, HTML4, вимагав наявності цього атрибута в тезі `<script>`. Зазвичай він мав значення `type = "text / javascript"`. На поточний момент цього більше не потрібно. Більш того, в сучасному стандарті HTML сенс цього атрибута повністю змінився. Тепер він може використовуватися для JavaScript-модулів.

Атрибут `language`: `<script language = ...>`

Цей атрибут повинен був ставити мову, на якій написаний скрипт. Але так як JavaScript є мовою за замовчуванням, в цьому атрибуті вже немає необхідності.

### 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 папці.

Можна вказати і повну URL-адресу. наприклад:

```
<script  
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js  
></script>
```

Для підключення декількох скриптів використовуйте кілька ключових слів:

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

## Структура коду

Почнемо вивчення мови з розгляду основних «будівельних блоків» коду.

### Інструкції

Інструкції - це синтаксичні конструкції і команди, які виконують дії.

Ми вже бачили інструкцію alert ( 'Привет!'), Яка відображає повідомлення «Привет!».

У нашому коді може бути стільки інструкцій, скільки ми захочемо. Інструкції можуть відділятися крапкою з комою.

Наприклад, тут ми розділили повідомлення «Привіт Світ» на два виклики alert:

```
alert('Привіт'); alert('Світ');
```

## Змінні

JavaScript-додатком 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. Ми можемо використовувати змінні для зберігання товарів, відвідувачів і інших даних.

Для створення змінної в JavaScript використовуйте ключове слово `let`.

Наведена нижче інструкція створює (іншими словами: оголошує або визначає) змінну з ім'ям «message»:

```
let message;
```

Тепер можна помістити 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
```

*var замість let*

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

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

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

### 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. Глобальну область видимості інакше ще називають кодом верхнього рівня.

### Глобальні, локальні змінні

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

Змінна, оголошена всередині функції, називається локальною. Локальна змінна доступна в будь-якому місці всередині тіла функції, в якій вона була оголошена. Локальна змінна створюється кожен раз 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 = "глобальна"; // Глобальна змінна
function checkscope() {
  var x = "локальна"; // Локальна змінна з тим же ім'ям
  document.write(x); // Використовується локальна змінна, а не
глобальна
}
checkscope(); // => "локальна"
```

## Блокові змінні

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
```

## Відмінності `let` і `var`

Зазвичай `var` не використовується в сучасних скриптах, але все ще може ховатися в старих.

На перший погляд, поведінка `var` схожа на `let`. Наприклад, оголошення змінної:

```
function sayHi() {
  var phrase = "Привіт"; // локальна змінна, "var" замість "let"
  alert(phrase); // Привіт
}

sayHi();

alert(phrase); // Помилка: phrase не визначена
```

... Проте, відмінності все ж є.

## Для «var» не існує блокової області видимості

Область видимості змінних var обмежується функціями, або, якщо змінна глобальна, то скриптом. Такі змінні доступні за межами блоку.

Наприклад:

```
if (true) {  
    var test = true; // використовуємо var замість let  
}  
  
alert(test); // true, змінна існує поза блоком if
```

Так як var ігнорує блоки, ми отримали глобальну змінну test.

А якби ми використовували let test замість var test, тоді змінна була б видна тільки всередині if:

```
if (true) {  
    let test = true; // використовуємо let  
}  
  
alert(test); // Error: test is not defined
```

Аналогічно для циклів: var не може бути блокової або локальної всередині циклу:

```
for (var i = 0; i < 10; i++) {  
    // ...  
}  
  
alert(i); // 10, змінна i доступна поза циклом, тому що є глобальною змінною
```

Якщо блок коду знаходиться всередині функції, то var стає локальною змінною в цій функції:

```
function sayHi() {  
    if (true) {  
        var phrase = "Привіт";    }  
    alert(phrase); } // спрацьовує і виводить "Привіт"  
  
sayHi();  
  
alert(phrase); // Помилка: phrase не визначена
```

## 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;
```

Мови програмування, в яких таке можливо, називаються «динамічно типізовані». Це означає, що типи даних є, але змінні не прив'язані до жодного з них.

### Число

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

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

## Рядок

Рядок (string) в JavaScript повинен бути укладений в лапки.

```
let str = "Привіт";
```

```
let str2 = 'Одинарні лапки';
```

```
let phrase = `обернені лапки дозволяють вбудовувати змінні ${str}`;
```

В JavaScript існує три типи лапок:

1. Подвійні лапки: "Привіт".
2. Одинарні лапки: 'Привіт'.
3. Зворотні лапки: `Привіт`.

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 \$ {...}. наприклад:

```
let name = "Иван";
```

```
// Вставим переменную
```

```
alert( `Привет, ${name}!` ); // Привет, Иван!
```

```
// Вставим выражение
```

```
alert( `результат: ${1 + 2}` ); // результат: 3
```

```
let name = "Ivan";
```

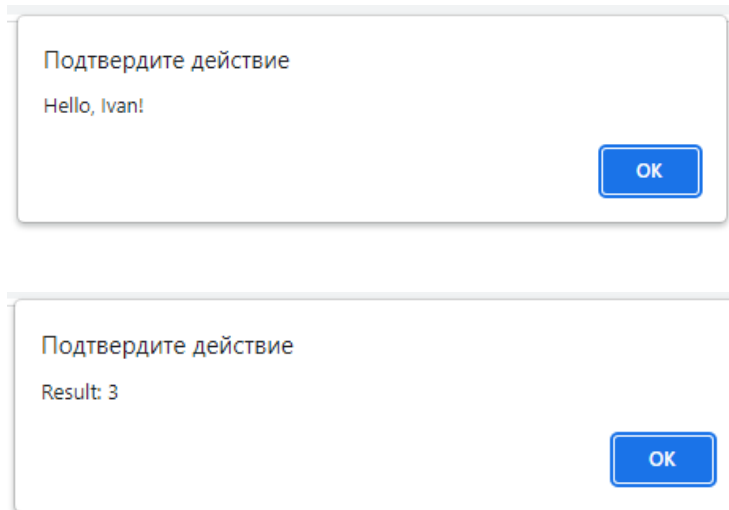
```
// Вставимо змінну
```

```
alert(`Hello, ${name}!` );
```

```
// Вставимо вираз
```

```
alert(`Result: ${1 + 2}` );
```





Вираз всередині `$ {...}` обчислюється, і його результат стає частиною рядка. Ми можемо покласти туди все, що завгодно: змінну `name`, або вираз `1 + 2`, або щось більш складне.

Зверніть увагу, що це можна робити тільки в зворотних лапках. Інші лапки не мають такої функціональності вбудовування!

```
alert( "результат: ${1 + 2}" ); // результат: ${1 + 2} (подвійні лапки нічого не роблять)
```

Немає окремого типу даних для одного символу.

### Булевий (логічний) тип

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».

Наприклад:

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

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

### Значення «null»

Спеціальне значення null не відноситься ні до одного з типів, описаних вище.

Воно формує окремий тип, який містить тільки значення null:

```
let age = null;
```

В JavaScript null не є «посиланням на неіснуючий об'єкт» або «нульовим покажчиком», як в деяких інших мовах.

Це просто спеціальне значення, яке представляє собою «нічого», «порожньо» або «значення невідомо».

### Значення «undefined»

Спеціальне значення undefined також стоїть особно. Воно формує тип з самого себе так само, як і null.

Воно означає, що «значення не було присвоєно».

Якщо змінна оголошена, але їй не присвоєно ніякого значення, то її значенням буде undefined:

```
let age;  
  
alert(age); // виведе "undefined"
```

### Об'єкти і символи

Тип object (об'єкт) - особливий.

Всі інші типи називаються «примітивними», тому що їх значеннями можуть бути тільки прості значення (будь то рядок, або число, або щось ще). В об'єктах же зберігають колекції даних або більш складні структури.

Об'єкти займають важливе місце в мові і вимагають особливої уваги.

Об'єкт може бути створений за допомогою фігурних дужок `{...}` з необов'язковим списком властивостей. Властивість - це пара «ключ: значення», де ключ - це рядок (так зване «ім'ям властивості»), а значення може бути чим завгодно.

Порожній об'єкт ( «порожній ящик») 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 літеральної нотацією.

### Літерали і властивості

При використанні літерального синтаксису `{...}` ми відразу можемо помістити в об'єкт кілька властивостей у вигляді пар «ключ: значення»:

```
let user = { // об'єкт
```

```
  name: "John", // під ключом "name" зберігається значення "John"
```

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

```
};
```

У кожної властивості є ключ (також званий «ім'я» або «ідентифікатор»). Після імені властивості слідує двокрапка `:"`, і потім вказується значення властивості. Якщо в об'єкті є кілька властивостей, то вони перераховуються через кому.

Для звернення до властивостей використовується запис «через крапку»:

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

Значення може бути будь-якого типу. Давайте додамо властивість з логічним значенням:

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

Для видалення властивості ми можемо використовувати оператор delete:

```
delete user.age;
```

## Symbol

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 тільки ці два типи даних.

Розберемо символи, побачимо, що хорошого вони нам дають.

Створюються нові символи за допомогою функції Symbol ():

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

При створенні символу можна дати опис (також зване ім'я), в основному використовується для налагодження коду:

```
let id = Symbol("id"); // створюємо символ id з описом (ім'ям) "id"
```

Оператор typeof

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:

1. Синтаксис оператора: `typeof x`.
2. Синтаксис функції: `typeof (x)`.

Іншими словами, він працює з дужками або без дужок. Результат однаковий.

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)
```

### **Взаємодія: `alert`, `prompt`, `confirm`**

Так як ми будемо використовувати браузер як демо-середовище, нам потрібно познайомитися з декількома функціями його інтерфейсу, а саме: `alert`, `prompt` і `confirm`.

#### **`alert`**

З цією функцією ми вже знайомі. Вона показує повідомлення і чекає, поки користувач натисне кнопку «ОК».

Наприклад:

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



Це невелике вікно з повідомленням називається модальним вікном. Поняття модальное означає, що користувач не може взаємодіяти з інтерфейсом решти сторінки, натискати на інші кнопки і т.д. до тих пір, поки взаємодіє з вікном. В даному випадку - поки не буде натиснута кнопка «ОК».

## **prompt**

Функція `prompt` приймає два аргументи:

```
result = prompt(title, [default]);
```

Цей код відобразить модальне вікно з текстом, полем для введення тексту і кнопками ОК / Скасування.

### **title**

Текст для відображення у вікні.

### **default**

Необов'язковий другий параметр, який встановлює початкове значення в поле для тексту в вікні.

## Квадратні дужки в синтаксисі [...]

Квадратні дужки навколо default в описаному вище синтаксисі означають, що параметр факультативний, необов'язковий.

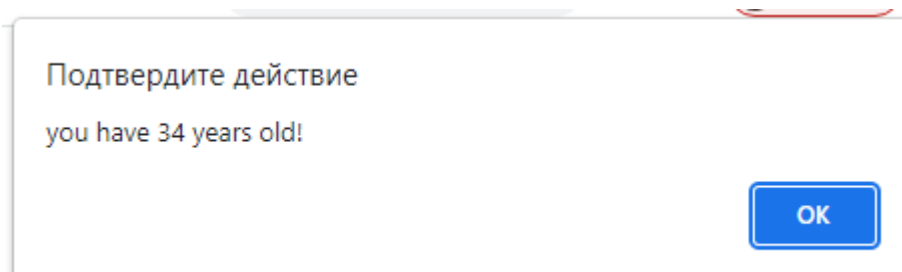
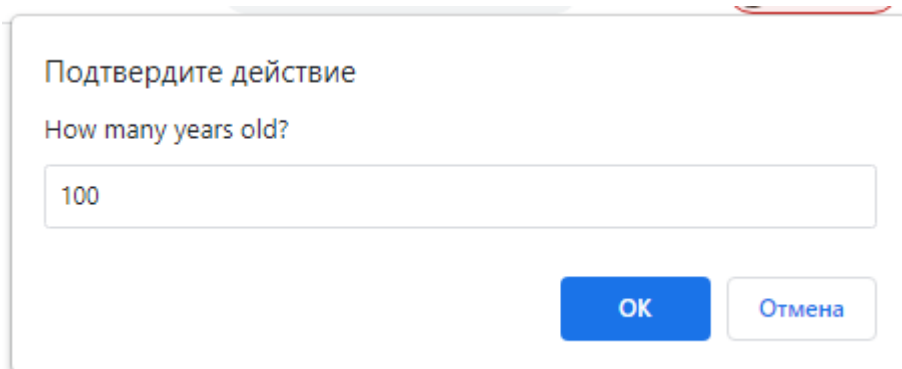
Користувач може надрукувати що-небудь в поле введення і натиснути ОК. Введений текст буде присвоєно змінної result. Користувач також може скасувати введення натисканням на кнопку «Скасування» або натиснувши на клавішу Esc. В цьому випадку значенням result стане null.

Виклик prompt повертає текст, вказаний в полі для введення, або null, якщо введення скасований користувачем.

Наприклад:

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

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

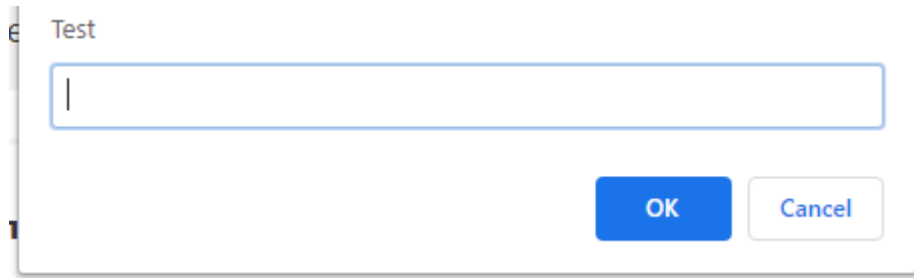


Для IE: завжди встановлюйте значення за замовчуванням

Другий параметр є необов'язковим, але якщо не вказати його, то Internet Explorer вставить рядок "undefined" в поле для введення.

Запустіть код в Internet Explorer і подивіться на результат:

```
let test = prompt("Test");
```



Щоб prompt добре виглядав в ІЕ, рекомендується завжди вказувати другий параметр:

```
let test = prompt("Test", ''); // <-- для ІЕ
```

## **confirm**

Синтаксис:

```
result = confirm(question);
```

Функція confirm відображає модальне вікно з текстом питання question і двома кнопками: ОК і Скасування.

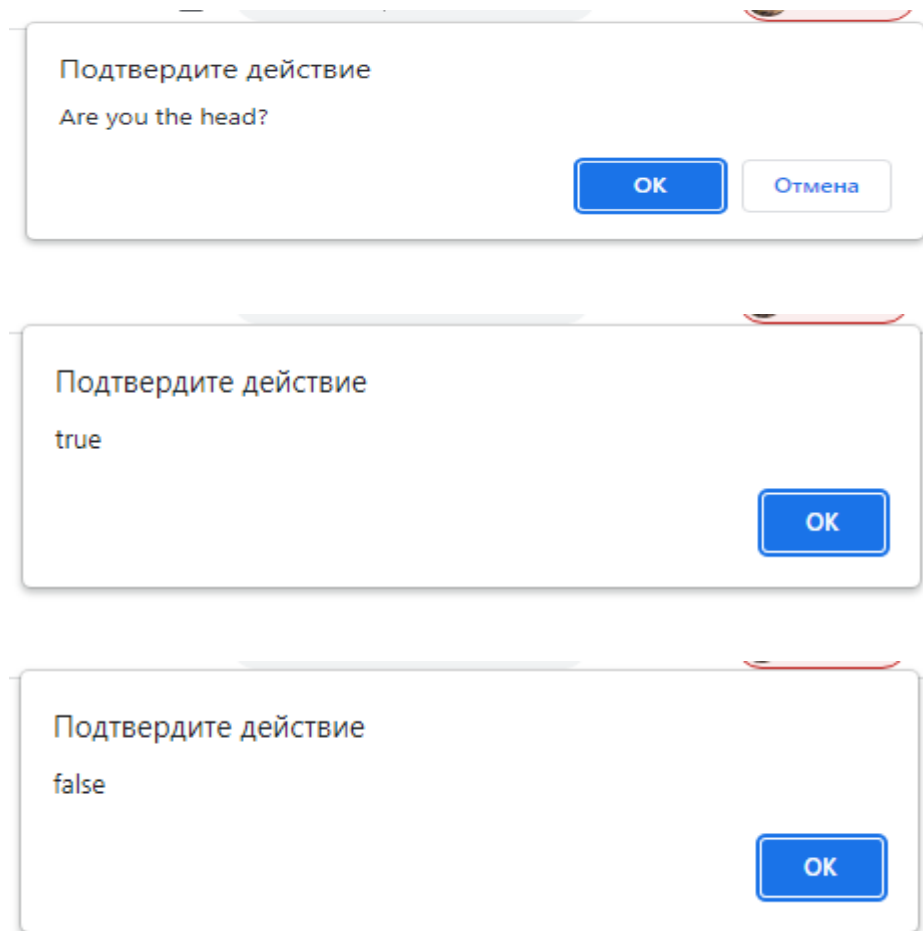
Результат - true, якщо натиснута кнопка ОК. В інших випадках - false.

Наприклад:

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

```
alert( isBoss ); // true, якщо натиснута ОК
```





Всі ці методи є модальними: зупиняють виконання скриптів і не дозволяють користувачеві взаємодіяти з іншою частиною сторінки до тих пір, поки вікно не буде закрито.

На всі зазначені методи поширюються два обмеження:

Розташування вікон визначається браузером. Зазвичай вікна знаходяться в центрі.

Візуальне відображення вікон залежить від браузера, і ми не можемо змінити їх вигляд.

## 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)`. Для примітивних значень працює очевидним чином.

Чисельне - Відбувається в математичних операціях. Може бути викликано за допомогою `Number (value)`.

Перетворення підпорядковується правилам:

Значення	Стає
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>true / false</code>	<code>1 / 0</code>
<code>string</code>	Пробільні символи по краях обрізаються. Далі, якщо залишається порожній рядок, то отримуємо <code>0</code> , інакше з непорожній рядки «зчитується» число. При помилку результат <code>NaN</code> .

Логічне - Відбувається в логічних операціях. Може бути викликано за допомогою `Boolean (value)`.

Підпорядковується правилам:

Значення	Стає
<code>0, null, undefined, NaN, ""</code>	<code>false</code>
будь-яке інше значення	<code>true</code>

Більшу частину з цих правил легко зрозуміти і запам'ятати. Особливі випадки, в яких часто припускаються помилок:

undefined при чисельному перетворенні стає NaN, не 0.

"0" і рядки з одних прогалін типу " " при логічному перетворенні завжди true.

Приклад

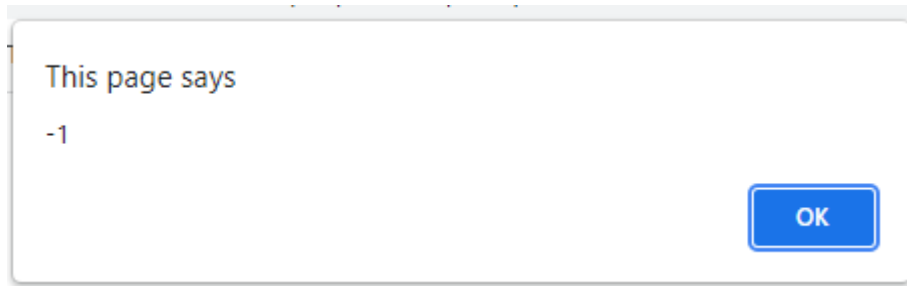
```
let str = "123";  
alert(typeof str); // string  
  
let num = Number(str); // стає числом 123  
alert(typeof num); // number
```



## Базові оператори

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

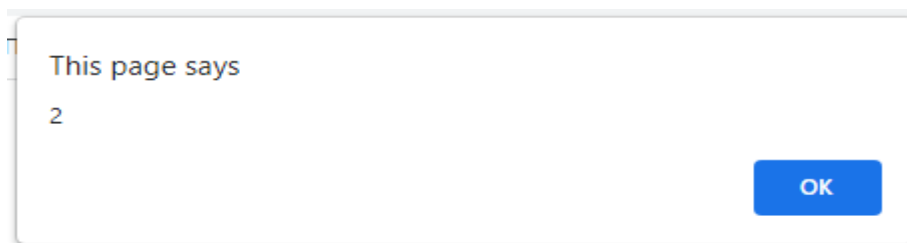
```
let x = 1;  
  
x = -x;  
  
alert( x ); // -1, застосували унарний мінус
```



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

```
let x = 1, y = 3;
```

```
alert( y - x ); // 2, бінарний мінус віднімає значення
```



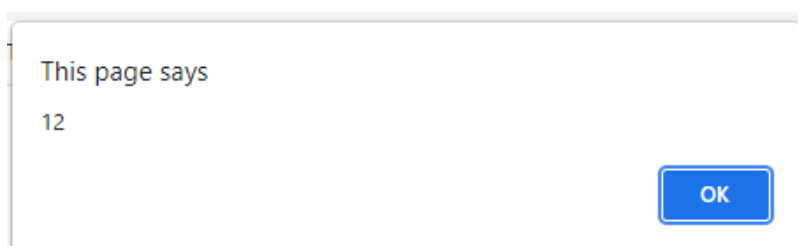
Додавання рядків за допомогою бінарного +

Давайте розглянемо більш доступного режиму операторів JavaScript, які виходять за рамки шкільної арифметики.

Зазвичай за допомогою плюса '+' складають числа.

```
alert( '1' + 2 ); // "12"
```

```
alert( 2 + '1' ); // "21"
```



Приведення до числа, унарний +

Плюс + існує в двох формах: бінарної, яку ми використовували вище, і унарною.

Унарний, тобто застосований до одного значення, плюс + нічого не робить з числами. Але якщо операнд не числиться, унарний плюс перетворює його в число.

Наприклад:

```
// 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
```



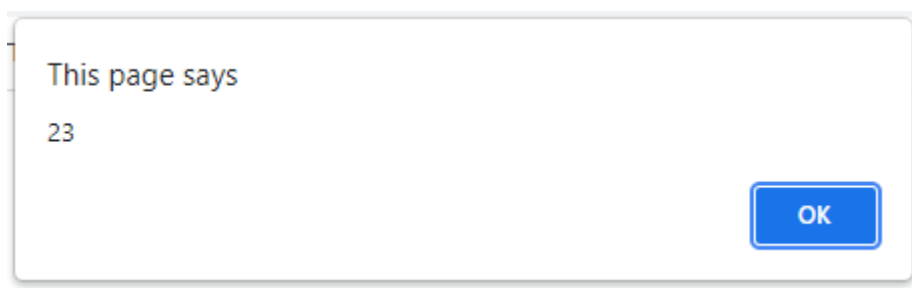
Насправді це те ж саме, що і Number (...), тільки коротше.

Необхідність перетворювати рядки в числа виникає дуже часто. Наприклад, зазвичай значення полів HTML-форми - це рядки. А що, якщо їх потрібно, наприклад, скласти?

Бінарний плюс складе їх як рядки:

```
let apples = "2";  
let oranges = "3";
```

```
alert( apples + oranges ); // "23", бінарний плюс об'єднує рядки
```



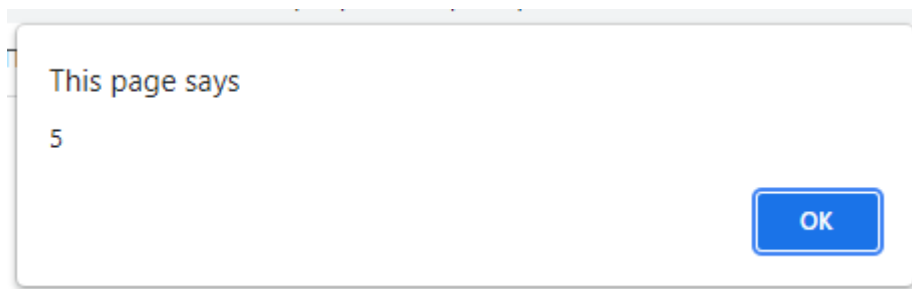
Тому використовуємо унарний плюс, щоб перетворити до числа:

```
let apples = "2";
```

```
let oranges = "3";
```

```
// операнди попередньо перетворенв числа
```

```
alert( +apples + +oranges ); // 5
```



## Присвоєння

Коли змінній щось присвоюють, наприклад,  $x = 2 * 2 + 1$ , то спочатку виконається арифметика, а вже потім відбудеться присвоєння = зі збереженням результату в x.

```
let x = 2 * 2 + 1;
```

```
alert( x ); // 5
```

### Присвоєння = повертає значення

Той факт, що = є оператором, а не «магічною» конструкцією мови, має цікаві наслідки.

Більшість операторів в JavaScript повертають значення. Для деяких це очевидно, наприклад додавання + або множення \*. Але і оператор присвоєння не є винятком.

Виклик  $x = \text{value}$  записує value в x і повертає його.

Завдяки цьому присвоєння можна використовувати як частину більш складного виразу:

```
let a = 1;
```

```
let b = 2;
```

```
let c = 3 - (a = b + 1);
```

```
alert( a ); // 3
```

```
alert( c ); // 0
```

В наведеному вище прикладі результатом  $(a = b + 1)$  буде значення, яке присвоюється змінній a (тобто 3). Потім воно використовується для подальших обчислень.

Однак писати таким в такому стилі не рекомендується. Такі трюки не зроблять ваш код зрозумілішим або читабельним.

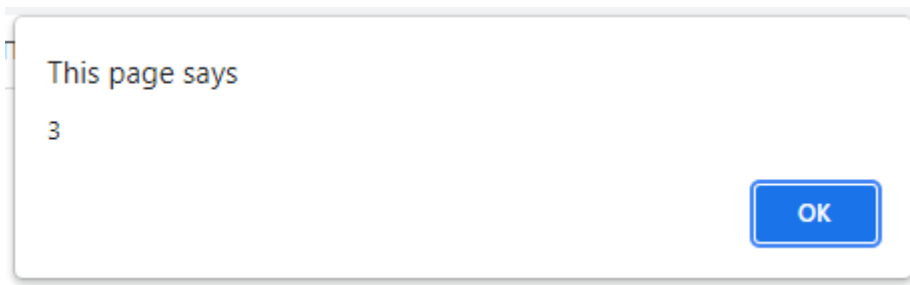
## Інкремент / декремент

Однією з найбільш частих числових операцій є збільшення або зменшення на одиницю.

Для цього існують навіть спеціальні оператори:

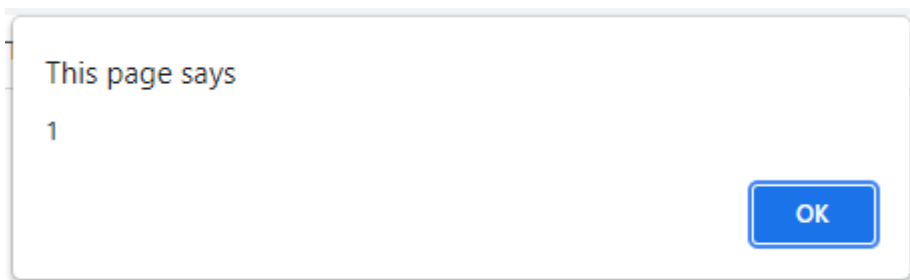
Інкремент ++ збільшує змінну на 1:

```
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
```



Важливо: інкремент / декремент можна застосувати тільки до змінної. Спроба використовувати його на значенні, типу 5 ++, призведе до помилки.



## Оператори порівняння

В JavaScript вони записуються так:

- Більше / менше:  $a > b$ ,  $a < b$ .
- Більше / менше або дорівнює:  $a \geq b$ ,  $a \leq b$ .
- Так само:  $a == b$ . Зверніть увагу, для порівняння використовується подвійний знак рівності  $==$ . Один знак рівності  $a = b$  означав би присвоєння.
- Не дорівнює. В JavaScript записується як  $a != b$ .

### Результат порівняння має логічний тип

Всі оператори порівняння повертають значення логічного типу:

true - означає «так», «вірно», «істина».

false - означає «ні», «не так», «брехня».

### Порівняння рядків

Щоб визначити, що один рядок більше другий, JavaScript використовує «алфавітний» або «словниковий» порядок.

Іншими словами, рядки порівнюються посимвольно.

Наприклад:

```
alert( 'Я' > 'А' ); // true
```

```
alert( 'Коти' > 'Кода' ); // true
```

```
alert( 'дієвий' > 'ді' ); // true
```

Алгоритм порівняння двох рядків досить простий:

1. Спочатку порівнюються перші символи рядків.
2. Якщо перший символ першого рядка більше (менше), ніж перший символ другого, то перший рядок більше (менше) другого. Порівняння завершено.
3. Якщо перші символи рівні, то таким же чином порівнюються вже другі символи рядків.
4. Порівняння триває, поки не закінчиться один з рядків.
5. Якщо обидва рядки закінчуються одночасно, то вони рівні. Інакше, більшим вважається більш довгий рядок.

### Порівняння різних типів

При порівнянні значень різних типів JavaScript призводить кожне з них до числа.

Наприклад:

```
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.

Наприклад:

```
alert( true == 1 ); // true
```

```
alert( false == 0 ); // true
```

### Суворе (строге) порівняння

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.

Як же тоді відрізнити 0 від false?

**Оператор строгої рівності === перевіряє рівність без приведення типів.**

Іншими словами, 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 довше писати, але він робить код більш очевидним і залишає менше місця для помилок.

### Порівняння з null і undefined

Поведінка null і undefined при порівнянні з іншими значеннями - 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 інших операторів порівняння  $<>$   $<=>$  =

Значення null / undefined перетворюються до чисел: null стає 0, а undefined - NaN.

Подивимося, які цікаві речі трапляються, коли ми застосовуємо ці правила. І, що більш важливо, як уникнути помилок при їх використанні.

### *Дивний результат порівняння null і 0*

Порівняємо null з нулем:

```
alert( null > 0 ); // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
```

З точки зору математики це дивно. Результат останнього порівняння говорить про те, що "null більше або дорівнює нулю", тоді результат одного з порівнянь вище повинен бути true, але вони обидва хибні.

Причина в тому, що нестрога рівність і порівняння  $>$   $<>$   $=$   $<=$  працюють по-різному. Порівняння перетворюють null в число, розглядаючи його як 0. Тому вираз (3)  $\text{null} \geq 0$  істинний, а  $\text{null} > 0$  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)  $\text{null} == 0$  помилковий.

### Незрівнянне значення undefined

Значення undefined незрівнянно з іншими значеннями:

```
alert( undefined > 0 ); // false (1)
```

```
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

Чому ж порівняння `undefined` з нулем завжди помилково?

На це є такі причини:

- 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.
- нестрога рівність (3) повертає `false`, тому що `undefined` дорівнює тільки `null`, `undefined` і нічому більше.

### Умовне розгалуження: `if`, `'?'`

При написанні скриптів часто постає завдання зробити однотипну дію багато разів.

Наприклад, вивести товари зі списку один за іншим. Або просто перебрати всі числа від 1 до 10 і для кожного виконати однаковий код.

Для багаторазового повторення однієї ділянки коду передбачені цикли.

### Цикл «while»

Цикл `while` має наступний синтаксис:

```
while (condition) {
    // код або
    // "тіло циклу"
}
```

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++; }
```

This page says

0

OK

This page says

1

OK

This page says

2

OK

Одне виконання тіла циклу називається ітерацією. Цикл в прикладі вище робить три ітерації.

Якби рядок `i ++` був відсутній в прикладі вище, то цикл повторювався б (в теорії) вічно. На практиці, звичайно, браузер не дозволить такому трапитися, він надасть користувачеві можливість зупинити «підвисший» скрипт, а JavaScript на стороні сервера доведеться «вбити» процес.

Будь-який вираз або змінна може бути умовою циклу, а не тільки порівняння: умова `while` обчислюється і перетворюється в логічне значення.

Наприклад, `while (i)` - більш короткий варіант `while (i != 0)`:

```
let i = 3;
```

```
while (i) { // коли i буде рівно 0, умова стане false, і цикл зупиниться
```

```
  alert( i );
```

```
  i--;
```

```
}
```

### Цикл «do ... while»

Перевірку умови можна розмістити під тілом циклу, використовуючи спеціальний синтаксис do..while:

```
do {  
    // тіло цикла  
} while (condition);
```

Цикл спочатку виконає тіло, а потім перевірить умова condition, і поки її значення дорівнює true, вона буде виконуватися знову і знову.

Наприклад:

```
let i = 0;  
do {  
    alert( i );  
    i++;  
} while (i < 3);
```

Така форма синтаксису виправдана, якщо ви хочете, щоб тіло циклу виповнилося хоча б один раз, навіть якщо умова виявиться помилковим. На практиці частіше використовується форма з передумовою: while (...) {...}.

### Цикл «for»

Найпоширеніший цикл - цикл for.

Виглядає він так:

```
for (початкова умова; умова; крок) {  
    // ... тіло циклу ...  
}
```

Цикл нижче виконує alert(i) для i від 0 до (але не включаючи) 3:

```
for (let i = 0; i < 3; i++) { // виведе 0, 1, 2  
    alert(i);  
}
```

Розглянемо конструкцію for детальніше:

- 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.

В цілому, алгоритм роботи циклу виглядає наступним чином:

Виконати \* початок \*

→ (Якщо \* умова \* == true → Виконати \* тіло \*, Виконати \* крок \*)

→ (Якщо \* умова \* == true → Виконати \* тіло \*, Виконати \* крок \*)

→ (Якщо \* умова \* == true → Виконати \* тіло \*, Виконати \* крок \*)

→ ...

Тобто, початок виконується один раз, а потім кожна ітерація полягає в перевірці умови, після якої виконується тіло і крок.

Приклад

```
// 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
```

*Вбудоване оголошення змінної*

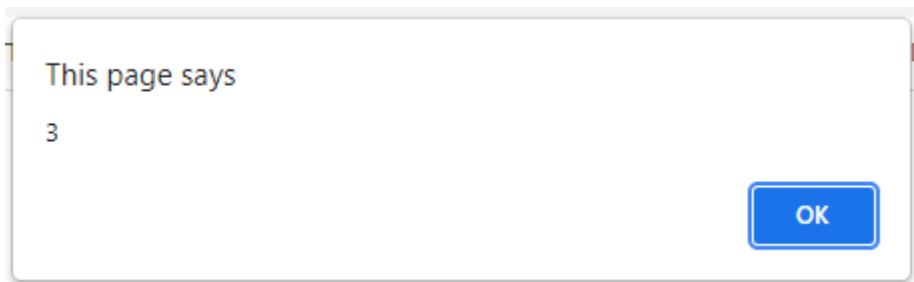


У прикладі змінна лічильника `i` була оголошена прямо в циклі. Це так зване «вбудоване» оголошення змінної. Такі змінні існують тільки всередині циклу.

```
for (let i = 0; i < 3; i++) {  
    alert(i); // 0, 1, 2  
}  
alert(i); // помилка, змінна не існує за межами циклу
```

Замість оголошення нової змінної ми можемо використовувати вже існуючу:

```
let i = 0;  
for (i = 0; i < 3; i++) { // використовуємо існуючу змінну  
    alert(i); // 0, 1, 2  
}  
alert(i); // 3, змінна доступна
```



### Переривання циклу: «break»

Зазвичай цикл завершується при обчисленні умови в `false`.

Але ми можемо вийти з циклу в будь-який момент за допомогою спеціальної директиви `break`.

Наприклад, наступний код підраховує суму чисел, що вводяться до тих пір, поки відвідувач їх вводить, а потім - видає:

```
let sum = 0;  
while (true) {  
    let value = +prompt("ввести число", '');  
    if (!value) break; // (*)  
    sum += value; }  
alert( 'Сума: ' + sum );
```

The image displays four sequential alert dialog boxes, each with the title "This page says".

- The first dialog box contains the text "ввести число" (enter number) and a text input field with the value "1". It has "OK" and "Cancel" buttons.
- The second dialog box contains the text "ввести число" and a text input field with the value "6". It has "OK" and "Cancel" buttons.
- The third dialog box contains the text "ввести число" and a text input field with the value "7". It has "OK" and "Cancel" buttons.
- The fourth dialog box contains the text "Сума: 7" (Sum: 7) and has a single "OK" button.

Директива `break` в рядку (\*) повністю припиняє виконання циклу і передає управління на рядок за його тілом, тобто на `alert`.

Взагалі, поєднання «нескінченний цикл + `break`» - відмінна штука для тих ситуацій, коли умова, за якою потрібно перерватися, знаходиться не на початку або наприкінці циклу, а посередині.

### Перехід до наступної ітерації: continue

Директива continue - «полегшена версія» break. При її виконанні цикл не переривається, а переходить до наступної ітерації (якщо умова все ще true).

Її використовують, якщо зрозуміло, що на поточному повторі циклу робити більше нічого.

Наприклад, цикл нижче використовує continue, щоб виводити тільки непарні значення:

```
for (let i = 0; i < 10; i++) {  
  // якщо true, пропустити частину цикла  
  if (i % 2 == 0) continue;  
  alert(i); } // 1, затем 3, 5, 7, 9
```

Для парних значень i, директива continue припиняє виконання тіла циклу і передає управління на наступну ітерацію for (з наступним числом). Таким чином alert викликається тільки для непарних значень.

### Конструкція "switch"

Конструкція switch замінює собою відразу кілька if.

Вона являє собою більш наочний спосіб порівняти вираз відразу з декількома варіантами.

Синтаксис

Конструкція switch має один або більше блок case і необов'язковий блок default.

Виглядає вона так:

```
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 від відповідної директиви `case` і далі, до найближчого `break` (або до кінця `switch`).

Якщо жоден `case` не співпав - виконується (якщо є) варіант `default`.

Приклад використання `switch` (спрацював код виділений):

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Мало' );
    break;
  case 4:
    alert( 'В точку!' );
    break;
  case 5:
    alert( 'Перебір ' );
    break;
  default:
    alert('нема таких значень');
```

Тут оператор `switch` послідовно порівнює `a` з усіма варіантами з `case`.

Спочатку 3, потім - так як немає збігу - 4. Збіг знайдено, буде виконаний цей варіант, з рядка `alert ( 'В точку!')` і далі, до найближчого `break`, який перерве виконання.



Якщо break немає, то виконання піде нижче за наступними case, при цьому інші перевірки ігноруються.

Потрібно відзначити, що перевірка на рівність завжди строга. Значення повинні бути одного типу, щоб виконувалося рівність.

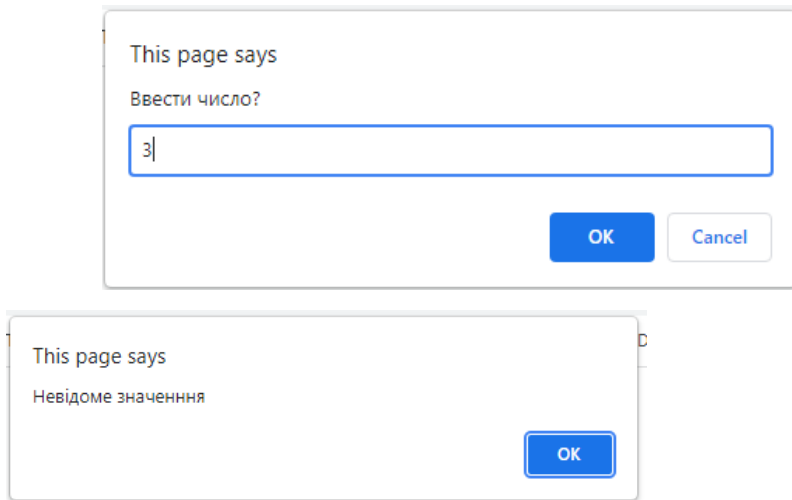
Для прикладу, давайте розглянемо наступний код:

```
let arg = prompt("Ввести число?");
switch (arg)
{ case '0':
  case '1':
    alert( 'Один або нуль' );
    break;
  case '2':
    alert( 'Два' );
    break;
  case 3:
    alert( 'Ніколи не виконається!' );
    break;
  default:
    alert( 'Невідоме значення' ); }
```

Для '0' і '1' виконається перший alert.

Для '2' - другий alert.

Але для 3, результат виконання prompt буде рядок "3", яка не відповідає стогій рівності === з числом 3. Таким чином, ми маємо «мертвий код» в case 3! Виконається варіант default.



## Функції

Найчастіше нам треба повторювати одну і ту ж дію в багатьох частинах програми.

Наприклад, необхідно красиво вивести повідомлення при вітанні відвідувача, при виході відвідувача з сайту, ще де-небудь.

Щоб не повторювати один і той же код в багатьох місцях, придумані функції. Функції є основними «будівельними блоками» програми.

### Оголошення функції

Для створення функцій ми можемо використовувати оголошення функції.

Приклад оголошення функції:

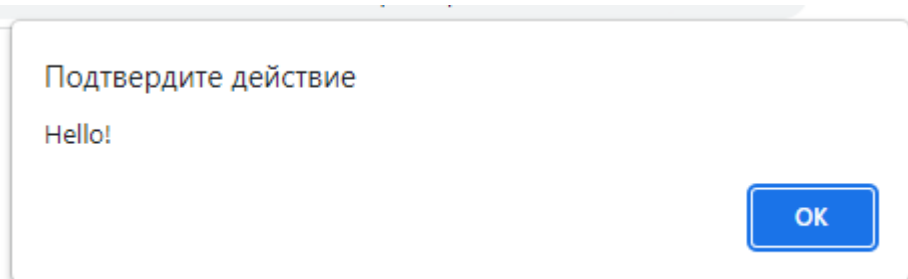
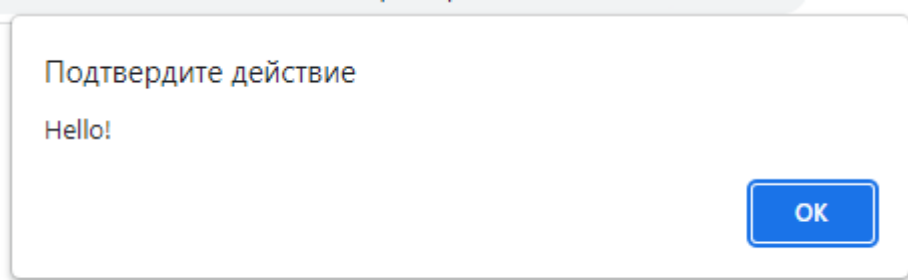
```
function showMessage() {  
    alert('Hello!');  
}  
  
function ім'я(параметри) {  
    ...тіло...  
}
```

Наша нова функція може бути викликана за її іменем: `showMessage ()`.

Наприклад:

```
function showMessage() {  
    alert( 'Hello!' );  
}
```

```
showMessage();  
showMessage();
```



Виклик `showMessage ()` виконує код функції. Тут ми побачимо повідомлення двічі.

Цей приклад явно демонструє одне з головних призначень функцій: позбавлення від дублювання коду.

Якщо знадобиться поміняти повідомлення або спосіб його виведення - досить змінити його в одному місці: в функції, яка його виводить.

### Локальні змінні

Змінні, оголошені всередині функції, видно тільки всередині цієї функції.

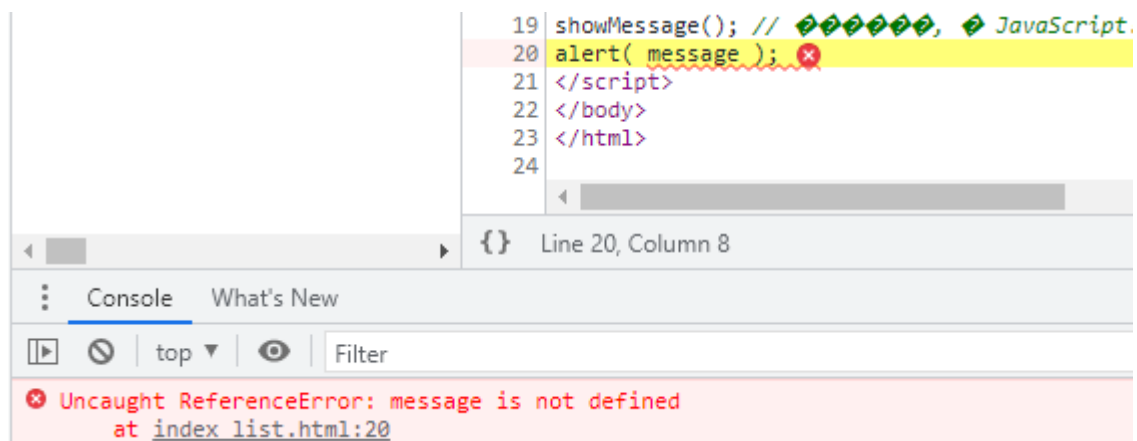
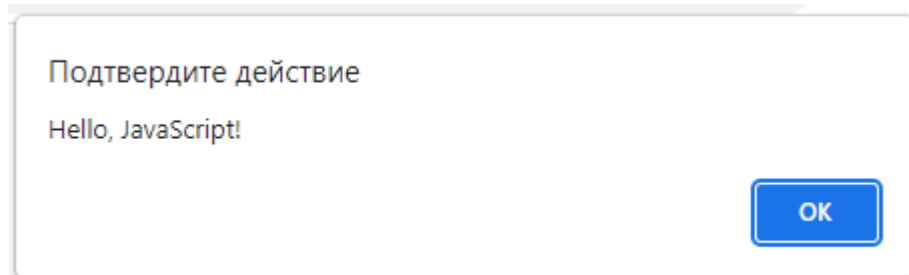
Наприклад:

```
function showMessage() {  
    let message = "Hello, JavaScript!"; // локальна змінна  
    alert( message );
```

```

}
showMessage(); Hello, JavaScript!
alert( message ); //
<помилка, область видимості змінної тільки в середині функції

```



### Зовнішні змінні

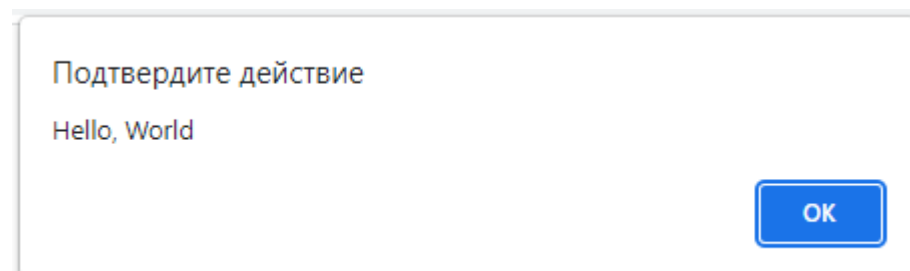
У функції є доступ до зовнішніх змінних, наприклад:

```

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.

Зовнішня змінна використовується, тільки якщо всередині функції немає такої локальної.

Якщо однойменна змінна оголошується всередині функції, тоді вона перекриває зовнішню.

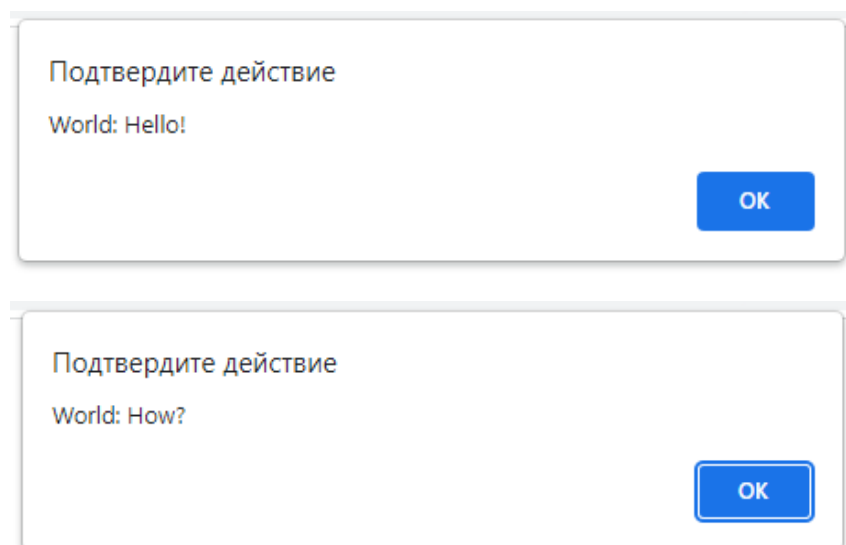
### Параметри

Ми можемо передати всередину функції будь-яку інформацію, використовуючи параметри (також звані аргументами функції).

У нижчеподаному прикладі функції передаються два параметри: `from` і `text`.

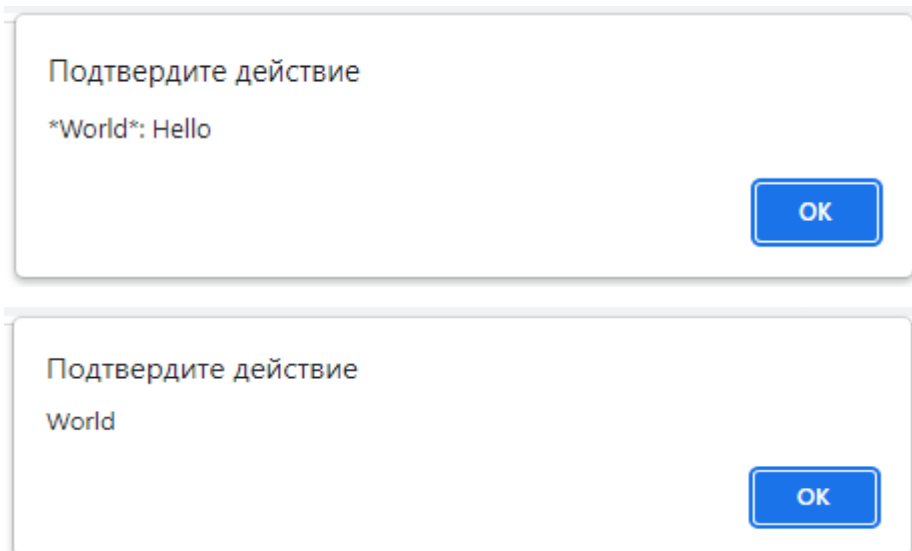
```
function showMessage(from, text) { // аргументи: from, text
    alert(from + ': ' + text);
}
showMessage('World', 'Hello!'); // World: Hello! (*)
showMessage('World', 'How?'); // World : How? (**)
```

Коли функція викликається в рядках (\*) і (\*\*), передані значення копіюються в локальні змінні `from` і `text`. Потім вони використовуються в тілі функції.



Ось ще один приклад: у нас є змінна `from`, і ми передаємо її функції. Зверніть увагу: функція змінює значення `from`, але цю змінну не видно зовні. Функція завжди отримує тільки копію значення:

```
function showMessage(from, text) {  
  from = '*' + from + '*'; // змінимо "from"  
  alert(from + ': ' + text); }  
let from = "World";  
showMessage(from, "Hello");  
// значення "from" залишилося тим самим, функція змінила значення  
локальної змінної  
alert(from);
```



### Параметри за замовчуванням

Якщо параметр не вказано, то його значенням стає `undefined`.

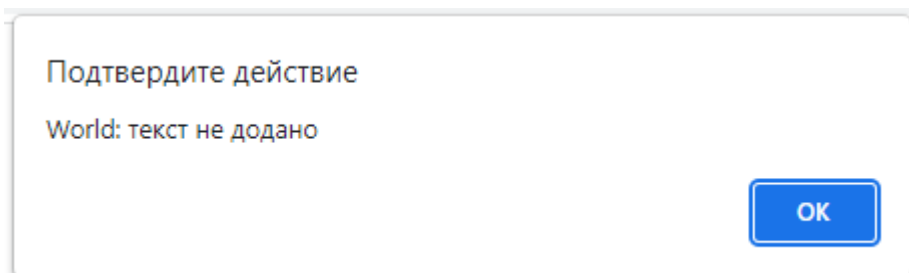
Наприклад, вищезгадана функція `showMessage (from, text)` може бути викликана з одним аргументом:

```
showMessage("World");
```

Це не призведе до помилки. Такий виклик виведе "Аня: undefined". У виклику не вказано параметр `text`, тому передбачається, що `text === undefined`.

Якщо ми хочемо задати параметру text значення за замовчуванням, ми повинні вказати його після =:

```
function showMessage(from, text = "текст не додано") { alert( from +  
": " + text ); }  
showMessage("World");
```



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. наприклад:

```
function showMessage(from, text = anotherFunction()) {  
  // anotherFunction() виконається тільки, якщо не передано text,  
  результатом буде значення text }
```

### *Обчислення параметрів за замовчуванням*

В JavaScript параметри за замовчуванням обчислюються кожен раз, коли функція викликається без відповідного параметра.

В наведеному вище прикладі anotherFunction () буде викликатися кожен раз, коли showMessage () викликається без параметра text.

### Повернення значення

Функція може повернути результат, який буде переданий в код, який її викликав.

Найпростішим прикладом може служити функція складання двох чисел:

```
function sum(a, b) {  
    return a + b;  
}  
  
let result = sum(1, 2);  
alert( result ); // 3
```

This page says

3

OK

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` вище).

Можливо використовувати `return` і без значення. Це призведе до негайного виходу з функції.

Наприклад:

```
function showText(age) {  
    if ( !checkAge(age) ) {  
        return;  
    }  
  
    alert( "Текст" ); // (*)  
    // ...  
}
```

У коді вище, якщо `checkAge (age)` поверне `false`, `showMovie` не виконає `alert`.

Результат функції з порожнім `return` або без нього – `undefined`

Якщо функція не повертає значення, це все одно, як якщо б вона повертала `undefined`.

## **Function Expression**

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).

Він виглядає ось так:

```
let sayHi = function() {  
    alert( "hello" );  
};
```

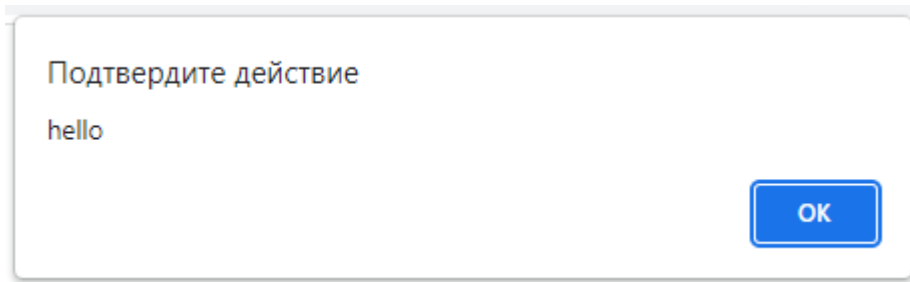
У коді вище функція створюється і явно присвоюється змінній, як будь-яке інше значення. По суті все одно, як ми визначили функцію, це просто значення, збережене в змінній sayHi.

Звичайно, функція - не звичайне значення, в тому сенсі, що ми можемо викликати його за допомогою дужок: sayHi ().

Але все ж це значення. Тому ми можемо робити з ним те ж саме, що і з будь-яким іншим значенням.

Ми можемо скопіювати функцію в іншу змінну:

```
function sayHi() { // (1) створюємо  
    alert("hello"); }  
let func = sayHi; // (2) копіюємо  
func(); // hello // (3) викликаємо копію  
sayHi(); // hello // попередня також працює
```



### Функції-«колбеки»

Розглянемо ще приклади функціональних виразів і передачі функції як значення.

Давайте напишемо функцію `ask (question, yes, no)` з трьома параметрами:

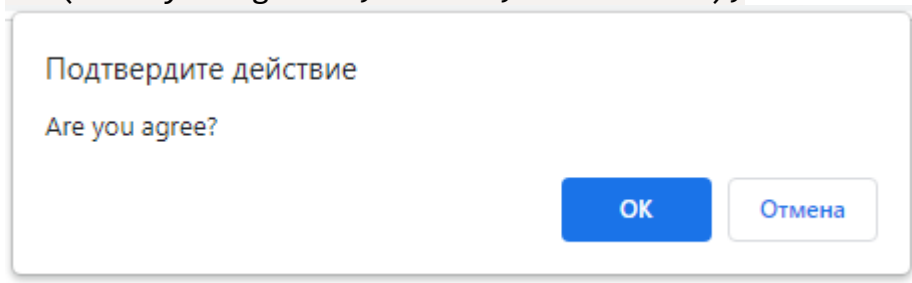
`Question` - текст питання

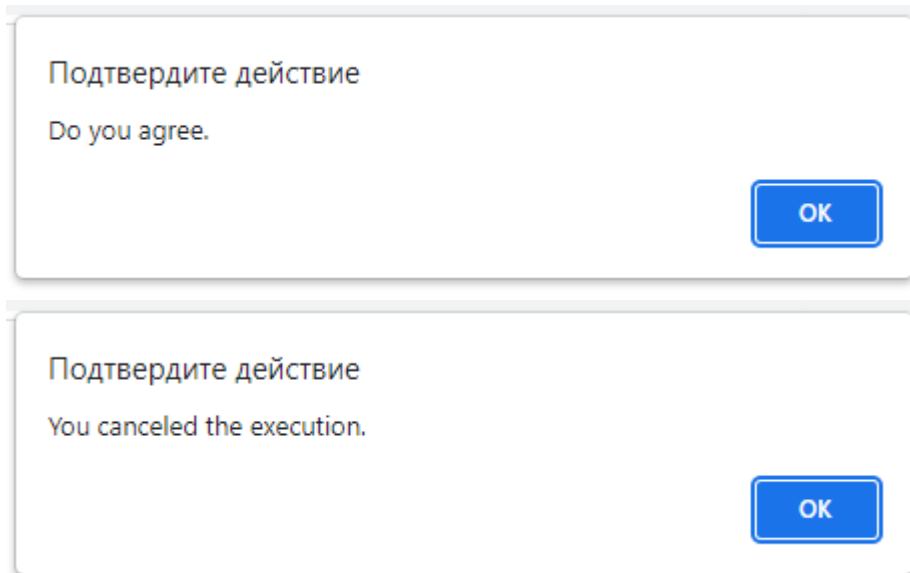
`Yes` - функція, яка буде викликатися, якщо відповідь буде «Yes»

`No` - функція, яка буде викликатися, якщо відповідь буде «No»

Наша функція повинна задати питання `question` і, в залежності від того, як відповідь користувач, викликати `yes ()` або `no ()`:

```
function ask(question, yes, no) {  
  if (confirm(question)) yes()  
  else no();  
}  
function showOk() {  
  alert("Do you agree.");  
}  
function showCancel() {  
  alert("You canceled the execution.");  
}  
// Використання: функції showOk, showCancel передаються як аргументи  
ask("Are you agree?", showOk, showCancel);
```





На практиці подібні функції дуже корисні. Основна відмінність «реальної» функції `ask` від прикладу вище буде в тому, що вона використовує більш складні способи взаємодії з користувачем, ніж простий виклик `confirm`. У браузерах такі функції зазвичай відображають красиві діалогові вікна.

Аргументи функції `ask` ще називають функціями-колбеками або просто колбеками.

Ключова ідея в тому, що ми передаємо функцію і очікуємо, що вона викликається назад пізніше, якщо це буде необхідно. У нашому випадку, `showOk` стає колбеком 'для відповіді «yes», а `showCancel` - для відповіді «no».

Ми можемо переписати цей приклад значно коротше, використовуючи Function Expression:

```
function ask(question, yes, no) {  
  if (confirm(question)) yes()  
  else no();  
}  
ask(  
  "Are you agree?",  
  function () {alert("Do you agree.");},  
  function () {alert("Do you agree.");},  
);
```

```
function () {alert("You canceled the execution.");}  
);
```

Тут функції оголошуються прямо всередині виклику ask (...). У них немає імен, тому вони називаються анонімними. Такі функції недоступні зовні ask (бо вони не присвоєні змінним), але це якраз те, що нам потрібно.

### Функції-стрілки

Існує ще більш простий і короткий синтаксис для створення функцій, який краще, ніж синтаксис Function Expression.

Він називається «функції-стрілки» (arrow functions), тому що виглядає наступним чином:

```
let func = (arg1, arg2, ...argN) => expression
```

Такий код створює функцію func з аргументами arg1..argN і обчислює expression праворуч від їх використання, повертаючи результат.

Іншими словами, це більш короткий варіант такого запису:

```
let func = function(arg1, arg2, ...argN) {  
  return expression;  
};
```

Давайте поглянемо на конкретний приклад:

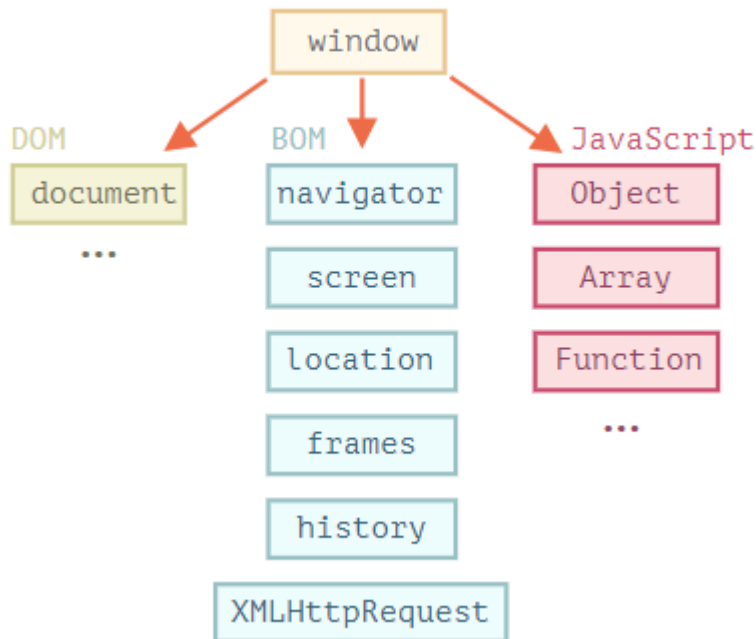
```
let sum = (a, b) => a + b;  
/* Більш коротка форма для:  
let sum = function(a, b) {  
  return a + b;  
};  
*/  
alert( sum(1, 2) ); // 3
```

Тобто, (a, b) => a + b задає функцію з двома аргументами a і b, яка при запуску обчислює вираз праворуч a + b і повертає його результат.



Розглянемо роботу зі сторінкою - як отримувати елементи, маніпулювати їхніми розмірами, динамічно створювати інтерфейси і взаємодіяти з відвідувачем.

На зображенні нижче в загальних рисах показано, що є для JavaScript у браузерному оточенні:



Як ми бачимо, є кореневий об'єкт `window`, який виступає в 2 ролях:

1. По-перше, це глобальний об'єкт для JavaScript-коду.
2. По-друге, він також є вікном браузера і має в своєму розпорядженні методи для управління ним.

Наприклад, тут ми використовуємо `window` як глобальний об'єкт:

```
function sayHi() {  
    alert("Hello");  
}  
// глобальні функції доступні як методи глобального об'єкта:  
window.sayHi();
```

А тут ми використовуємо `window` як об'єкт вікна браузера, щоб дізнатися його висоту:

```
alert(window.innerHeight); // внутрення висота окна браузера
```

Існує набагато більше властивостей і методів для управління вікном браузера.

### DOM (Document Object Model)

Document Object Model, скорочено DOM - об'єктна модель документа, яка представляє весь вміст сторінки у вигляді об'єктів, які можна змінювати.

Об'єкт **document** - основна «вхідна точка». 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
```

### BOM (Browser Object Model)

Об'єктна модель браузера (Browser Object Model, BOM) - це додаткові об'єкти, що надаються браузером (оточенням), щоб працювати з усім, крім документа.

Наприклад:

- 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.

Ось як ми можемо використовувати об'єкт location:

```
alert(location.href); // показує поточний URL
if (confirm("go to the site? {
```

```
location.href = "https://html.spec.whatwg.org"; // перенаправляє  
браузер на іншу URL }  
}
```

Всі ці об'єкти доступні за допомогою JavaScript, ми можемо використовувати їх для зміни сторінки.

Наприклад, **document.body** - об'єкт для тега <body>.

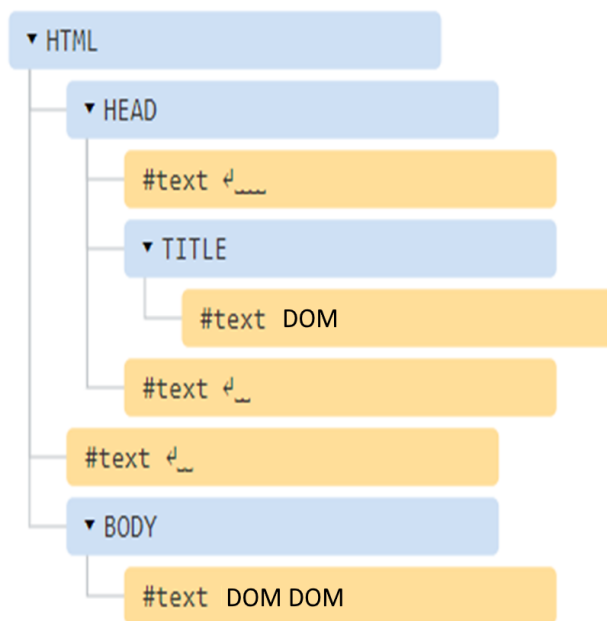
Якщо запустити цей код, то <body> стане червоним на 3 секунди:

```
document.body.style.background = 'red';  
setTimeout(() => document.body.style.background = '', 3000);
```

Приклад DOM

```
<html>  
<head>  
  <title>DOM</title>  
</head>  
<body>  
  DOM DOM  
</body>  
</html>
```

DOM - це уявлення HTML-документа у вигляді дерева тегів. Ось як воно виглядає:



Кожен вузол цього дерева - це об'єкт.

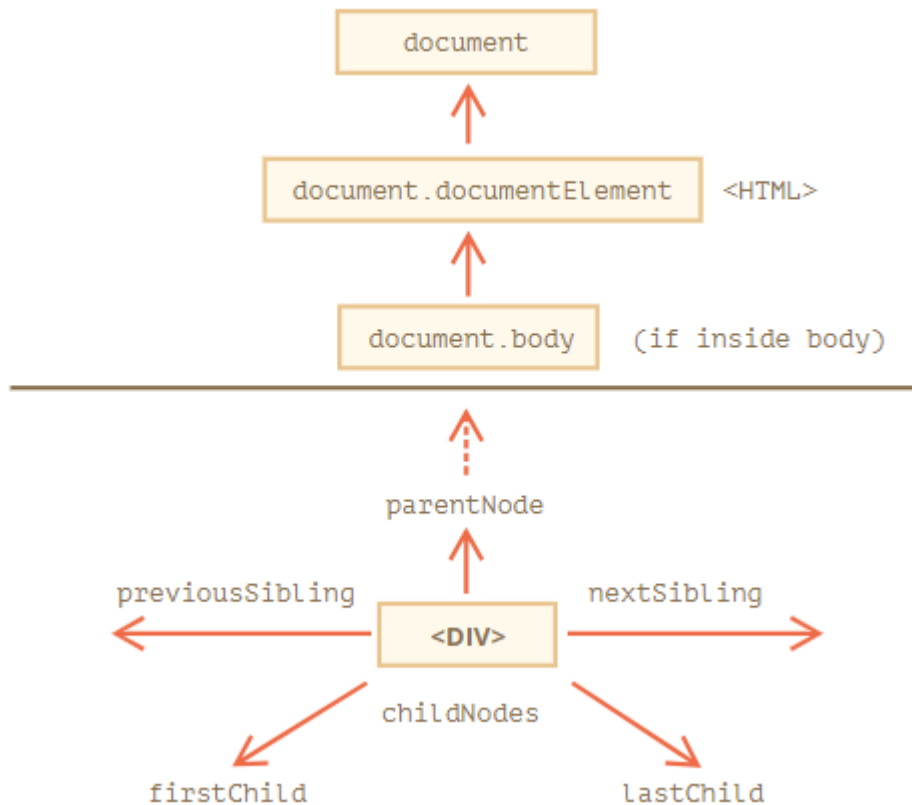
Теги є вузлами-елементами (або просто елементами). Вони утворюють структуру дерева: `<html>` - це кореневий вузол, `<head>` і `<body>` його дочірні вузли і т.д.

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.

### Навігація по DOM-елементів

Всі операції з DOM починаються з об'єкта `document`. Це головна «точка входу» в DOM. З нього ми можемо отримати доступ до будь-якого вузла.

Так виглядають основні посилання, за якими можна переходити між вузлами DOM:



Зверху: **documentElement** і **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**.

Є одна тонкість: `document.body` може дорівнювати `null`

Не можна отримати доступ до елемента, якого 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( "3 HEAD: " + document.body ); // null, <body> ще нема
  </script>
</head>
<body>
  <script>
    alert( "3 BODY: " + document.body ); // HTMLBodyElement, він є
  </script>
</body>
</html>
```

Діти: `childNodes`, `firstChild`, `lastChild`

Тут і далі ми будемо використовувати два принципово різних терміни:

Дочірні вузли (або діти) - елементи, які є безпосередніми дітьми вузла. Іншими словами, елементи, які лежать безпосередньо всередині даного. Наприклад, `<head>` і `<body>` є дітьми елемента `<html>`.

Нащадки - все елементи, які лежать всередині даного, включаючи дітей, їхніх дітей і т.д.

Колекція **`childNodes`** містить список всіх дітей, включаючи текстові вузли.

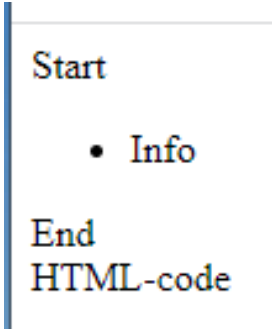
Приклад нижче послідовно виведе дітей `document.body`:

```
<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>

```



Властивості **firstChild** і **lastChild** забезпечують швидкий доступ до першого і останнього дочірньому елементу.

Вони, по суті, є всього лише скороченнями. 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 ()**.

## DOM-колекції

Як ми вже бачили, **childNodes** схожий на масив. Насправді це не масив, а колекція – об’єкт перебору.

І є два важливих наслідки з цього:

1. Для перебору колекції ми можемо використовувати **for..of**:

```

for (let node of document.body.childNodes) {
    alert(node); // покаже всі вузли з колекції
}

```

Це працює, тому що колекція є об'єктом перебору(є необхідний для цього метод `Symbol.iterator`).

2. Методи масивів не працюватимуть, бо колекція - це не масив:

```
alert(document.body.childNodes.filter); // undefined (y  
колекції нема метода filter!)
```

Перший пункт - це добре для нас. Другий - буває незручний, але можна пережити. 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

DOM-колекції - тільки для читання

Майже всі DOM-колекції, за невеликим винятком, живі. Іншими словами, вони відображають поточний стан DOM.

Якщо ми збережемо посилання на **`elem.childNodes`** і додамо / видалимо вузли в DOM, то вони з'являться в збереженій колекції автоматично.

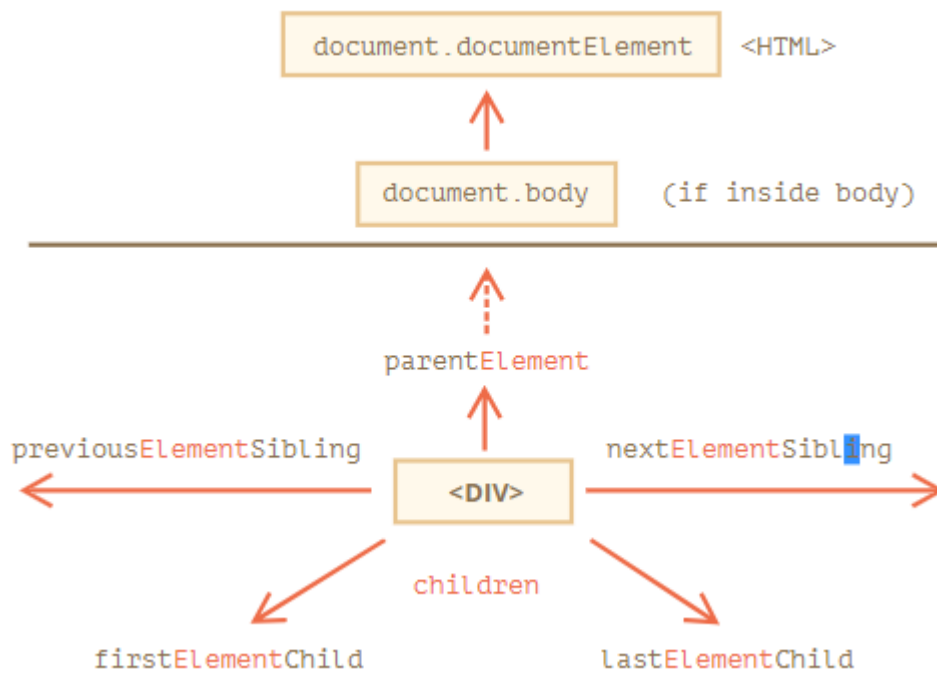
### Навігація тільки за елементами

Навігаційні властивості, описані вище, відносяться до всіх вузлів в документі. Зокрема, в `childNodes` знаходяться і текстові вузли і вузли-елементи і вузли-коментарі, якщо вони є.

Але для більшості завдань текстові вузли і вузли-коментарі нам не потрібні. Ми хочемо маніпулювати вузлами-елементами, які представляють собою теги і формують структуру сторінки.

Тому давайте розглянемо додатковий набір посилань, які враховують тільки вузли-елементи:





Ці посилання схожі на ті, що раніше, тільки в кількох місцях додається слово **Element**:

**children** - колекція дітей, які є елементами.

**firstElementChild**, **lastElementChild** - перший і останній дочірній елемент.

**previousElementSibling**, **nextElementSibling** - сусіди-елементи.

**parentElement** - батько-елемент.

Властивість **parentElement** повертає батько-елемент, а **parentNode** повертає «будь-якого батька». Зазвичай ці властивості однакові: вони обидві отримують батька.

За винятком **document.documentElement**:

```

alert( document.documentElement.parentNode ); // виведе document
alert( document.documentElement.parentElement ); // виведе null
  
```

Причина в тому, що батьком кореневого вузла **document.documentElement** (**<html>**) є **document**. Але **document** - це не вузол-елемент, так що **parentNode** поверне його, а **parentElement** немає.

Змінімо один із прикладів вище: замінімо **childNodes** на **children**. Тепер цикл виводить тільки елементи:

```

<html>
<body>
  <div>Початок</div>

  <ul>
    <li>Інфо</li>
  </ul>

  <div>Кінець</div>

  <script>
    for (let elem of document.body.children) {
      alert(elem); // DIV, UL, DIV, SCRIPT
    }
  </script>
  ...
</body>
</html>

```

### Пошук: getElement \*, querySelector \*

Властивості навігації по DOM хороші, коли елементи розташовані поруч. А що, якщо ні? Як отримати довільний елемент сторінки?

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.

Наприклад:

```

<table id="table">
  <tr>
    <td>один</td><td>два</td>
  </tr>
  <tr>
    <td>три</td><td>чотири</td>
  </tr>
</table>

<script>

```

```
// виводить вміст першого рядка
alert( table.rows[0].cells[1].innerHTML ) // "два"
</script>
```

Значення `id` має бути унікальним. У документі може бути тільки один елемент з даними `id`.

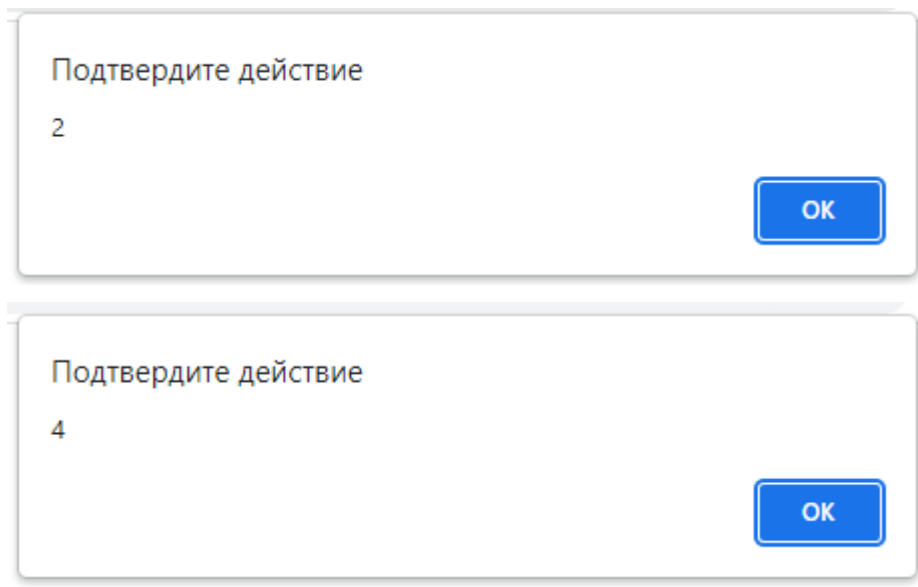
Метод `getElementById` можна викликати тільки для об'єкта `document`. Він здійснює пошук по `id` по всьому документу.

## **querySelectorAll**

Самий універсальний метод пошуку - це **`elem.querySelectorAll(css)`**, він повертає всі елементи всередині `elem`, що задовольняє даному CSS-селектору.

Наступний запит отримує всі елементи `<li>`, які є останніми нащадками в `<ul>`:

```
<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>
```



- 1
- 2
- 3
- 4

Цей метод дійсно потужний, тому що можна використовувати будь-який CSS-селектор. **querySelectorAll** повертає статичну колекцію. Це схоже на фіксований масив елементів.

Псевдокласи теж працюють

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.

### **querySelector**

Метод **elem.querySelector(css)** повертає перший елемент, який відповідає цьому CSS-селектору.

Інакше кажучи, 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. Це швидше, крім того, його коротше писати.

### **matches**

Попередні методи шукали по DOM.

Метод **elem.matches (css)** нічого не шукає, а перевіряє, чи задовольняє **elem** CSS-селектору, і повертає **true** або **false**.

Цей метод зручний, коли ми перебираємо елементи (наприклад, в масиві або в чомусь подібному) і намагаємося вибрати ті з них, які нас цікавлять.

Наприклад:

```
<a href="http://example.com/1111">...</a>
<a href="http://ii.ua">...</a>

<script>
  // може бути будь-яка колекція замість document.body.children
  for (let elem of document.body.children) {
    if (elem.matches('a[href$="zip"]')) {
      alert("Link: " + elem.href );
    }
  }
</script>
```

### **getElementsBy\***

Існують також інші методи пошуку елементів по тегу, класу і так далі.

На даний момент, вони швидше історичні, так як **querySelector** більш ніж ефективний.

Тут ми розглянемо їх для повноти картини, також ви можете зустріти їх в старому коді.

**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.

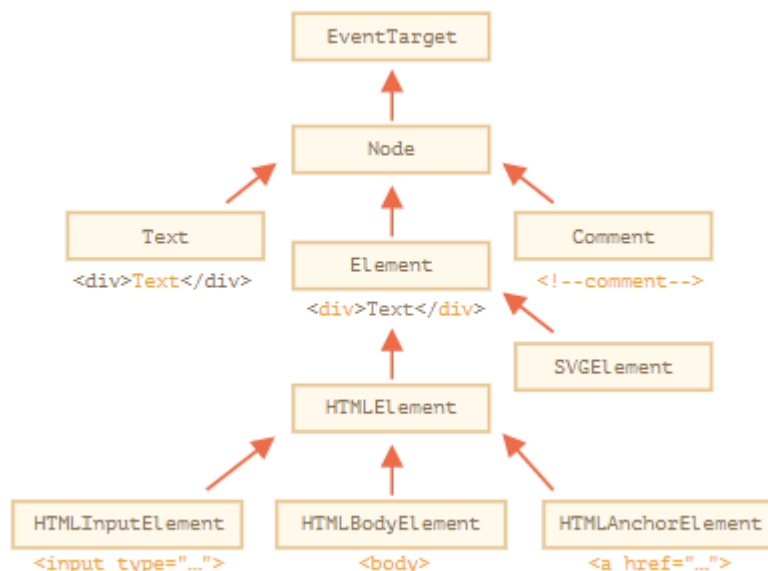
**document.getElementsByTagName(name)** повертає елементи з заданим атрибутом name.

### Класи DOM-вузлів

У різних DOM-вузлів можуть бути різні властивості. Наприклад, у вузла, відповідного тегу <a>, є властивості, пов'язані з посиланнями, а у відповідного 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 успадковує Node і інші DOM-вузли.



Існують наступні класи:

**EventTarget** - це кореневий «абстрактний» клас. Об'єкти цього класу ніколи не створюються. Він служить основою, завдяки якій всі DOM-вузли підтримують так звані «події», про які ми поговоримо пізніше.

**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. Але є певні класи вузлів,

які успадковують від нього: **Text** - для текстових вузлів, **Element** - для вузлів-елементів та **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.

Розглянемо DOM-об'єкт для тега `<input>`. Він належить до класу **HTMLInputElement**.

Він отримує властивості і методи з (в порядку спадкування):

**HTMLInputElement** - цей клас надає специфічні для елементів форми властивості,

**HTMLElement** - надає загальні для HTML-елементів методи (і геттери / сеттери),

**Element** - надає типові методи елемента,

**Node** - надає загальні властивості DOM-вузлів,

**EventTarget** - забезпечує підтримку подій,

... і, нарешті, він успадковується від **Object**, тому доступні також методи «звичайного об'єкта», такі як `hasOwnProperty`.

Для того, щоб дізнатися ім'я класу DOM-вузла, згадаємо, що 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 спадкування можна також за допомогою `instanceof`:

```
alert( document.body instanceof HTMLBodyElement ); // true
alert( document.body instanceof HTMLElement ); // true
alert( document.body instanceof Element ); // true
alert( document.body instanceof Node ); // true
alert( document.body instanceof EventTarget ); // true
```

### Тег: `nodeName` і `tagName`

Отримавши DOM-вузол, ми можемо дізнатися ім'я його тега з властивостей `nodeName` і `tagName`:

Наприклад:

```
alert( document.body.nodeName ); // BODY
alert( document.body.tagName ); // BODY
```

Чи є якась різниця між `tagName` і `nodeName`?

Так, вона відображена в назвах властивостей, але не очевидна.

- Властивість `tagName` є тільки у елементів `Element`.
- Властивість `nodeName` визначено для будь-яких вузлів `Node`:
  1. для елементів воно дорівнює `tagName`.
  2. для інших типів вузлів (текст, коментар і т.д.) воно містить рядок з типом вузла.

### **innerHTML**: вміст елемента

Властивість `innerHTML` дозволяє отримати HTML-вміст елемента у вигляді рядка.

Ми також можемо змінювати його.

Приклад нижче показує вміст `document.body`, а потім повністю замінює його:

```
body>
```



```
<p>Параграф</p>
<div>DIV</div>
```

```
<script>
  alert( document.body.innerHTML ); // поточний вміст
  document.body.innerHTML = 'Новый BODY!'; // заміна вмісту
</script>
```

```
</body>
```

```
<body>
```

```
<p>Text</p>
```

```
<div>DIV</div>
```

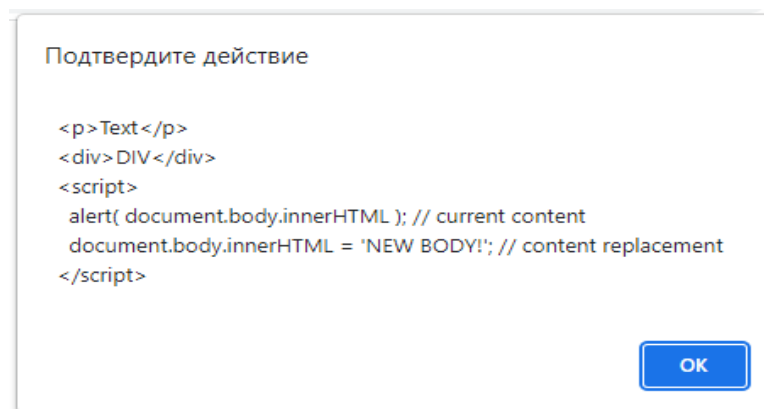
```
<script>
```

```
  alert( document.body.innerHTML ); // current content
```

```
  document.body.innerHTML = 'NEW BODY!'; // content replacement
```

```
</script>
```

```
</body>
```



NEW BODY!

Якщо `innerHTML` вставляє в документ тег `<script>` - він стає частиною HTML, але не запускається.

Будьте уважні: «**innerHTML** + `=`» здійснює перезапис

Ми можемо додати HTML до елемента, використовуючи **elem.innerHTML** + `= "ще html"`.

Ось так:

```
chatDiv.innerHTML += "<div>Hello<img src='1.gif'/> !</div>";
```

```
chatDiv.innerHTML += "How are you?";
```

На практиці цим слід користуватися з великою обережністю, так як фактично відбувається не додавання, а перезапис.

Іншими словами, `innerHTML +=` робить наступне:

1. Старий вміст видаляється.
2. На його місце стає нове значення `innerHTML` (з доданим рядком).

### **outerHTML: HTML елемента цілком**

Властивість `outerHTML` містить HTML елемента цілком. Це як `innerHTML` плюс сам елемент.

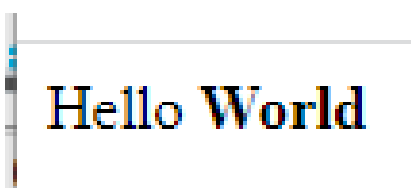
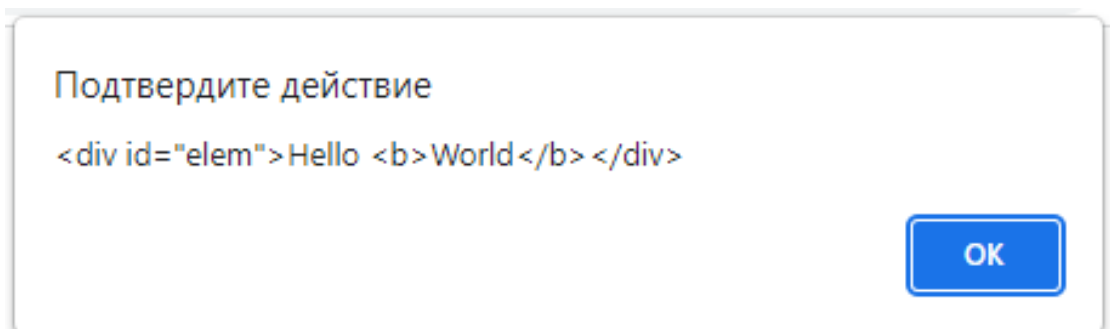
Подивимося на приклад:

```
<div id="elem">Hello <b>World</b></div>
```

```
<script>
```

```
  alert(elem.outerHTML); // <div id="elem">Hello <b>World</b></div>
```

```
</script>
```



### **nodeValue / data: вміст текстового вузла**

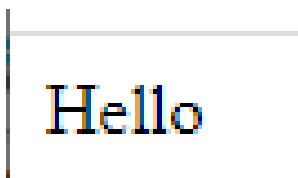
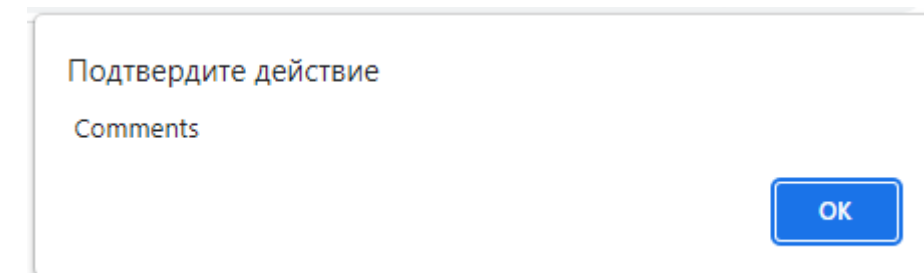
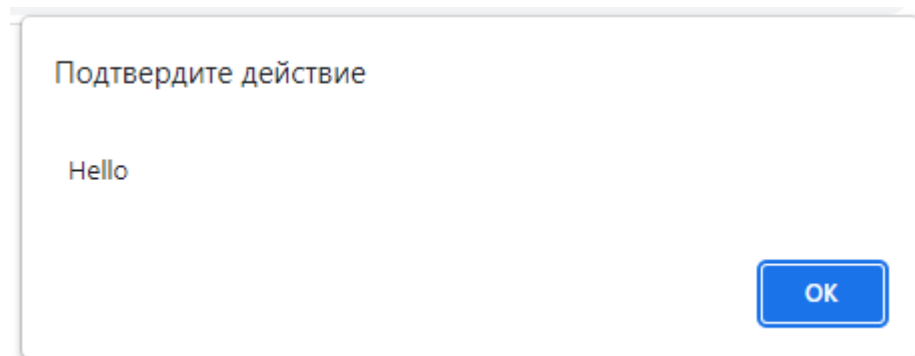
Властивість `innerHTML` є тільки у вузлів-елементів.

У інших типів вузлів, зокрема, у текстових, є свої аналоги: властивості `nodeValue` і `data`. Ці властивості дуже схожі при використанні, є лише невеликі

відмінності в специфікації. Ми будемо використовувати `data`, тому що вона коротше.

Прочитаємо вміст текстового вузла і коментаря:

```
<body>
  Hello
  <!-- Comments -->
  <script>
    let text = document.body.firstChild;
    alert(text.data); // Hello
    let comment = text.nextSibling;
    alert(comment.data); // Comments
  </script>
</body>
```

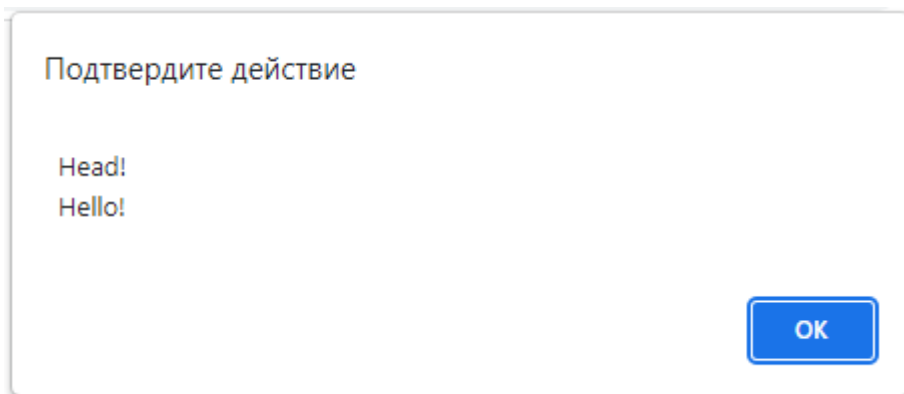


### **textContent: просто текст**

Властивість `textContent` надає доступ до тексту всередині елемента за вирахуванням всіх `<тегів>`.

Наприклад:

```
<div id="news">
  <h1>Head!</h1>
  <p>Hello!</p>
</div>
<script>
  // Hello!
  alert(news.textContent);
</script>
```



Як ми бачимо, повертається тільки текст, ніби всі `<теги>` були вирізані, але текст в них залишився.

На практиці рідко з'являється необхідність читати текст таким чином.

Набагато корисніше можливість записувати текст в `textContent`, тому що дозволяє писати текст «безпечним способом».

Уявімо, що у нас є довільний рядок, введений користувачем, і ми хочемо показати його.

З **innerHTML** вставка відбувається «як HTML», з усіма HTML-тегами.

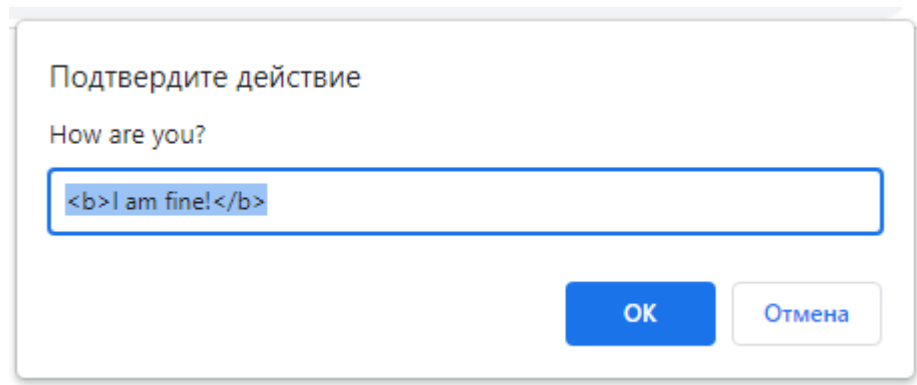
З **textContent** вставка виходить «як текст», все символи трактуються буквально.

Порівняємо два тега `div`:

```

<div id="elem1"></div>
<div id="elem2"></div>
<script>
  let name = prompt("How are you?", "<b>I am fine!</b>");
  elem1.innerHTML = name;
  elem2.textContent = name;
</script>

```



**I am fine!**  
 <b>I am fine!</b>

У першому <div> ім'я приходить «як HTML»: всі теги стали саме тегами, тому ми бачимо ім'я, виділене жирним шрифтом.

У другому <div> ім'я приходить «як текст», тому ми бачимо <b> Вінні-пух! </b>.

### Властивість «hidden»

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, як в прикладі нижче:

```

<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>
```

Both DIV's at the bottom are invisible

Технічно, hidden працює так само, як `style = "display: none"`. Але його застосування простіше.

Мигаючий елемент:

```
<div id="elem"> Flashing element </div>
<script>
  setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

Flashing element

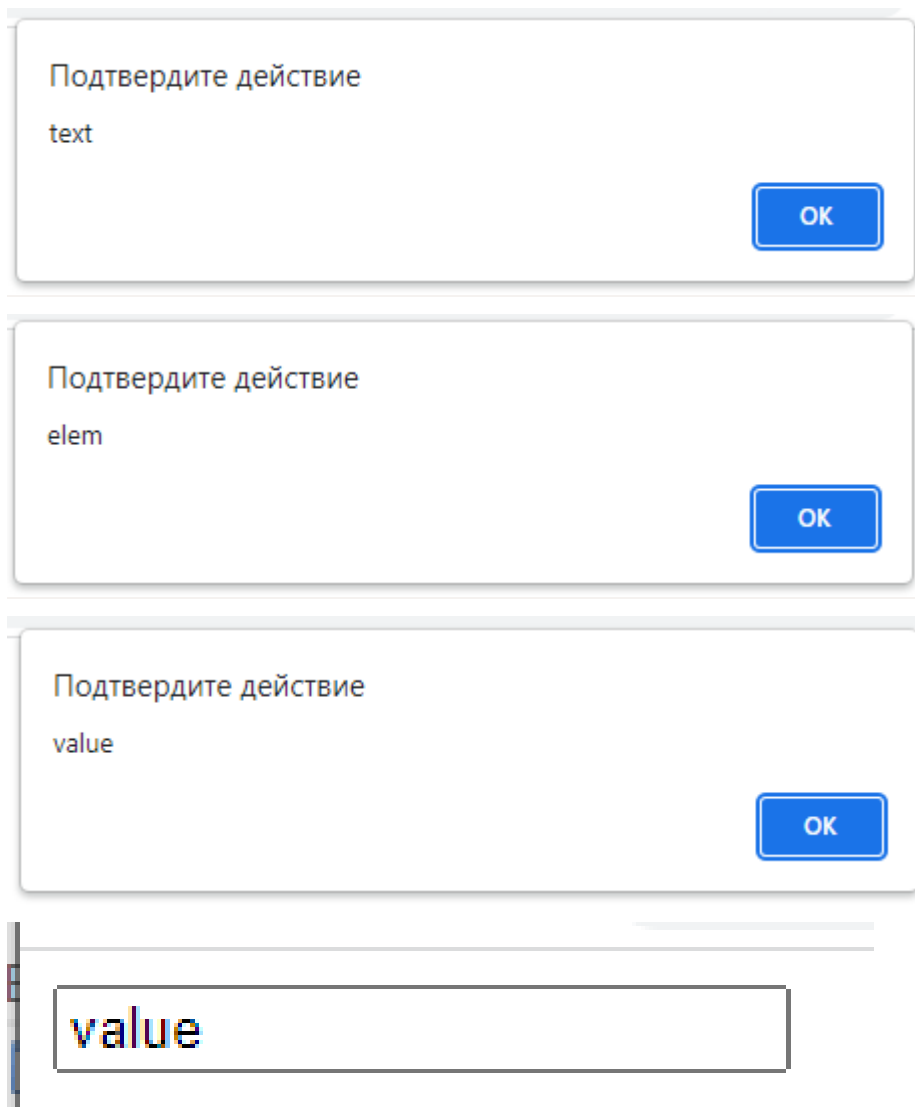
Інші властивості

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`).
- ...і багато інших...

Наприклад:

```
<input type="text" id="elem" value="value">
<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // Value
</script>
```



Більшість стандартних HTML-атрибутів мають відповідні DOM-властивості і ми можемо отримати до них доступ.

### 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.

Наприклад, для такого тега `<body id = "page">` у DOM-об'єкта буде така властивість `body.id = "page"`.

Але перетворення атрибута в властивість відбувається не один-в-один!

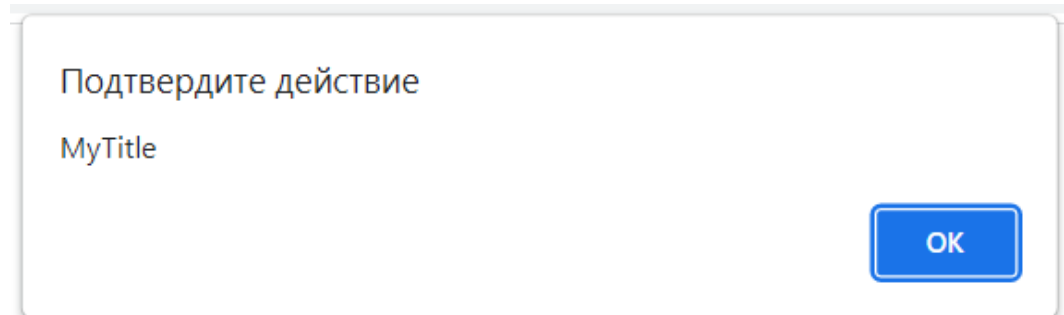
## DOM-властивості.

Раніше ми вже бачили вбудовані DOM-властивості. Їх багато. Але технічно нас ніхто не обмежує, і якщо цього мало - ми можемо додати свою власну властивість.

DOM-вузли - це звичайні об'єкти JavaScript. Ми можемо їх змінювати.

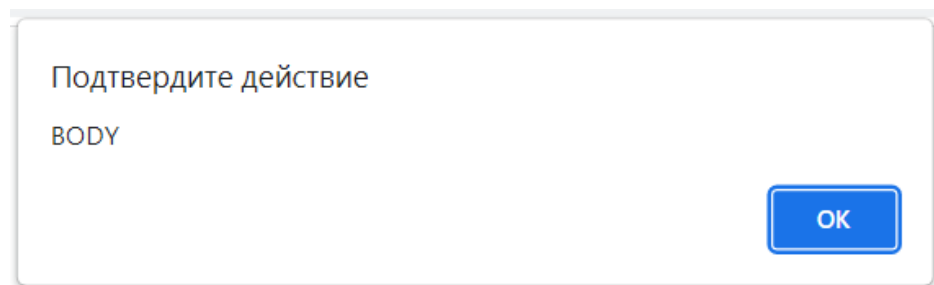
Наприклад, створимо нову властивість для `document.body`:

```
document.body.myData = {  
  name: 'MyName',  
  title: 'MyTitle' };  
alert(document.body.myData.title); // MyTitle
```



Ми можемо 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 поведуться так само, як і звичайні об'єкти JavaScript:



Їм можна присвоїти будь-яке значення.

Вони чутливі до регістру (потрібно писати `elem.nodeType`, а не `elem.NoDeType`).

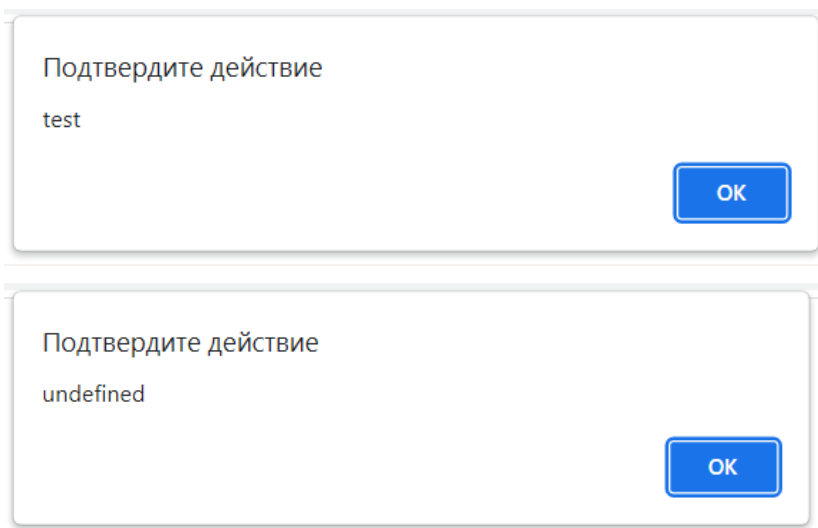
### HTML-атрибути

В HTML у тегів можуть бути атрибути. Коли браузер парсить HTML, щоб створити DOM-об'єкти для тегів, він розпізнає стандартні атрибути і створює DOM-властивості для них.

Таким чином, коли у елемента є `id` або інший стандартний атрибут, створюється відповідне властивість. Але цього не відбувається, якщо атрибут нестандартний.

Наприклад:

```
<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. Наприклад, атрибут `"type"` є стандартним для елемента `<input>`

(HTMLInputElement), але не є стандартним для <body> (HTMLBodyElement). Стандартні атрибути описані в специфікації для відповідного класу елемента.

Таким чином, для нестандартних атрибутів не буде відповідних DOM-властивостей. Чи є спосіб отримати такі атрибути?

Всі 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**.

Ось демонстрація читання нестандартної властивості:

```
<body something="non-standard">
  <script>
    alert(document.body.getAttribute('something')); //non-standard
  </script>
</body>
```

У HTML-атрибутів є такі особливості:

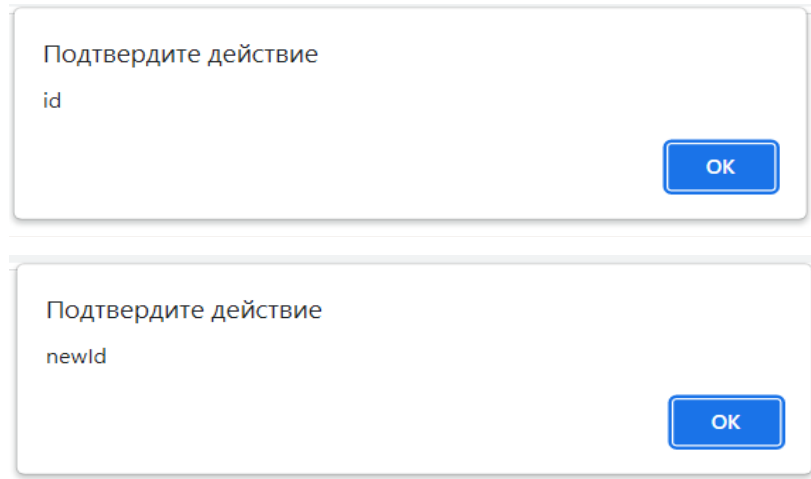
- їх імена регістронезалежні (id те ж саме, що і ID).
- їх значення завжди є рядками.

Синхронізація між атрибутами і властивостями

Коли стандартний атрибут змінюється, відповідна властивість автоматично оновлюється. Це працює і у зворотний бік (з деякими винятками).

У прикладі нижче id модифікується як атрибут, і можна побачити, що властивість також змінено. Те ж саме працює і у зворотний бік:

```
(input.id); // id
input.id = 'newId'; // властивість => атрибут
alert(input.getAttribute('id')); // newId
</script>
```

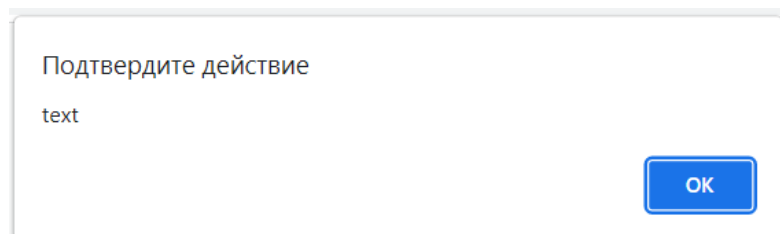


Подтвердите действие  
id

Подтвердите действие  
newId

Але є й винятки, наприклад, `input.value` синхронізується тільки в один бік - атрибут → значення, але не в зворотній:

```
<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 оновилось)
</script>
```



Подтвердите действие  
text

Подтвердите действие

text

OK

newValue

В наведеному вище прикладі:

- зміна атрибута `value` оновила властивість.
- але зміна властивості не вплинула на атрибут.

### DOM-властивості типізовані

DOM-властивості 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>
```

Подтвердите действие

OK

Подтвердите действие

true

OK

---

☒ checkbox

### Зміна документа

Модифікації DOM - це ключ до створення «живих» сторінок.

Тут ми побачимо, як створювати нові елементи «на льоту» і змінювати вже існуючі.

Приклад: показати повідомлення

Розглянемо методи на прикладі - а саме, додамо на сторінку повідомлення, яке буде виглядати трохи краще, ніж alert.

Ось таке:

```
<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>
```



**Hello!** Message.

Це був приклад HTML. Тепер давайте створимо такий же div, використовуючи JavaScript (припускаємо, що стилі в HTML або в зовнішньому CSS-файлі).

### Створення елемента

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');
```

### Створення повідомлення

У нашому випадку повідомлення - це div з класом alert і HTML в ньому:

```
let div = document.createElement('div');
```

```
div.className = "alert";
```

```
div.innerHTML = "<strong> Hello! </strong> Message.";
```

Ми створили елемент, але поки він тільки в змінній. Ми не можемо бачити його на сторінці, оскільки він не є частиною документа.

### Методи вставки

Щоб наш div з'явився, нам потрібно вставити його де-небудь в document. Наприклад, в document.body.

Для цього є метод **append**, в нашому випадку: **document.body.append (div)**.

Ось повний приклад:

```
<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);
</script>

```



Ось 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.

Рядки вставляються безпечним способом, як робить це `elem.textContent`.

Тому ці методи можуть використовуватися тільки для вставки DOM-вузлів або текстових фрагментів.

А що, якщо ми хочемо вставити HTML саме «як html», з усіма тегами та іншим, як робить це `elem.innerHTML`?

### **insertAdjacentHTML / Text / Element**

З цим може допомогти іншій, досить 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 вставлено саме «як HTML».

Наприклад:

```
<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>
```

---

111

222

Так ми можемо додавати довільний HTML на сторінку.

У метода є два брати:

**`elem.insertAdjacentText (where, text)`** - такий же синтаксис, але рядок `text` вставляється «як текст», замість HTML,

**`elem.insertAdjacentElement (where, elem)`** - такий же синтаксис, але вставляє елемент `elem`.



Вони існують, в основному, щоб уніфікувати синтаксис. На практиці часто використовується тільки `insertAdjacentHTML`. Тому що для елементів і тексту у нас є методи **append / prepend / before / after** - їх швидше написати, і вони можуть вставляти як вузли, так і текст.

### Видалення вузлів

Для видалення вузла є методи **node.remove()**.

Наприклад, зробимо так, щоб наше повідомлення видалялося через секунду:

```
<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>
```

### Клонування вузлів: **cloneNode**

Як вставити ще одне подібне повідомлення?

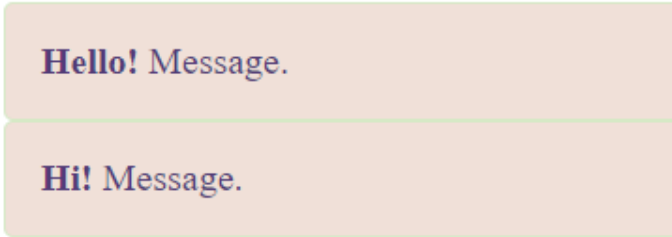
Ми могли б створити функцію і помістити код туди. Альтернатива - клонувати існуючий `div` і змінити текст всередині нього (при необхідності).

Іноді, коли у нас є великий елемент, це може бути швидше і простіше.

Виклик **elem.cloneNode (true)** створює «глибокий» клон елемента - з усіма атрибутами і дочірніми елементами. Якщо ми викличемо **elem.cloneNode (false)**, тоді клон буде без дочірніх елементів.

Приклад копіювання повідомлення:

```
<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); // клонування повідомлення
  div2.querySelector('strong').innerHTML = 'Hi!'; // зміна клонованого
елемента
  div.after(div2); // показати клонований елемент після існуючого div
</script>
```



Hello! Message.

Hi! Message.

## DocumentFragment

DocumentFragment є спеціальним DOM-вузлом, який служить обгорткою для передачі списків вузлів.

## document.write

Є ще один метод додавання вмісту на веб-сторінку: **document.write**.

```
< <p> Hi </p>  
<script>  
    document.write('<b>Message</b>');  
</script>  
<p> Hello </p>
```

---

Hi

**Message**

Hello

Виклик document.write (html) записує html на сторінку «прямо тут і зараз».

## 4. Порядок виконання лабораторної роботи

- 1) ознайомитися з теоретичними відомостями;
- 2) виконати завдання
- 3) оформити звіт, який включатиме: титульний аркуш, завдання, теоретичні відомості, результати і висновки по роботі;
- 4) продемонструвати результат на комп'ютері і захистити лабораторну роботу.

## Лабораторна робота №7

### 1. Тема

JavaScript. Події. Обробники подій. Спливання. Делегування подій.

### 2. 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. Використовувати **event.target**. Обробник **onclick** застосувати для списку, а не для кожного елементу.

Створити меню (кілька кнопок), додати один обробник для всього меню і атрибуту **data-\*** для кожної кнопки, в відповідності з методами, які вони викликають.

Застосувати прийом проектування «Поведінка» (додавання елементам поведінки **behavior** за допомогою атрибута **data-\***).

3)

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

### 3. Теоретичні відомості

Подія - це сигнал від браузера про те, що щось сталося. Все DOM-вузли подають такі сигнали (хоча події бувають і не тільки в DOM).

Ось список найбільш часто використовуваних DOM-подій, поки просто для ознайомлення:

#### 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 / покидає елемент.

**mousedown / mouseup** - коли натиснули / віджали 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 / відпускає клавішу.

#### Обробники подій

Події можна призначити обробник, тобто функцію, яка спрацює, як тільки подія відбулася.

Саме завдяки обробникам JavaScript-код може реагувати на дії користувача.

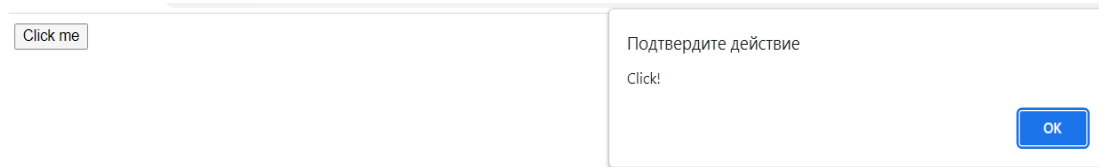
Є кілька способів призначити події обробник. Зараз ми їх розглянемо, починаючи з найпростішого.

#### 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:

```
<input value="Click me" onclick="alert('Click!')" type="button">
```



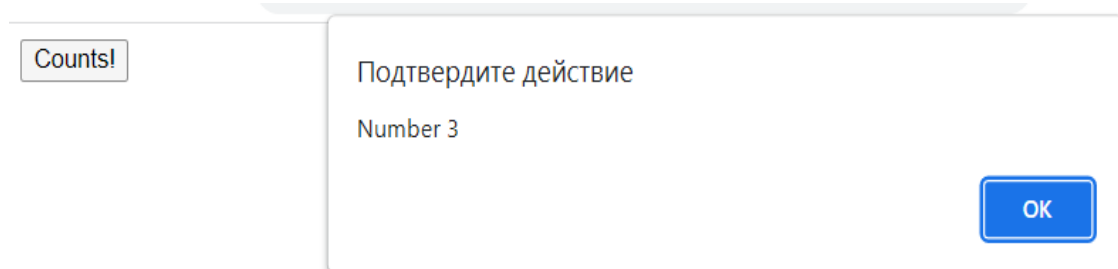
При натисканні мишкою на кнопці виконається код, вказаний в атрибуті `onclick`.

Зверніть увагу, для вмісту атрибута `onclick` використовуються одинарні лапки, так як сам атрибут знаходиться в подвійних. Якщо ми забудемо про це і поставимо подвійні лапки всередині атрибута, ось так: `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.

Наступний приклад при натисканні запускає функцію `count ()`:

```
<script>
    function count() {
        for(let i=1; i<=3; i++) {
            alert("Number " + i);
        }
    }
</script>
<input type="button" onclick="count()" value="Counts!">
```



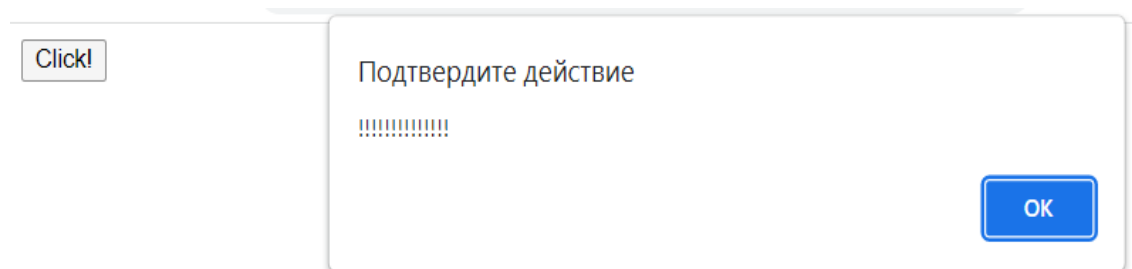
Як ми пам'ятаємо, атрибут HTML-тега не чутливий до регістру, тому `ONCLICK` буде працювати так само, як `onClick` і `onCLICK` ... Але, як правило, атрибути пишуть в нижньому регістрі: `onclick`.

## Використання властивості DOM-об'єкта

Можна призначати обробник, використовуючи властивість DOM-елемента `on <подія>`.

Наприклад, `elem.onclick`:

```
<input id="elem" type="button" value="Click!">
<script>
  elem.onclick = function() {
    alert('!!!!!!!!!!!!!!');
  };
</script>
```



Якщо обробник заданий через атрибут, то браузер читає HTML-розмітку, створює нову функцію з вмісту атрибута і записує в властивість.

Обробник завжди зберігається у властивості DOM-об'єкта, а атрибут - лише один із способів його ініціалізації.

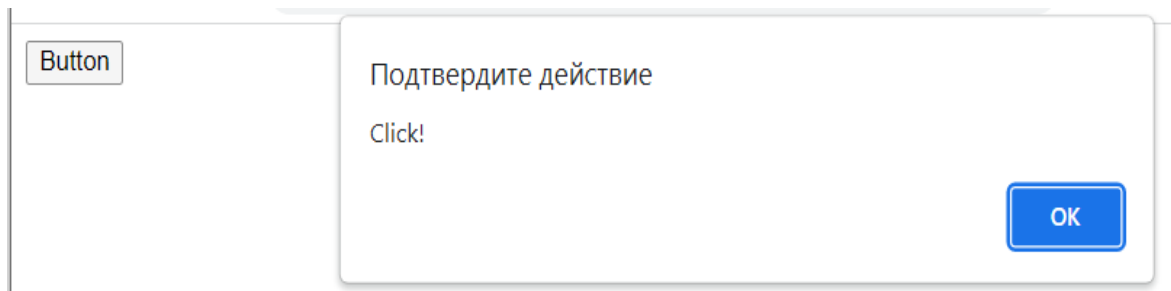
Ці два приклади коду працюють однаково:

Тільки HTML:

```
<input type="button" onclick="alert('Click!')"
value="Botton">
```

HTML + JS:

```
<input type="button" id="button" value="Button">
<script>
  button.onclick = function() {
    alert('Click!');
  };
</script>
```

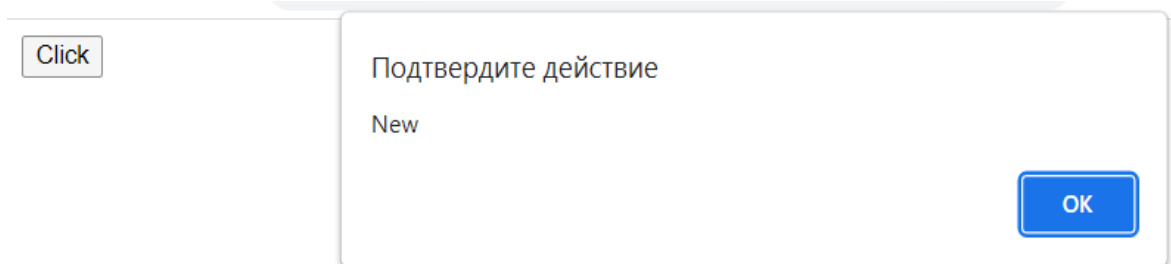


Так як у елемента DOM може бути тільки одна властивість з ім'ям `onclick`, то призначити більше одного обробника так не можна.

У прикладі нижче призначення через JavaScript перезапише обробник з атрибута:

```
<input type="button" id="elem" onclick="alert('Old')"  
value="Click">
```

```
<script>  
  elem.onclick = function() { // перезапише існуючий обробник  
    alert('New'); // виведеться тільки це  
  };  
</script>
```



До речі, обробником можна призначити і вже існуючу функцію:

```
function sayHi() {  
  alert('Hi!');  
}  
elem.onclick = sayHi;
```

Прибрати обробник можна призначенням `elem.onclick = null`.

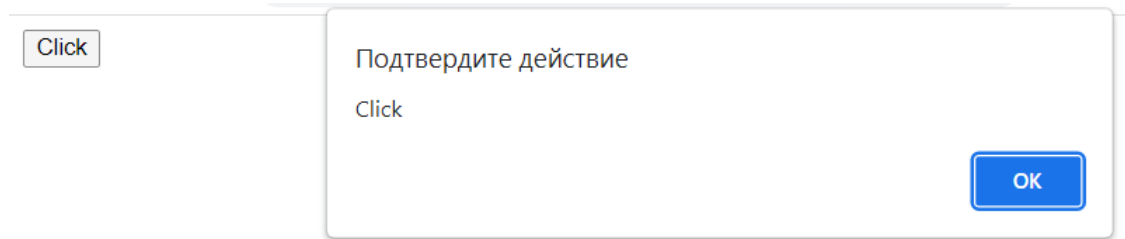
### Доступ до елемента через `this`

У середині обробника події `this` посилається на поточний елемент, тобто на той, на якому, як кажуть, «висить» (тобто призначений) обробник.



У коді нижче button виводить свій вміст, використовуючи this.innerHTML:

```
button onclick="alert(this.innerHTML)">Click</button>
```



Якщо ви тільки починаєте працювати з подіями, зверніть увагу на наступні моменти.

Функція повинна бути присвоєна як sayHi, а не sayHi ().

```
// правильно  
button.onclick = sayHi;
```

```
// неправильно  
button.onclick = sayHi();
```

Якщо додати дужки, то sayHi () - це вже виклик функції, результат якого (рівний undefined, так як функція нічого не повертає) буде присвоєно onclick. Так що це не буде працювати.

А ось в розмітці, на відміну від властивості, дужки потрібні:

```
<input type="button" id="button" onclick="sayHi()">
```

Цю різницю просто пояснити. При створенні обробника браузером з атрибута, він автоматично створює функцію з тілом зі значення атрибута: sayHi ().

Так що розмітка генерує таку властивість:

```
button.onclick = function() {  
    sayHi(); // вміст атрибута  
};
```

### **addEventListener**

Фундаментальний недолік описаних вище способів призначення обробника - неможливість повісити кілька обробників на одну подію.

Наприклад, одна частина коду хоче при кліці на кнопку робити її підсвіченою, а інша - видавати повідомлення.

Ми хочемо призначити два обробника для цього. Але нова DOM-властивість перезапише попереднє:

```
input.onclick = function() { alert(1); }  
// ...  
input.onclick = function() { alert(2); } // заміна попереднього  
обробника
```

Розробники стандартів досить давно це зрозуміли і запропонували альтернативний спосіб призначення обробників за допомогою спеціальних методів **addEventListener** і **removeEventListener**. Вони вільні від зазначеного недоліку.

Синтаксис додавання обробника:

```
element.addEventListener(event, handler[, options]);
```

**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 preventDefault () – відміна дії браузера.

Для видалення обробника слід використовувати **removeEventListener**:

```
element.removeEventListener(event, handler[, options]);
```

Видалення вимагає саме ту ж функцію

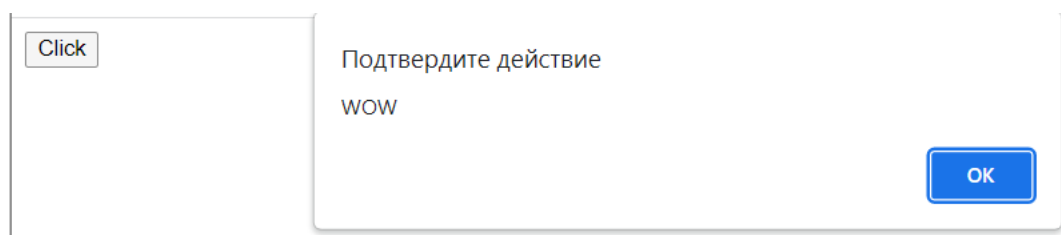
Для видалення потрібно передати саме ту функцію-обробник яка була призначена.

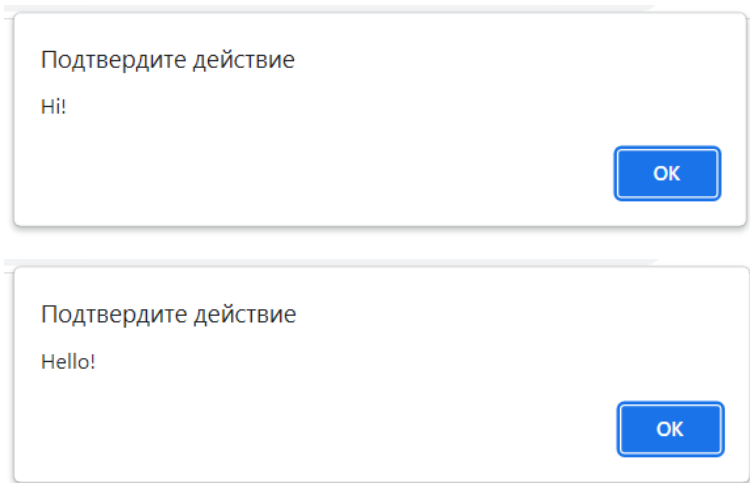
Звернемо увагу - якщо функцію обробник не зберіг де-небудь, ми не зможемо її видалити. Немає методу, який дозволяє отримати з елемента обробники подій, призначені через `addEventListener`.

```
function handler() {  
    alert( 'Hello!' ); }  
input.addEventListener("click", handler); //...  
input.removeEventListener("click", handler);
```

Метод **`addEventListener`** дозволяє додавати кілька обробників на одну подію одного елемента, наприклад:

```
<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>
```





Як видно з прикладу вище, можна одночасно призначати обробники і через DOM-властивість і через `addEventListener`. Однак, щоб уникнути плутанини, рекомендується вибрати один спосіб.

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 });
```

Так що `addEventListener` більш універсальний. Хоча зауважимо, що таких подій меншість, це швидше виняток, ніж правило.

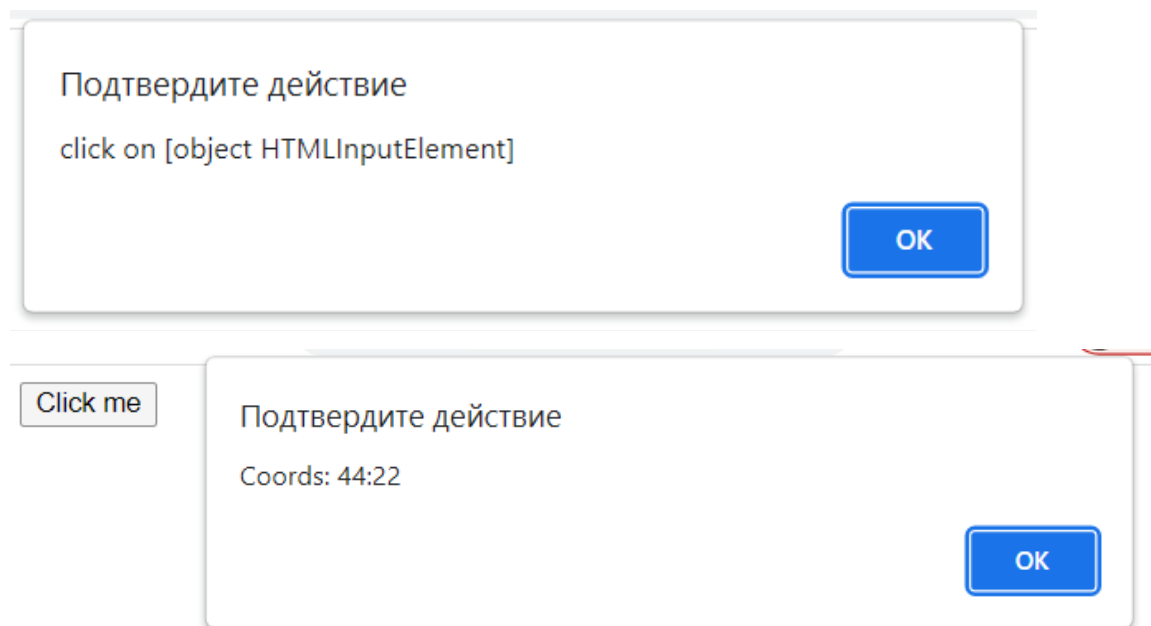
### Об'єкт події

Щоб добре обробити подію, можуть знадобитися деталі того, що сталося. Не просто «клік» або «натискання клавіші», а також - які координати покажчика миші, яка клавіша натиснута і так далі.

Коли відбувається подія, браузер створює об'єкт події, записує в нього деталі і передає його в якості аргументу функції-обробника.

Приклад нижче демонструє отримання координат миші з об'єкта події:

```
<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>
```



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**

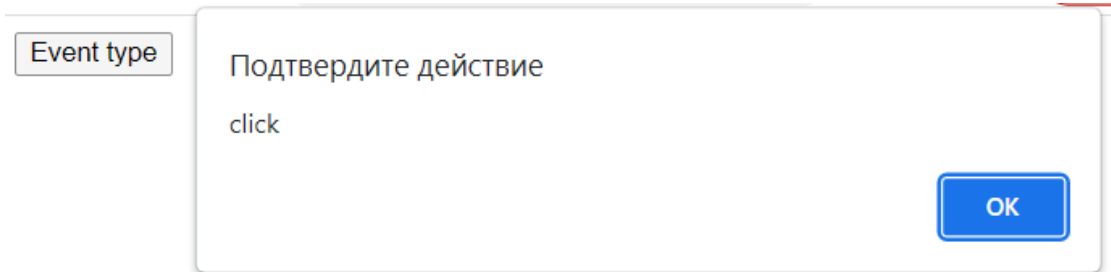
Координати курсора в момент кліка щодо вікна, для подій миші.

Є також і ряд інших властивостей, в залежності від типу подій.

Об'єкт події доступний і в HTML

При призначенні обробника в HTML, теж можна використовувати об'єкт event, ось так:

```
<input type="button" onclick="alert(event.type)" value="Event type">
```



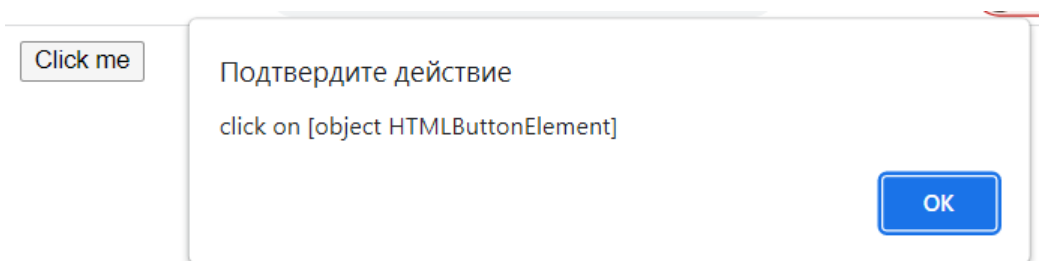
Це можливо тому, що коли браузер з атрибута створює функцію-обробник, то вона виглядає так: **function (event) {alert (event.type)}**. Тобто, її перший аргумент називається "event", а тіло взято з атрибута.

### Об'єкт-обробник: **handleEvent**

Ми можемо призначити обробником не тільки функцію, а й об'єкт за допомогою `addEventListener`. У цьому випадку, коли відбувається подія, викликається метод об'єкта `handleEvent`.

Наприклад:

```
<button id="elem">Click me</button>
<script>
  elem.addEventListener('click', {
    handleEvent(event) {
      alert(event.type + " on " + event.currentTarget);
    }
  });
</script>
```



Як бачимо, якщо `addEventListener` отримує об'єкт як обробника, він викликає `object.handleEvent(event)`, коли відбувається подія.

Ми також можемо використовувати клас для цього:

```
<button id="elem">Click</button>

<script>
  class Menu {
    handleEvent(event) {
      switch(event.type) {
        case 'mousedown':
          elem.innerHTML = "MOUSEDOWN";
          break;
        case 'mouseup':
          elem.innerHTML += "...and MOUSEUP";
          break;
      }
    }
  }

  let menu = new Menu();
  elem.addEventListener('mousedown', menu);
  elem.addEventListener('mouseup', menu);
</script>
```

---

Click

---

MOUSEDOWN...and MOUSEUP

---

MOUSEDOWN

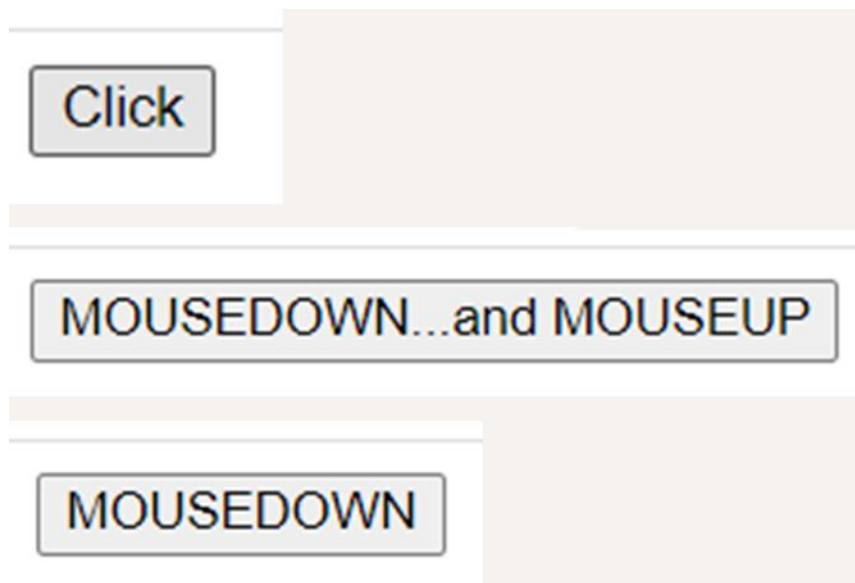
Тут один і той же об'єкт обробляє обидві події. Зверніть увагу, ми повинні явно призначити обидва обробника через **`addEventListener`**. Тоді об'єкт `menu` буде отримувати події **`mousedown`** і **`mouseup`**, але не інші (непризначення) типи подій.

Метод `handleEvent` не обов'язково повинен виконувати всю роботу сам. Він може викликати інші методи, які заточені під обробку конкретних типів подій, ось так:

Тепер обробка подій розділена по методам, що спрощує підтримку коду.

```
<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>
```





Таким чином є три способи призначення обробників подій:

Атрибут HTML: `onclick = "..."`.

DOM-властивість: `elem.onclick = function`.

Спеціальні методи: `elem.addEventListener (event, handler [, phase])` для додавання, `removeEventListener` для видалення.

HTML-атрибути використовуються рідко тому, що JavaScript в HTML-тегу Багато коду там не напишеш.

DOM-властивості цілком можна використовувати, але ми не можемо призначити більше одного обробника на один тип події.

Останній спосіб самий гнучкий, проте потрібно писати найбільше коду. Є кілька типів подій, що працюють тільки через нього, наприклад `transitionend` і `DOMContentLoaded`. Також `addEventListener` підтримує об'єкти в якості обробників подій. В цьому випадку викликається метод об'єкта `handleEvent`.

Не важливо, як ви призначаєте обробник - він отримує об'єкт події першим аргументом. Цей об'єкт містить подробиці про те, що сталося.

### **Обробка подій**

Почнемо з прикладу.

Цей обробник для `<div>` спрацює, якщо ви клікнете на одному з вкладених тегів, або `<em>` або `<code>`:

```
<div onclick="alert('Обробник!')">
```

`<em>`Якщо клікнути на `<code>ЕМ</code>`, спрацює обробник на `<code>DIV</code></em>`

```
</div>
```

Якщо клікнути на ЕМ, спрацює обробник на DIV

Подтвердите действие

Обробник!

ОК

Чому ж спрацював обробник на `<div>`, якщо клік стався на `<em>`?

## Спливання

Принцип спливання дуже простий.

Коли на елементі відбувається подія, обробники спочатку спрацьовують на ньому, потім на його батьку, потім вище і так далі, вгору по ланцюжку предків.

Наприклад, є 3 вкладених елемента `FORM > DIV > P` з обробником на кожному:

```
<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

DIV

P

Клік по внутрішньому `<p>` викличе обробник `onclick`:

1. Спочатку на самому `<p>`.
2. Потім на зовнішньому `<div>`.
3. Потім на зовнішньому `<form>`.

І так далі вгору по ланцюжку до самого `document`.

Тому якщо клікнути на `<p>`, то ми побачимо три оповіщення: `p → div → form`.

Цей процес називається «спливанням», тому що події «спливають» від внутрішнього елемента вгору через батьків.

Майже всі події спливають.

Наприклад, подія `focus` не спливає. Однак, варто розуміти, що це швидше виняток, ніж правило, все-таки більшість подій спливають.

### **event.target**

Завжди можна дізнатися, на якому конкретно елементі відбулася подія.

Найглибший елемент, який викликає подію, називається цільовим елементом, і він доступний через `event.target`.

Відмінності від `this` (`= event.currentTarget`):

**event.target** - це «цільовий» 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 «зловить» 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

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="example.css">
</head>
<body>
  Клик покажет оба: и <code>event.target</code>, и <code>this</code> для
  сравнения:
  <form id="form">FORM
    <div>DIV
      <p>P</p>
    </div>
  </form>
  <script src="script.js"></script>
</body>
</html>
```

example.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;
}
```

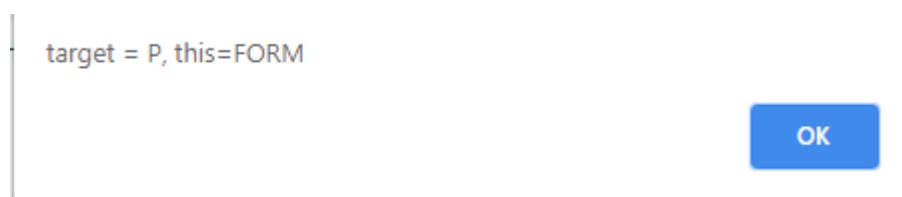
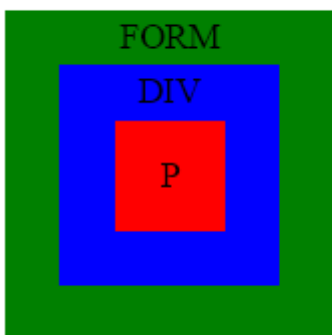
script.js

```
form.onclick = function(event) {
  event.target.style.backgroundColor = 'yellow';
```

```
// браузеру нужно некоторое время, чтобы зарисовать всё жёлтым
setTimeout(() => {
  alert("target = " + event.target.tagName + ", this=" + this.tagName);
  event.target.style.backgroundColor = ''
}, 0);
};
```

Результат

Клик покажет оба: и event.target, и this для сравнения:



Приклад:

### **index.html**

```
<HTML>
  <head>
    <link rel="stylesheet" href="mycss.css">
  </head>
  <body>
    Клік покаже обидва: i <code>event.target</code>, i
    <code>this</code> для порівняння:
    <form id="form">FORM
      <div>DIV
        <p>P</p>
      </div>
    </form>
    <script src="script.js"></script>
  </body>
</HTML>
```

### **example.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;
}
```

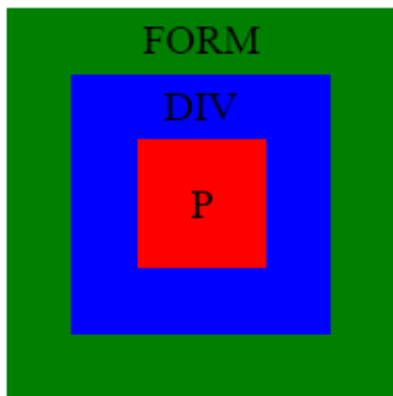
```
body {
  line-height: 25px;
  font-size: 16px;
}
```

### **script.js**

```
form.onclick = function(event) {
  event.target.style.backgroundColor = 'yellow';
  // браузеру потрібен деякий час, щоб замалювати все жовтим
  setTimeout(() => {
    alert("target = " + event.target.tagName + ", this=" +
this.tagName);
    event.target.style.backgroundColor = ''
  }, 0);
};
```

Результат

Клік покаже обидва: і `event.target`, і `this` для порівняння:



Подтвердите действие

target = P, this=FORM

OK

Подтвердите действие

target = DIV, this=FORM

OK

Подтвердите действие

target = FORM, this=FORM

OK

Можлива і ситуація, коли `event.target` і `this` - один і той же елемент, наприклад, якщо клік був безпосередньо на самому елементі `<form>`, а не на його піделементи.

### Припинення спливання



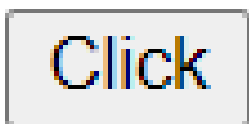
Спливання йде з «цільового» елемента прямо наверх. За замовчуванням подія буде спливати до елемента `<html>`, а потім до об'єкта `document`, а іноді навіть до `window`, викликаючи всі обробники на своєму шляху.

Але будь-який проміжний обробник може вирішити, що подія повністю оброблено, і зупинити спливання.

Для цього потрібно викликати метод `event.stopPropagation()`.

Наприклад, тут при кліці на кнопку `<button>` обробник `body.onclick` не спрацює:

```
<body onclick="alert(`сюди вспливанн не дійде`)">  
  <button onclick="event.stopPropagation()">Click</button>  
</body>
```



### Занурення

Існує ще одна фаза з життєвого циклу події - «занурення» (іноді її називають «перехоплення»). Вона дуже рідко використовується в реальному коді.

Стандарт DOM Events описує 3 фази проходу події:

1. Фаза занурення (capturing phase) - подія спочатку йде зверху вниз.
2. Фаза мети (target phase) - подія досягло цільового (вихідного) елемента.
3. 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** ось так:

```
elem.addEventListener(..., {capture: true})  
// або просто "true", як скорочення для {capture: true}
```

```
elem.addEventListener(..., true)
```

Існують два варіанти значень опції **capture**:

Якщо аргумент **false** (за замовчуванням), то подію буде спіймано при спливанні.

Якщо аргумент **true**, то подію буде перехоплено при зануренні.

### Делегування подій

Спливання і перехоплення подій дозволяють реалізувати один з найважливіших прийомів розробки - делегування.

Ідея в тому, що якщо у нас є багато елементів, події на яких потрібно обробляти схожим чином, то замість того, щоб призначати обробник кожному, ми ставимо один обробник на їх загального предка.

З нього можна отримати цільовий елемент **event.target**, зрозуміти на якому саме нащадку відбулася подія і обробити його.

Наше завдання - реалізувати підсвічування комірки `<td>` при кліці.

```
<link type="text/css" rel="stylesheet" href="mycss.css">
```

```
<table id="b-table">
```

```
<tr>
```

```
<th colspan="3">Таблиця <em>1</em>:</th>
```

```
</tr>
```

```
<tr>
```

```
<td class="nw">...<strong>1</strong>...</td>
```

```
<td class="n">...<strong>2</strong>...</td>
```

```
<td class="ne">...<strong>3</strong>...</td>
```

```
</tr>
```

```
<tr>
```

```
<td class="w">.A1.</td>
```

```
<td class="s">...</td>
```

```
<td class="e">...</td>
```

```
</tr>
```

```
<tr>
```

```

<td class="sw">.A2.</td>
<td class="s">...</td>
<td class="se">...</td>
</tr>
</table>
<script>

```

**Таблиця 1:**

...1...	...2...	...3...
.A1.	...	
.A2.	...	

Замість того, щоб призначати обробник onclick для кожної комірки <td> (їх може бути дуже багато) - ми повісимо «єдиний» обробник на елемент <table>.

Він буде використовувати **event.target**, щоб отримати елемент, на якому відбулася подія, і підсвітити його.

Код буде таким:

```

let selectedTd;
table.onclick = function(event) {
  let target = event.target; // де був клік?
  if (target.tagName !== 'TD') return; // не на TD? тоді йдемо далі
  highlight(target); // підсвітити TD
};
function highlight(td) {
  if (selectedTd) { // прибрати існуючу підсвітку, якщо вона є
    selectedTd.classList.remove('highlight');
  }
  selectedTd = td;
  selectedTd.classList.add('highlight'); // підсвітити новий td
}

```

Таблиця 1:

...1...	...2...	...3...
.A1.	...	
.A2.	...	...

Таблиця 1:

...1...	...2...	...3...
.A1.	...	
.A2.	...	...

Такому коду немає різниці, скільки клітинок в таблиці. Ми можемо додавати, видаляти `<td>` з таблиці динамічно в будь-який час, і підсвічування буде стабільно працювати.

Однак, у поточній версії коду є недолік.

Клік може бути не на тезі `<td>`, а всередині нього.

У нашому випадку, якщо поглянути на HTML-код таблиці, видно, що комірка `<td>` містить вкладені теги, наприклад `<strong>`:

```
<td>...
```

```
<strong>1</strong> ...
```

```
</td>
```

Природно, якщо клік відбудеться на елементі `<strong>`, то він стане значенням `event.target`.

Усередині обробника `table.onclick` ми повинні по `event.target` розібратися, був клік всередині `<td>` чи ні.

Ось покращений код:

```
table.onclick = function(event) {  
  let td = event.target.closest('td'); // (1)  
  if (!td) return; // (2)  
  if (!table.contains(td)) return; // (3)
```

```
highlight(td); // (4)
};
```

Розберемо приклад:

1. Метод `elem.closest (selector)` повертає найближчого предка, відповідного селектору. В даному випадку нам потрібен `<td>`, що знаходиться вище по дереву від вихідного елемента.
2. Якщо `event.target` не міститься всередині елемента `<td>`, то виклик поверне `null`, і нічого не станеться.
3. 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.
4. І якщо це так, то підсвічуються його.

У підсумку ми отримали короткий код підсвічування, швидкий і ефективний, якому абсолютно не важливо, скільки всього в таблиці `<td>`.

### Застосування делегування: дії в розмітці

Є й інші застосування делегування.

Наприклад, нам потрібно зробити меню з різними кнопками: «Зберегти (save)», «Завантажити (load)», «Пошук (search)» і т.д. І є об'єкт з відповідними методами `save`, `load`, `search` ... Як їх зістикувати?

Перше, що може прийти в голову - це знайти кожну кнопку і призначити їй свій обробник серед методів об'єкта. Але існує більш елегантне рішення. Ми можемо додати один обробник для всього меню і атрибуту **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.

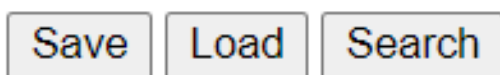
Приклад:

```
<div id="menu">
  <button data-action="save">Save</button>
  <button data-action="load">Load</button>
```

```

    <button data-action="search">Поиск</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>

```



Зверніть увагу, що метод `this.onClick` в рядку, зазначеному зірочкою (\*), прив'язується до контексту поточного об'єкта `this`. Це важливо, тому що 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.

Так що ж дає нам тут делегування?

Не потрібно писати код, щоб привласнити обробник кожній кнопці. Досить просто створити один метод і помістити його в розмітку.

Структура HTML стає по-справжньому гнучкою. Ми можемо додавати / видаляти кнопки в будь-який час.

Ми також можемо використовувати класи **.action-save**, **.action-load**, але підхід з використанням атрибутів **data-action** є більш семантично. Їх можна використовувати і для стилізації в правилах CSS.

### Прийом проектування «поведінку»

Делегування подій можна використовувати для додавання елементів «поведінки» (**behavior**), декларативно задаючи обробники установкою спеціальних HTML-атрибутів і класів.

Прийом проектування «поведінка» складається з двох частин:

1. Елементу ставиться призначений для користувача атрибут, що описує його поведінку.
2. За допомогою делегування ставиться обробник на документ, який ловить всі кліки (або інші події) і, якщо елемент має необхідний атрибут, робляє відповідну дію.

### Поведінка: «Лічильник»

Наприклад, тут HTML-атрибут `data-counter` додає кнопкам поведінку: «збільшити значення при кліці»:

```
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) { // якщо є
атрибут
            event.target.value++;
        }
    })
}
```

```
});  
</script>
```

---

Лічильник:  Ще один лічильник:

---

Лічильник:  Ще один лічильник:

Якщо натиснути на кнопку - значення збільшиться. Звичайно, нам важливі не лічильники, а загальний підхід, який тут продемонстрований.

Елементів з атрибутом **data-counter** може бути скільки завгодно. Нові можуть додаватися в HTML-код в будь-який момент. За допомогою делегування ми фактично додали новий «псевдостандартний» атрибут в HTML, який додає елементу нову можливість ( «поведінку»).

#### Поведінка: «Перемикач» (Toggler)

Ще один приклад поведінки. Зробимо так, що при кліці на елемент з атрибутом **data-toggle-id** буде ховатися / показуватися елемент із заданим id:

```
<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>
```



---

Показати форму підписки

---

Показати форму підписки

Ваша пошта:

---

Показати форму підписки

Ще раз підкреслимо, що ми зробили. Тепер для того, щоб додати приховування-розкриття будь-якого елемента, 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 використовуємо поведінку. Обробники на рівні документа зроблять це можливим для елемента в будь-якому місці сторінки.

Ми можемо комбінувати кілька варіантів поведінки на одному елементі.

#### 4. Порядок виконання лабораторної роботи

- 1) ознайомитися з теоретичними відомостями;
- 2) виконати завдання
- 3) оформити звіт, який включатиме: титульний аркуш, завдання, теоретичні відомості, результати і висновки по роботі;
- 4) продемонструвати результат на комп'ютері і захистити лабораторну роботу.

## Лабораторна робота №8

### 1. Тема

JavaScript. Події миші

### 2. Завдання

1)

Використовувати тему, обрану в 4 роботі.

Використати події **mouseover**, **mouseout** та **event.target**, **event.relatedTarget**.

При русі показчика миші над елементами, вони повинні змінювати свій стиль

Реалізувати перетягування елемента (текст/картинка/файл) з одного місця в інше, використавши **mousedown**, **mousemove**, **mouseup**.

2)

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

### 3. Теоретичні відомості

#### Події миші

Події миші і їх властивості.

Ці події бувають не тільки через мишу, але і емулюються на інших пристроях, зокрема, на мобільних, для сумісності.

#### **Типи подій миші**

Ми можемо розділити події миші на дві категорії: «прості» і «комплексні».

#### Прості події

Самі часто використовувані прості події:

**mousedown / mouseup**

Кнопка миші натиснута / відпущена над елементом.

**mouseover / mouseout**

Курсор миші з'являється над елементом і йде з нього.

**mousemove**

Кожен рух миші над елементом генерує цю подію.

### **contextmenu**

Викликається при спробі відкриття контекстного меню, як правило, натисканням правої кнопки миші. Але, зауважимо, 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**

Викликається подвійним кліком на елементі.

Комплексні події складаються з простих, тому в теорії ми могли б без них обійтися. Але добре, що вони існують, тому що працювати з ними дуже зручно.

### Порядок подій

Одна дія може викликати кілька подій.

Наприклад, клік мишкою спочатку викликає mousedown, коли кнопка натиснута, потім mouseup і click, коли вона відпущена.

У разі, коли одна дія ініціює кілька подій, порядок їх виконання фіксований. Тобто обробники подій викликаються в наступному порядку: mousedown → mouseup → click.

У вікні тесту нижче всі події миші записуються, і якщо затримка між ними більше 1 секунди, то вони поділяються горизонтальною лінією.

При цьому ми також можемо побачити властивість which, яке дозволяє визначити, яка кнопка миші була натиснута.

### Отримання інформації про кнопку: which

Події, пов'язані з кліком, завжди мають властивість `which`, яка дозволяє визначити натиснуту кнопку миші.

Ця властивість не використовується для подій `click` і `contextmenu`, оскільки перша відбувається тільки при натисненні лівої 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 `which` дозволяє розрізнити між собою «натискання правої кнопки» і «натискання лівої кнопки».

#### Є три можливих значення:

**`event.which == 1` - ліва кнопка**

**`event.which == 2` - середня кнопка**

**`event.which == 3` - права кнопка**

#### Модифікатори: `shift`, `alt`, `ctrl` і `meta`

Всі події миші включають в себе інформацію про натиснуті клавіші-модифікатори.

#### Властивості об'єкта події:

**`shiftKey`:** Shift

**`altKey`:** Alt (або Opt для Mac)

**`ctrlKey`:** Ctrl

**`metaKey`:** Cmd для Mac

Вони рівні `true`, якщо під час події була натиснута відповідна клавіша.

Наприклад, кнопка внизу працює тільки при комбінації Alt + Shift + клік:

```
<button>Alt+Shift+Click me!</button>
```

```
<script>
```

```
document.body.children[0].onclick = function(e) {
```

```
  if (!e.altKey || !e.shiftKey) return;
```

```
  alert('!!!!!!!');
```

```
}
```

```
</script>
```

Alt+Shift+Click me!

Подтвердите действие

!!!!!!!

OK

### Координати: clientX / Y, pageX / Y

Всі події миші мають координати двох видів:

Щодо вікна: clientX і clientY.

Щодо 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 (приклад знаходиться в iframe, тому координати визначаються щодо цього iframe):  
<input onmousemove="this.value = event.clientX+'-'+event.clientY" value="Point me ">

Point me

90:27

Координати щодо документа pageX, pageY відраховуються не від вікна, а від лівого верхнього кута документа.

<input onmousemove="this.value = event.pageX+'-'+event.pageY">

---

49:27

### Відключаємо виділення

Подвійний клік миші має побічний ефект, який може бути незручний в деяких інтерфейсах: він виділяє текст.

Наприклад, подвійний клік на текст нижче виділяє його в доповнення до нашого обробника:

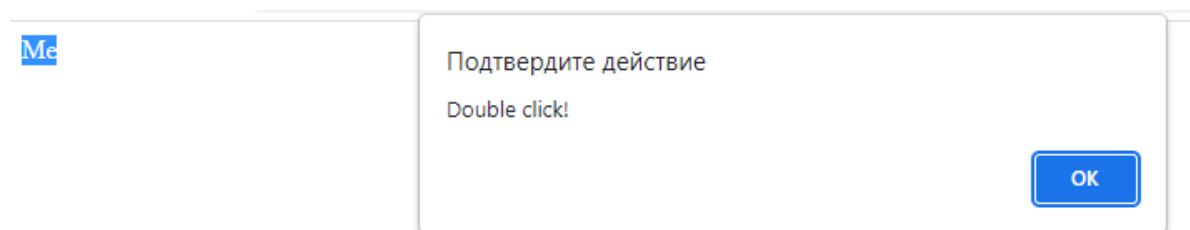
Якщо ви спробуєте скопіювати текст в `<div>`, у вас це не вийде, тому що спрацьовування події онсору за замовчуванням заборонено.

Звичайно, користувач має доступ до HTML-коду сторінки і може взяти текст звідти, але не всі знають, як це зробити.

```
<span ondblclick="alert('Double click!')">Me</span>
```

---

Me



Якщо затиснути ліву кнопку миші і, не відпускаючи кнопку, провести мишею, то також буде виділення, яке в інтерфейсах може бути «недоречно».

Є кілька способів заборонити виділення.

В даному випадку самим розумним буде скасувати дію браузера за замовчуванням при події `mousedown`, це скасує обидва цих виділення:

```
<div ondblclick="alert('Me')" onmousedown="return false">  
  "Me", без виділення  
</div>
```

"Me", без виділення

Подтвердите действие

Me

OK

Тепер виділений жирним елемент не виділяється при подвійному натисканні, а також на ньому не можна почати виділення, затиснувши кнопку миші.

Зауважимо, що текст всередині нього як і раніше можна виділити, якщо почати виділення не на самому тексті, а до нього або після. Зазвичай це нормально сприймається користувачами.

### Запобігання копіювання

Якщо ми хочемо відключити виділення для захисту вмісту сторінки від копіювання, то ми можемо використовувати іншу подію: **oncopy**.

```
<div oncopy="alert('Копіювання заборонено');return false">  
  Копіювання заборонено  
</div>
```

Копіювання заборонено

Подтвердите действие

Копіювання заборонено

OK

### Рух миші: **mouseover /out, mouseenter / leave**

Розглянемо події, що виникають при русі покажчика миші над елементами сторінки.

### Події **mouseover / mouseout, relatedTarget**

Подія **mouseover** відбувається в момент, коли курсор виявляється над елементом, а подія **mouseout** - в момент, коли курсор йде з елемента.

Ці події є особливими, бо у них є властивість **relatedTarget**. Вона «доповнює» 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).

Властивість relatedTarget може бути null.

Це нормально і означає, що покажчик миші перейшов ні з іншого елемента, а з-за меж вікна браузера. Або ж, навпаки, пішов за межі вікна.

### Пропуск елементів

Подія **mousemove** відбувається при русі миші. Однак, це не означає, що вказані подія генерується при проходженні кожного пікселя.

Браузер періодично перевіряє позицію курсора і, помітивши зміни, генерує подію **mousemove**.

Це означає, що якщо користувач рухає мишкою дуже швидко, то деякі DOM-елементи можуть бути пропущені.

Якщо курсор миші пересунути дуже швидко з елемента #FROM на елемент #TO, як це показано вище, то елементи <div> між ними (або деякі з них) можуть бути пропущені. Подія **mouseout** може запуститися на елементі #FROM і потім відразу ж згенерує **mouseover** на елементі #TO.

Це добре з точки зору продуктивності, тому що якщо проміжних елементів багато, навряд чи ми дійсно хочемо обробляти вхід і вихід для кожного.



З іншого боку, ми повинні мати на увазі, що курсор не «відвідує» всі елементи на своєму шляху. Він може і «стрибати».

Зокрема, можливо, що покажчик заплигне в середину сторінки з-за меж вікна браузера. У цьому випадку значення `relatedTarget` буде `null`, так як курсор прийшов «з нізвідки».

### Подія **mouseout** при переході на нащадка

Важлива особливість події `mouseout` - вона генерується в тому числі, коли покажчик переходить з елемента на його нащадка.

Тобто, візуально покажчик все ще на елементі, але ми отримаємо `mouseout`!

За логікою браузера, курсор миші може бути тільки над одним елементом в будь-який момент часу - над самим глибоко вкладеним і верхнім по `z-index`.

Таким чином, якщо курсор переходить на інший елемент (нехай навіть дочірній), то він залишає попередній.

Зверніть увагу на важливу деталь.

Подія `mouseover`, що відбувається на нащадку, спливає. Тому, якщо на батьківському елементі є такий обробник, то він його викличе.

### Події **mouseenter** і **mouseleave**

Події `mouseenter` / `mouseleave` схожі на `mouseover` / `mouseout`. Вони теж генеруються, коли курсор миші переходить на елемент або залишає його.

Але є і пара важливих відмінностей:

Переходи всередині елемента, на його нащадки і з них, які не рахуються.

Події `mouseenter` / `mouseleave` не спливають.

Події `mouseenter` / `mouseleave` гранично прості і зрозумілі.

Коли покажчик з'являється над елементом - генерується `mouseenter`, причому не має значення, де саме покажчик: на самому елементі або на його нащадку.

Подія `mouseleave` відбувається, коли курсор залишає елемент.

Події `mouseenter / leave` прості і легкі у використанні. Але вони не спливають. Таким чином, ми не можемо їх делегувати.

### **Drag'n'Drop з подіями миші**

Drag'n'Drop - відмінний спосіб поліпшити інтерфейс. 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 сучасному 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. Є й інші інтерфейсні задачі, які такими вбудованими подіями не реалізуються. Крім того, мобільні пристрої погано їх підтримують.

Тут ми будемо розглядати Drag'n'Drop за допомогою подій миші.

### **Алгоритм Drag'n'Drop**

Базовий алгоритм Drag'n'Drop виглядає так:

При **`mousedown`** - готуємо елемент до переміщення, якщо необхідно (наприклад, створюємо його копію).

Потім при **`mousemove`** пересуваємо 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.

У наступному прикладі ці кроки реалізовані для перенесення м'яча:

```

ball.onmousedown = function(event) { // (1) відстежити надискання
    // (2) підготувати до переміщення:
    // розмістити поверх решти вмісту та в абсолютних координатах
    ball.style.position = 'absolute';
    ball.style.zIndex = 1000;
    // перемістимо в body, щоб м'яч був точно не всередині
position:relative
    document.body.append(ball);

    // і встановимо абсолютно спозиційований м'яч під курсор
    moveAt(event.pageX, event.pageY);
// перемістити м'яч під координати курсору
// та змістити на половину ширини/висоти для центрування
function moveAt(pageX, pageY) {
    ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
    ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
}
function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
}
// (3) переміщати по екрану
document.addEventListener('mousemove', onMouseMove);

// (4) покласти м'яч, видалити непотрібні обробники подій
ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
};
};

```

Якщо запустити цей код, то ми помітимо, що при перенесенні м'яч «роздвоюється» і переноситься не саме м'яч, а його «клон».



Все тому, що браузер має свій власний Drag'n'Drop, який автоматично запускається і вступає в конфлікт з нашим. Це відбувається саме для картинок і деяких інших елементів.

Його потрібно відключити:

```
ball.ondragstart = function() {  
    return false;  
};
```

Тепер все буде в порядку.

Ще одна деталь - подія **mousemove** відстежується на **document**, а не на **ball**. З першого погляду здається, що миша завжди над м'ячем і обробник **mousemove** можна повісити на сам м'яч, а не на документ.

Подія **mousemove** виникає хоч і часто, але не для кожного пікселя. Тому через швидкий рух покажчик може зістрибнути з м'яча і опинитися де-небудь в середині документа (або навіть за межами вікна).

Ось чому ми повинні відслідковувати **mousemove** на всьому **document**, щоб зловити його.

### Правильне позиціонування

У прикладі вище м'яч позиціонується так, що його центр виявляється під покажчиком миші:

```
ball.style.left = pageX - ball.offsetWidth / 2 + 'px';  
ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
```

Непогано, але є побічні ефекти. Ми, для початку переносу, можемо натиснути мишею на будь-якому місці м'яча. Якщо м'ячик «узятий» за самий край - то на початку перенесення він різко «стрибає», центруючись під покажчиком миші.

Було б краще, якби початковий зсув курсору щодо елемента зберігався.

Де захопили, за ту «частину елемента» і переносимо:



Оновимо наш алгоритм:

1. Коли людина натискає на м'ячик (**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:

```
// onmousedown
let shiftX = event.clientX - ball.getBoundingClientRect().left;
let shiftY = event.clientY - ball.getBoundingClientRect().top;
```

2. Далі при перенесенні м'яча ми позиціонуємо його з тим же зрушенням щодо покажчика миші, ось так:

```
// onmousemove
// ball has position:absolute
ball.style.left = event.pageX - shiftX + 'px';
ball.style.top = event.pageY - shiftY + 'px';
```

Підсумковий код з правильним позиціонуванням:

```
ball.onmousedown = function(event) {
    let shiftX = event.clientX - ball.getBoundingClientRect().left;
    let shiftY = event.clientY - ball.getBoundingClientRect().top;
    ball.style.position = 'absolute';
    ball.style.zIndex = 1000;
    document.body.append(ball);

    moveAt(event.pageX, event.pageY);
```

```
// переносить м'яч на координати (pageX, pageY),
```

```
// Додатково враховуючи початковий зсув відносно вказівника миші
```

```

function moveAt(pageX, pageY) {
    ball.style.left = pageX - shiftX + 'px';
    ball.style.top = pageY - shiftY + 'px';
    function onMouseMove(event) {
        moveAt(event.pageX, event.pageY);
    }
    // переміщуємо м'яч по події mousemove
    document.addEventListener('mousemove', onMouseMove);
    // відпустити м'яч, видалити непотрібні обробники подій
    ball.onmouseup = function() {
        document.removeEventListener('mousemove', onMouseMove);
        ball.onmouseup = null;
    };
};
ball.ondragstart = function() {
    return false; };

```

Якщо захопити м'яч за правий нижній кут. У попередньому прикладі м'ячик «стрибне» серединою під курсор, в цьому - буде плавно переноситися з поточної позиції.

### **Цілі (мішені) перенесення (droppable)**

У попередніх прикладах м'яч можна було кинути просто де завгодно в межах вікна. В реальності ми зазвичай беремо один елемент і перетягуємо в інший. Наприклад, «файл» в «папку» або щось ще.

Абстрактно кажучи, ми беремо draggable елемент і поміщаємо його в інший елемент «ціль перенесення» (droppable).

Нам потрібно знати:

- куди користувач поклав елемент в кінці перенесення, щоб обробити його закінчення

- і, бажано, над якою потенційною мішенню (елемент, куди можна покласти, наприклад, зображення папки) він знаходиться в процесі перенесення, щоб підсвітити її.

Розглянемо рішення.

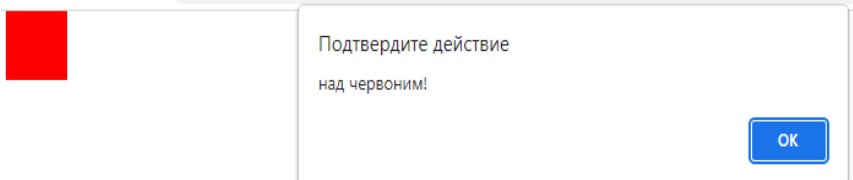
Можливо, встановити обробники подій **mouseover** / **mouseup** на елемент - потенційну мішень перенесення?

Але це не працює.

Проблема в тому, що при переміщенні переміщуваний елемент завжди знаходиться поверх інших елементів. А події миші спрацьовують тільки на верхньому елементі, але не на нижньому.

Наприклад, у нас є два елементи `<div>`: червоний поверх синього (повністю перекриває). Не вийде зловити подію на синьому, тому що червоний зверху:

```
<style>
  div {
    width: 50px;
    height: 50px;
    position: absolute;
    top: 0;
  }
</style>
<div style="background:blue" onmouseover="alert('ніколи не
спрацює')"></div>
  <div style="background:red" onmouseover="alert('над
червоним!')"></div>
```



Те ж саме з перетягуванням елементів. М'яч завжди знаходиться поверх інших елементів, тому події спрацьовують на ньому. Якби обробники ми не ставили на нижні елементи, вони не будуть виконані.

Ось чому початкова ідея поставити обробники на потенційні цілі перенесення нереалізована. Обробники не спрацюють.

Так що ж робити?

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
ball.hidden = true; // (*) ховаємо елемент, який переноситься
let elemBelow = document.elementFromPoint(event.clientX,
event.clientY);
// elemBelow - елемент під м'ячем (можлива мішень перенесення)
ball.hidden = false;
```

Зауважимо, нам потрібно заховати м'яч перед викликом функції (\*). В іншому випадку за цими координатами ми будемо отримувати м'яч, адже це і є елемент безпосередньо під покажчиком: **elemBelow = ball**. Так що ми ховаємо його і тут же показуємо назад.

Ми можемо використовувати цей код для перевірки того, над яким елементом ми «летимо», в будь-який час. І обробити закінчення перенесення, коли воно станеться.

Розширений код onMouseMove з пошуком потенційних цілей перенесення:

```
// потенційна ціль перенесення, над якою ми проходимо прямо зараз
let currentDroppable = null;
function onMouseMove(event) {
  moveAt(event.pageX, event.pageY);
  ball.hidden = true;
  let elemBelow = document.elementFromPoint(event.clientX,
event.clientY);
  ball.hidden = false;
  // подія mousemove може статися і коли покажчик за межами вікна
```



```

    // (м'яч перетягнули за межі екрану)
    // якщо clientX/clientY за межами вікна, elementFromPoint поверне
null
    if (!elemBelow) return;
    // потенційні цілі перенесення позначені класом droppable (може бути
й інша логіка)
    let droppableBelow = elemBelow.closest('.droppable');
    if (currentDroppable !== droppableBelow) {
        // ми або заходимо на ціль, або відходимо з неї
        // обидва значення можуть бути null
        // currentDroppable=null,
        // якщо ми були не над droppable до цієї події (наприклад, над
порожнім простором)
        // droppableBelow=null,
        // якщо ми не над droppable саме зараз, під час цієї події
        if (currentDroppable) {
            // логіка обробки процесу "виходу" з droppable (видаляємо
підсвічування)
            leaveDroppable(currentDroppable);
        }
        currentDroppable = droppableBelow;
        if (currentDroppable) {
            // логіка обробки процесу, коли ми "входимо" в елемент droppable
            enterDroppable(currentDroppable);
        }
    }
}
}
}

```

У наведеному нижче прикладі, коли м'яч перетягується через футбольні ворота, ворота підсвічуються.

style.css

```

#gate {
    cursor: pointer;
    margin-bottom: 100px;
}

```

```

    width: 83px;
    height: 46px;
}
#ball {
    cursor: pointer;
    width: 40px;
    height: 40px;
}
<html>
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="mycss.css">
</head>
<body>
    <p>Перемістіть м'яч.</p>
    
    
    <script>
        let currentDraggable = null;
        ball.onmousedown = function(event) {
            let shiftX = event.clientX - ball.getBoundingClientRect().left;
            let shiftY = event.clientY - ball.getBoundingClientRect().top;
            ball.style.position = 'absolute';
            ball.style.zIndex = 1000;
            document.body.append(ball);
            moveAt(event.pageX, event.pageY);

function moveAt(pageX, pageY) {
    ball.style.left = pageX - shiftX + 'px';
    ball.style.top = pageY - shiftY + 'px';
}
        function onMouseMove(event) {
            moveAt(event.pageX, event.pageY);

```

```

    ball.hidden = true;

    let elemBelow = document.elementFromPoint(event.clientX,
event.clientY);

    ball.hidden = false;

    if (!elemBelow) return;
    if (!elemBelow) return;

    let droppableBelow = elemBelow.closest('.droppable');
    if (currentDroppable !== droppableBelow) {
        if (currentDroppable) { // null если мы были не над droppable
до этого события
            // (например, над пустым пространством)
            leaveDroppable(currentDroppable);
        }
        currentDroppable = droppableBelow;
        if (currentDroppable) { // null если мы не над droppable
сейчас, во время этого события
            // (например, только что покинули droppable)
            enterDroppable(currentDroppable);
        }
    }
}

document.addEventListener('mousemove', onMouseMove);
ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
};

function enterDroppable(elem) {
    elem.style.background = 'pink';
}

```

```

function leaveDroppable(elem) {
    elem.style.background = '';
}
ball.ondragstart = function() {
    return false;
};
</script>
</body>
</html>

```

Тепер протягом всього процесу в змінній **currentDroppable** ми зберігаємо поточну потенційну ціль перенесення, над якою ми зараз, можемо її підсвітити або зробити щось ще.



## Порядок виконання лабораторної роботи

- 1) ознайомитися з теоретичними відомостями;
- 2) виконати завдання
- 3) оформити звіт, який включатиме: титульний аркуш, завдання, теоретичні відомості, результати і висновки по роботі;
- 4) продемонструвати результат на комп'ютері і захистити лабораторну роботу.

## СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Хмельюк М.С. Конспект лекцій з дисципліни «Основи клієнтської розробки»
2. Хмельюк М.С. Методичні вказівки до виконання лабораторних робіт для студентів з дисципліни «Основи клієнтської розробки»
3. Скот Чакон, Бен Штрауб Pro Git — профессиональный контроль версий, изд. – СПб.: Питер, 2019.- 496с.
4. Б. Лоусон, Р. Шарп - Изучаем HTML 5, изд: Питер, 2011
5. П. Лабберс — HTML 5 для профессионалов, изд: Вильямс, 2011
6. Бен Хеник — HTML и CSS Путь к совершенству, изд: O'Reilly (Питер), 2011. -336с
7. Дэвид Флэнаган JavaScript. Подробное руководство, изд.: Символ, 2017. - 1080
8. Інтернет-ресурс <https://learn.javascript.ru/>
9. Інтернет-ресурс <https://git-scm.com/book/ru/v2>
10. Інтернет-ресурс <http://habr.com>
11. Інтернет-ресурс <https://puzzleweb.ru/>

Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»  
Факультет інформатики та обчислювальної техніки  
Кафедра інформаційних систем та технологій

**Лабораторна робота № \_\_**  
з дисципліни «Основи клієнтської розробки»

Тема: «\_\_\_\_\_»

Виконав:  
студент групи -

Прізвище Ім'я.

Дата здачі \_\_\_\_\_

Захищено з балом \_\_\_\_\_

Перевірила:

ст. вик. кафедри ІСТ

Хмелюк Марина Сергіївна

Київ 2025