

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Електронне видання

**ОСНОВИ КЛІЄНТСЬКОЇ РОЗРОБКИ**

КОНСПЕКТ ЛЕКЦІЙ ДЛЯ СТУДЕНТІВ

СПЕЦІАЛЬНОСТІ

121, 123, 126

ОСВІТНЬОЇ ПРОГРАМИ

Інтегровані інформаційні системи

*Затверджено методичною радою ФІОТ*

Київ

КПІ ім. Ігоря Сікорського

2021

Основи клієнтської розробки: Конспект лекцій для студентів усіх форм навчання спеціальності 121, 123, 126/ Уклад.: М.С. Хмелюк. – К.: КПІ ім. Ігоря Сікорського, 2021.

*Гриф надано Методичною радою ФІОТ  
(Протокол від )*

Електронне видання  
Основи клієнтської розробки

Конспект лекцій 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*

# Лекція 1

## Мова розмітки HTML

### HTML - основи

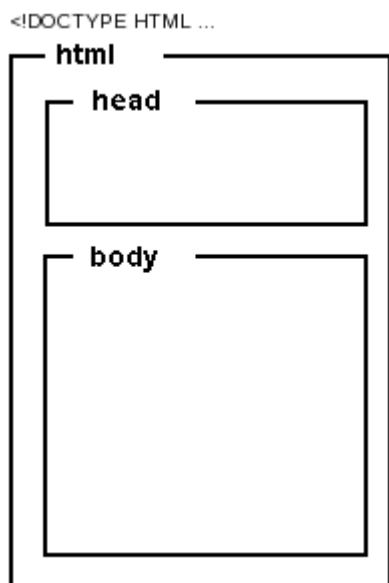
HTML - це мова розмітки, який представляє прості правила оформлення і компактний набір структурних і семантичних елементів розмітки (тегів).

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.

## 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 <>:

<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'язаній 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 Рисунок">  
</a>`

Усередині тега `<BODY>` використовується атрибут, що задає колір гіперпосилань



link - задає колір вихідних посилань

vlink - задає колір відвіданих посилань

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

```
<!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">`  `</a>`  
`</p>` - посилання на іншу сторінку того ж сайту, але посиланням є Рисунок

`<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 посилається:

`<a href="/index.html">` Посилання на сторінку в кореновому 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>`

## Лекція 2

**Кольори. Формати графічних файлів. Форматування тексту.  
Списки. Зображення. Фон**

### Колір в HTML

Комп'ютер може відобразити близько 16 мільйонів - кольорів. Альтернативним способом завдання кольору є вказівка коду кольору в системі RGB (від англ. Red, green, blue - червоний, зелений, синій). Суть системи полягає в тому, що будь-який колір може бути представлений як змішання основних кольорів - червоного, зеленого і синього. Колір записується у вигляді 6-символьного коду.

Код являє собою шістнадцяткове число від 000000 до FFFFFFFF. Перші дві цифри відповідають червоною компоненті, наступні дві - зеленої, останні дві - синьою. Значення 00 означає повну відсутність складової, значення FF (255) - максимум складової. Як шістнадцятирічних цифр використовуються десяткові цифри від 0 до 9 і латинські літери від A до F для позначення цифр від 10 до 15. Таким чином, виходить  $256^3 \approx 16.7$  млн. кольорів - цього достатньо, щоб відтворити будь-який колір, який розрізняє людське око.

Колір в HTML може бути заданий ключовими словами - назвами кольорів 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>`

Наприклад, можна набрати в пошуку запит «підбір кольору»:

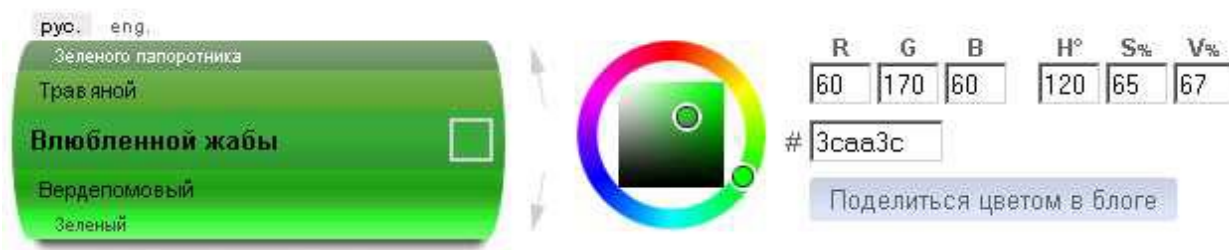


Рисунок 1 Інструмент підбору кольору

Подібні інструменти є у всіх графічних редакторах:

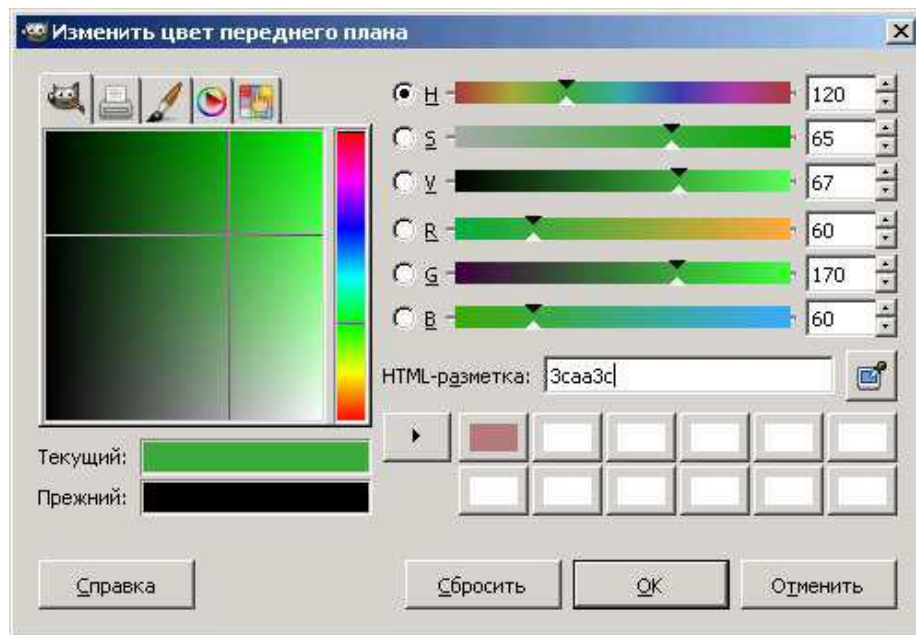


Рисунок 2. Діалог вибору кольору в редакторі GIMP.

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

### 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 <sub> 2 </sub> = 5`

У браузері:  $101_2 = 5$

HTML-код: `2 <sup> 8 </sup> = 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

1 - 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.

GIF - формат, який реалізує стиснення без втрати якості з обмеженою кольоровістю (від 2 до 256 кольорів) і підтримкою анімації - використовується для зберігання графіки, коли досить 256 (і менше) кольорів. Зазвичай це невеликі зображення. Також GIF підтримує прозорість.



JPEG реалізує стиснення зображень з втратами якості, при цьому обмеження на колір відсутні (підтримується 16 мільйонів кольорів). Розмір JPEG-файлу залежить від параметра «якість», який вказується при його збереженні: від 0 до 100. Чим вище якість, тим більше розмір файлу. Оптимальна ступінь якості залежить від зображення, в більшості випадків вона дорівнює 70-80. Не варто виставляти цей параметр менше 50 - на зображенні з'являться помітні дефекти або більше 95 - розмір файлу сильно зросте без видимого поліпшення якості.

Формат PNG існує в двох варіантах: PNG-8 і PNG-24. PNG-8, як і GIF, підтримує 256 кольорів, забезпечує в порівнянні з ним краще стиснення, але не підтримує анімацію. Формат PNG-24, як і JPEG, не має обмежень на кількість кольорів, але програє йому в розмірі файлу. Здійснює стиснення зображень без втрати якості, тому його варто застосовувати для зображень, що містять дрібні деталі.

Нижче наведена таблиця оптимального вибору формату файлу в співвідношенні якість / розмір файлу з урахуванням особливостей зображення.

Таблиця Вибір формату графічного файлу

Особливості зображення	Переважаючий формат
анімоване зображення	тільки GIF
маленьке зображення з невеликою кількістю кольорів	GIF або PNG-8
зображення з напівпрозорістю	тільки PNG
зображення з великою кількістю кольорів, наприклад фотографія	JPEG

зображення з великою кількістю кольорів з дрібними деталями, наприклад скріншот (знімок екрану)	PNG-24
---	--------

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>
```

Наприклад, зробимо фоновим зображенням сторінки рисунок bg1.jpg.

## Тест!

Рисунок 3. Файл bg1.jpg

```
<html>
```

```
<head>
```

```
<title>Тест фона</title>
```

```
</head>
```

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

```
</body>
```

```
</html>
```

В браузері:

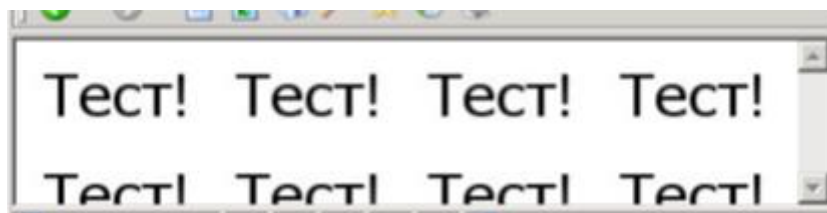


Рисунок 4. Розмноження фонового малюнка в браузері

Щоб зробити неповторюваний фон, необхідно вибрати картинку свідомо більшу, ніж розмір сторінки за шириною і висотою.

Якщо взяти картинку шириною 1 піксель і висотою, наприклад, 2000. пікселів на екрані вона буде розмножуватися тільки по горизонталі.

```
<html>
```

```
<head>
```

```
<title>Тест фона</title>
```

```
</head>
```

```
<body background="1px.gif">
```

```
<h1><font color="white" face="Arial">Фон</font></h1>
```

```
Используем картинку с градиентом!
```

```
</body>
```

```
</html>
```

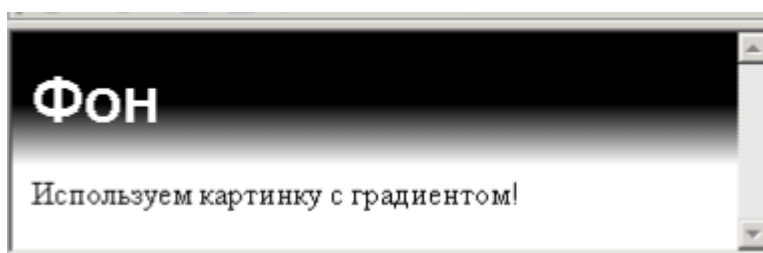


Рисунок 7. Використання зображення, що повторюється по горизонталі 1 px.

При використанні зображень в якості фону важливо забезпечити читабельність тексту на сторінці. Тому не варто застосовувати фотографії в якості фонових картинок.

## Лекція 3

### Таблиці

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 атрибуты>
  <tr>
    <td>...</td>
  </tr>
</table>
```

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

Атрибути для <col> і <colgroup>:

align - Встановлює вирівнювання вмісту колонки по краю.

span - Кількість колонок, до яких слід застосовувати атрибут.

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:

```

<table width="600">
  <thead>
    <tr>
      <td> 1111111 </td>
      <th> 2222222 </th>
    </tr>
  </thead>
  <tfoot align="center" style="background: #ffc">
    <tr>
      <td>Ячейка 1, расположенная в TFOOT</td>
      <td>Ячейка 2, расположенная в TFOOT</td>
    </tr>
  </tfoot>

  <tbody align="right" style="background: silver">
    <tr>
      <td>Ячейка 3, расположенная в TBODY</td>
      <td>Ячейка 4, расположенная в TBODY</td>
    </tr>
  </tbody>
</table>

```

Результат:

1111111	2222222
Ячейка 3, расположенная в TBODY	Ячейка 4, расположенная в TBODY
Ячейка 1, расположенная в TFOOT	Ячейка 2, расположенная в TFOOT

### 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>
```

### **Вкладені таблиці**

#### Пример

```
<TABLE BORDER="0" CELSPACING=10>
<TR><TD>
<TABLE BORDER="1">
<TR><TD>Вася</TD><TD>Петя</TD></TR>
<TR><TD>Маша</TD><TD>Даша</TD></TR>
</TABLE>
</TD>
<TD>
<TABLE BORDER="1">
<TR><TD>1</TD><TD>22</TD></TR>
<TR><TD>333</TD><TD>4444</TD></TR>
</TABLE>
</TD></TR>
```

</TABLE>

### Додавання заголовка таблиці

Заголовок таблиці можна створити за допомогою відомих вам тегів <h1> - <h6> Але оскільки ширина таблиці може відрізнятися 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>

## Основи верстання. Табличне верстання. Блочне верстання. iframe

Під верстанням веб-сторінки розуміють процес її створення шляхом компонування текстових і графічних елементів. При створенні дизайну сайту дизайнер розробляє макет в графічному редакторі. Макет є звичайним зображенням. Тому, щоб використовувати дизайн на сайті, верстальник «перетворює» макет в веб-сторінку, де розташування елементів задається з використанням HTML і CSS.

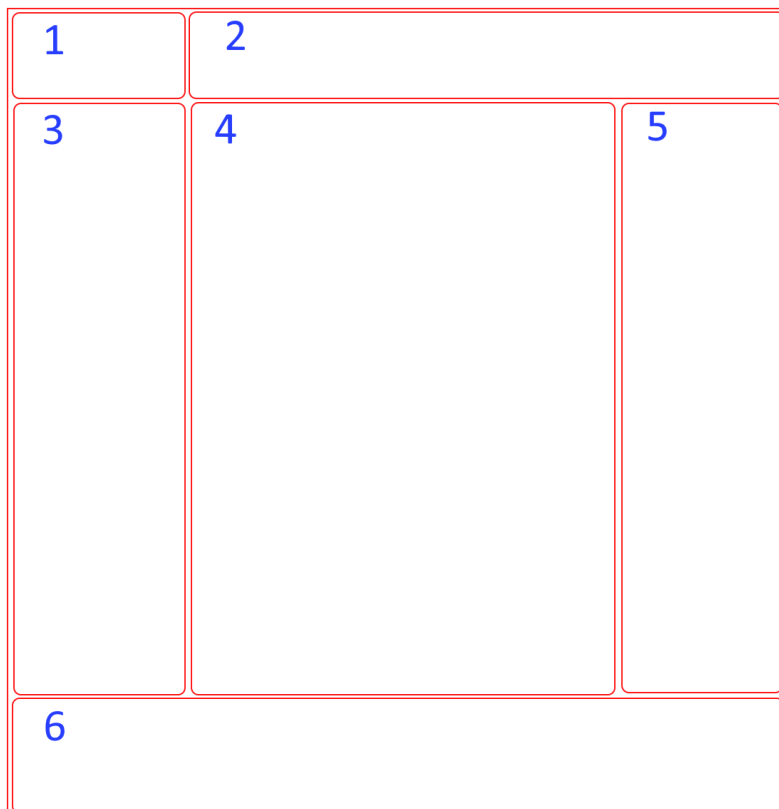


Рисунок 4.1 - Основні блоки сторінки

Як правило, веб-сторінка представляє собою набір прямокутних блоків.

Основні блоки на сторінці: 1 - логотип, 2 - верхня частина (header), 3 - ліва колонка, 4 - центральна колонка, 5 - права колонка, 6 - нижня частина (footer), 7 - загальний фон сторінки. Блоки в свою чергу можуть містити в собі інші дрібніші блоки: пункти меню, панелі, і т.п. У верхній частині розташовується основне навігаційне меню і форма пошуку, в лівій колонці - меню розділу, в правій колонці - важливі оголошення, в центральній -

основний зміст сторінки, а в нижній частині дублюється верхнє меню (щоб для переходу в інший розділ не потрібно було прокручувати всю сторінку знизу вгору) і контакти. Така верстка сторінки є з трьома колонками. Часто використовуються три колонки або дві. У двоколонковому верстанні, як правило, відсутня права колонка. Зрозуміло, існують і інші варіанти компоновки елементів, наприклад в одну колонку. На рис. нижче. показані моделі сторінок з однією або двома колонками.

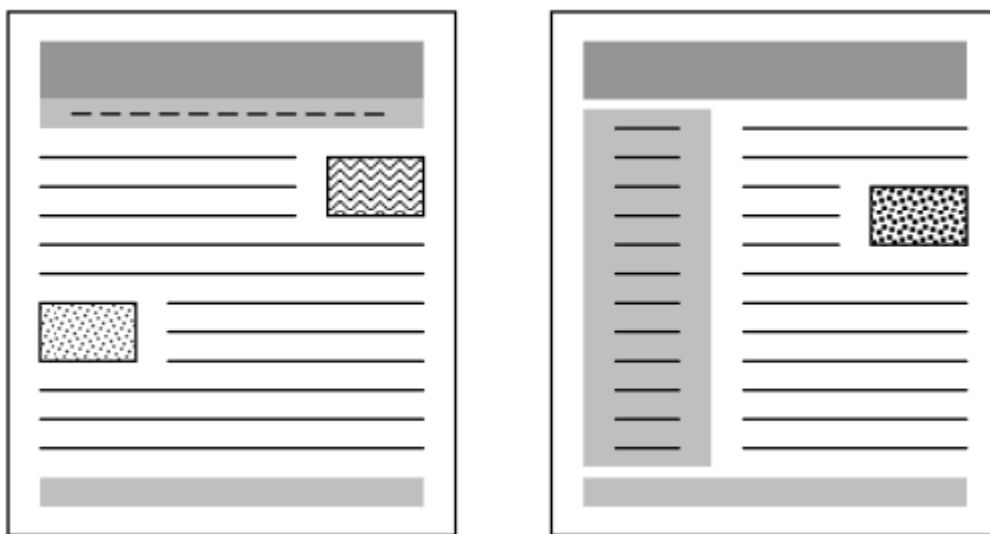


Рисунок 4.2 - Моделі сторінок з однією або двома колонками

Приклад сторінки з однією колонкою - пошукова система Google.com, двоколонковому - блог habrahabr.ru та інші.

### **Фіксоване і нефіксоване верстання**

Крім способу компоновання блоків найважливішою характеристикою сторінки є спосіб завдання її ширини. Існує два основні підходи:

1. Завдання строго фіксованої ширини сторінки. В цьому випадку, якщо ширина сайту перевищує ширину вікна браузера, з'явиться горизонтальна смуга прокрутки. Якщо ж ширина сайту буде менше ширини вікна, то з'явиться порожній простір з краю сторінки.

2. Прив'язка ширини сторінки до ширини екрану. У цьому випадку розмір блоків сторінки пропорційно залежить від розмірів екрану. Якщо вікно

звужується, то звужуються і блоки. Якщо вікно розтягується, блоки розширюються. Таке верстання часто називається «гумовим».

Приклади: [www.w3c.org](http://www.w3c.org), [www.icann.net](http://www.icann.net).

Між фахівцями ведуться суперечки, який з підходів краще. «Гумове» верстання може адаптуватися під різні роздільні здатності екранів, але, з іншого боку, на невеликому моніторі блоки сторінки можуть занадто звужитися, а на широкоформатному екрані - стати занадто широкими. В таких умовах користувачеві буде незручно читати текст на сайті.

Щоб цього уникнути, необхідно ставити мінімальну і максимальну ширину «гумової» сторінки. Тому такий спосіб верстки є технічно більш складним, ніж фіксований.

### Сумісність з браузерами

В процесі верстки необхідно домогтися коректного відображення сайту в найбільш популярних браузерах при різних роздільних здатностях екрану. На жаль, браузери можуть реалізувати неправильно деякі можливості CSS. Через це розробникам сайтів доводиться використовувати різні прийоми, що дозволяють створити сайт, який однаково відображається у всіх браузерах. Необхідно переглядати сторінки в браузерах Chrome, Mozilla FireFox, Opera, Internet Explorer, враховуючи мобільні версії - цими браузерами користується велика кількість аудиторії Інтернету.

Для перевірки дотримання браузерами веб-стандартів HTML, CSS і ін. є тест Acid.

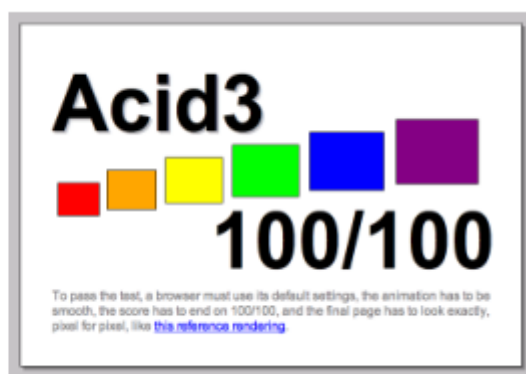
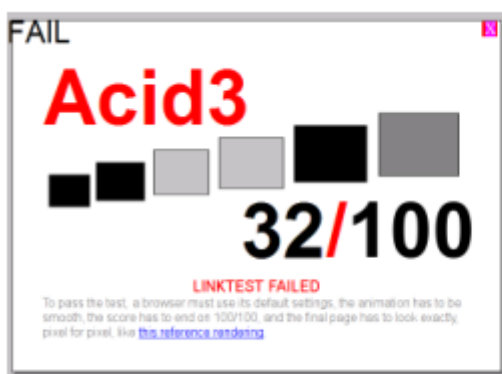


Рисунок 4.3 Проходження тесту Acid3 браузером Internet Explorer 8 (зліва) і Opera 10 (праворуч) .

Також необхідно передбачити роботу сайту при різних роздільних здатностях екрану. В даний час практично всі користувачі (98%) працюють з дозволом екрану  $1920 \times 1080$ . Тому максимальна ширина сайту не повинна перевищувати приблизно 1800 пікселів, тому що необхідно залишити запас для смуги прокрутки і рамки вікна браузера. В іншому випадку у користувача з невеликим екраном з'явиться горизонтальна смуга прокрутки, що сильно ускладнить читання сайту.

Способами верстання є використання блоків і таблиць.

### **Табличне верстання**

Найбільш простим, але мало використовуваним на сьогоднішній день, способом є верстання таблицями. Ідея полягає в тому, що всі елементи сторінки розміщуються в таблиці з невидимими кордонами.

На рисунку сторінка з трьома колонками:



Рисунок 4.4 - Сторінка з трьома колонками



Каркасом такої сторінки буде таблиця з п'яттю комірками. У таблиці буде три стовпці і три рядки, причому в першому і останньому рядку одна комірка буде розтягнута на 3 стовпці за допомогою атрибута `colspan`.

Використовуючи таблиці, зручно задавати розміри комірок (блоків сторінки). У прикладі ширина бічних колонок дорівнюватиме 200 пікселям, центральної - 500 пікселям. Таким чином, ми отримаємо жорсткий «каркас» сторінки. Висота блоків буде залежати від кількості вмісту в них. Висоту «шапки» і «підвалу» задамо 60 і 20 пікселів відповідно.

Код сторінки:

```
<html>
<head>
<title>Табличне верстання</title>
<style>
BODY {
margin: 0; /* обнуляються поля у стрінки */
text-align: center; /* вирівнювання по центру */
}
#main {
margin: 0 auto; /* центрування таблиці */
border-collapse: collapse; /* об'єднання комірок */
}
TD { /* стиль для всіх комірок */
vertical-align: top;
margin: 0;
padding: 5px;
}
#header { /* стиль заголовка */
background-color: #999999;
text-align: center;
height: 60px;
}
#left_col { /* стиль лівої колонки */
width: 200;
background-color: #bbbbbb;
}
#center_col { /* стиль центральної колонки */
width: 500px;
}
#right_col { /* стиль правої колонки */
width: 200px;
```

```

background-color: #bbbbbb;
}
#footer { /* стиль нижнього блока */
background-color: #999999;
text-align: center;
height: 20px;
}
</style>
</head>
<body>
<table id="main">
<tr>
<td colspan="3" id="header">
...
</td>
</tr>
<tr>
<td id="left_col">
...
</td>
<td id="center_col">
...
</td>
</tr>
<tr>
<td colspan="3" id="footer">
...
</td>
</tr>
</body>
</html>

```

Загальна ширина сторінки, враховуючи padding 5 пікселів буде дорівнює  $5 + 200 \{ \{ 1 \} \} + 5 + 5 + 500 + 5 + 5 + 200 + 5 = 930$  пікселів. Результат показаний на рисунку нижче

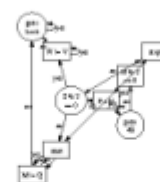
Investigation of Replication		
1) Introduction 2) Model 3) Implementation 4) Results 5) Related Work 6) Conclusion 7) Introduction 8) Model 9) Implementation 10) Results 11) Related Work 12) Conclusion 13) Introduction 14) Model 15) Implementation 16) Results 17) Related Work 18) Conclusion 19) Introduction 20) Model 21) Implementation 22) Results 23) Conclusion	<h2>1 Model</h2> <p>Suppose that there exists Markov models [2] such that we can easily harness simulated annealing. Despite the fact that systems engineers always believe the exact opposite, Lock depends on this property for correct behavior.</p>  <p>Figure 1: Our application prevents the construction of expert systems in the manner detailed above.</p> <p>Our approach is related to research into homogeneous configurations, modular architectures, and the exploration of the lookahead buffer. A recent unpublished undergraduate dissertation introduced a similar idea for self-learning modalities. Lock represents a significant advance above this work. On a similar note, recent work by Zhang suggests a heuristic for caching expertlocks, but does not offer an implementation [9,1]. Obviously, comparisons to this work are idiotic. The acclaimed framework does not cache simulated annealing as well as our method.</p> <h2>2 Conclusion</h2> <p>We disconfirmed in this paper that architecture and sensor networks can connect to solve the quandary, and our heuristic is no exception to that rule. In fact, the main contribution of our work is that we confirmed that although evolutionary programming can be made game-theoretic, multimodal, and modular, public-private key pairs [2] and simulated annealing can collude to answer the quandary. We expect to see many systems engineers move to emulating our methodology in the very near future.</p>	<p>After several minutes of difficult coding, we finally have a working implementation of Lock. Lock requires root access in order to create kernel-time algorithms. Our methodology requires root access in order to cache the synthesis of agents. It was necessary to cap the signal-to-noise ratio used by Lock to 600 heratops. It was necessary to cap the seek time used by our application to 753 MB/s. Steganographers have complete control over the coherence of 21 Trilog files, which of course is necessary so that red-black trees can be made variable, client-server, and variable.</p> <p>It is possible to justify having paid little attention to our implementation and experimental setup? He. We ran 100 automata on 61 nodes spread throughout the Internet network, and compared them against public-private key pairs running locally. (2) we ran local-area networks on 19 nodes spread throughout the underwire network, and compared them against local-area networks running locally. (3) we asked (and answered) what would happen if opportunistic wireless 2 bit architectures were used instead of wide-area networks, and (4) we measured NV-RAM speed as a function of hard disk space on a Nintendo Gameboy.</p>

Рисунок 4.5 - Верстання в три колонки за допомогою таблиці

Якщо внести невеликі зміни в CSS стилі, можна зробити сторінку, яка буде розтягуватися. Наприклад, можна залишити ширину бічних колонок фіксованою, а загальну ширину таблиці прив'язати до ширини вікна. Тоді розмір центральної колонки браузер буде обчислювати, віднімаючи з ширини таблиці ширину бічних колонок. Для цього достатньо для #main додати правило width: 90%, а у #center\_col прибрати властивість width.

Для компоновання складних сторінок можна використовувати вкладені таблиці. Розглянемо приклад.

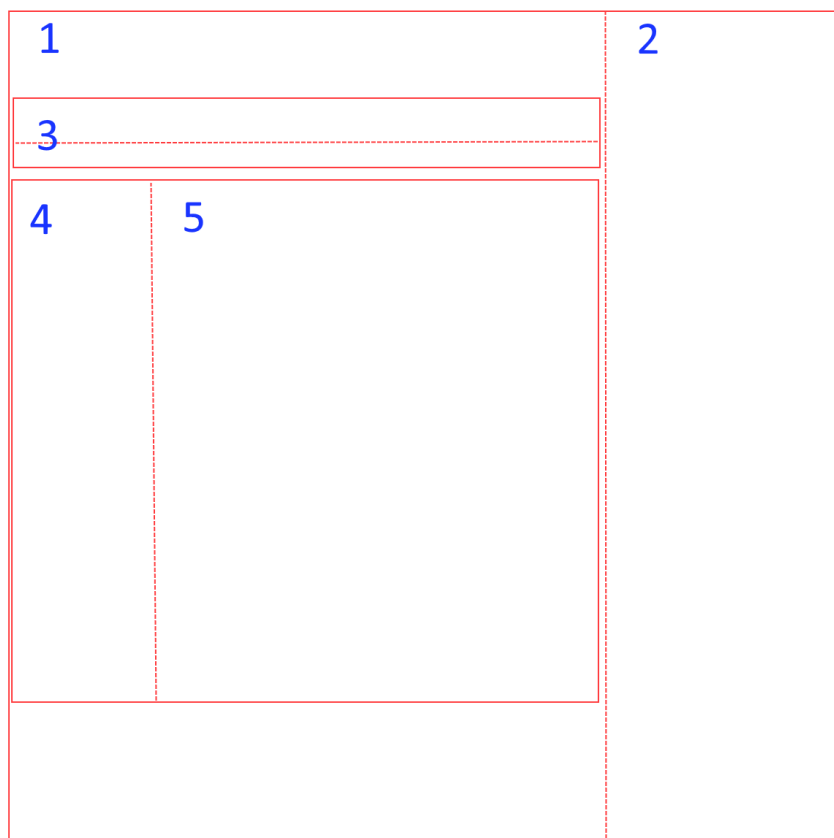


Рисунок 4.6 - Блоки сайту

Сайт складається з таблиці, яка розділяє його на дві частини - велику ліву (1) і меншу праву (2). Основний зміст розташовується в лівій частині. У неї вкладено ще 2 таблиці: таблиця з основним горизонтальним меню (3) і таблиця з меню розділу (4) і текстом сторінки (5).

Основними перевагами табличної верстки в порівнянні з блочною є простота і зручність реалізації, відсутність проблем відображення в більшості браузерів. Як недоліки відзначають збільшення обсягу HTML коду і більш повільне завантаження елементів сторінки. До того ж таблиці спочатку створювалися для відображення табличних даних, а не для компоновки сторінки. Тому при верстці таблиці використовуються абсолютно не за прямим призначенням.

Специфікація CSS пропонує інший інструмент: побудова сторінок з блокових елементів DIV. Такий підхід є більш логічним, відповідає вимогам веб-стандартів і рекомендацій, але з іншого боку складніший.

## Блочне верстання

Блочне верстання є набагато складнішим в освоєнні, ніж табличне.

Верстка за допомогою блоків є більш сучасною технологією, ніж таблична верстка. Основною ідеєю блокової верстки є використання елементів DIV і CSS-стилів.

Реалізуємо приклад з рисунка нижче за допомогою блочного верстання.

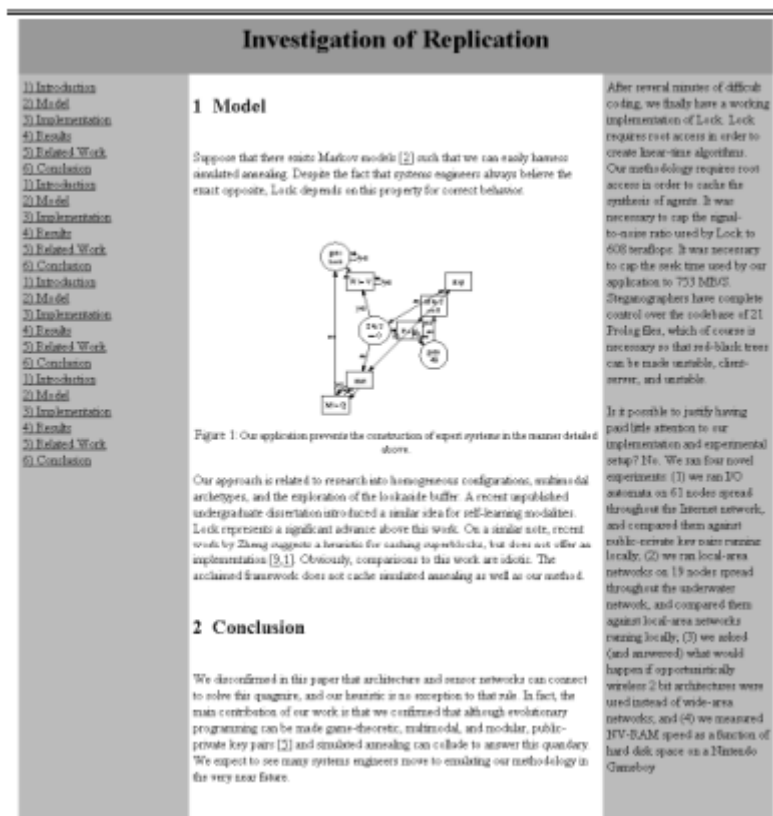


Рисунок 4.7 - Блоки сайту

Спочатку необхідно визначити в HTML-кодї сторінки основні блоки:

```
<div id="wrap">
<div id="header"></div>
<div id="left_col"></div>
<div id="center_col"></div>
<div id="right_col"></div>
<div id="footer"></div>
</div>
```

Блок wrap є контейнером ( «обгорткою») для всіх інших блоків сторінки: заголовка, лівої, середньої та правої колонок і «підвалу».

Задамо правила CSS. Рекомендується спочатку скинути параметри відступів для всіх елементів:

```
* {  
margin: 0;  
padding: 0;  
}
```

Задамо ширину блоку wrap і відцентруємо його:

Тепер задамо параметри інших блоків. Для того щоб ліва і права колонки зайняли своє місце, використовуємо правило float. Щоб опустити "підвал" вниз використовуємо clear.

```
#header {  
padding: 5px;  
background-color: #999999;  
text-align: center;  
height: 60px;  
}  
#left_col {  
float: left;  
padding: 5px;  
width: 200px;  
background-color: #bbbbbb;  
}  
#center_col {  
float: left;  
padding: 5px;  
width: 570px;  
background-color: #ffffff;  
}  
#right_col {  
float: right;  
padding: 5px;  
width: 200px;  
background-color: #bbbbbb;  
}  
#footer {  
clear: both;  
padding: 5px;  
background-color: #999999;  
text-align: center;  
height: 20px;  
}
```

## Результат показаний на рисунку

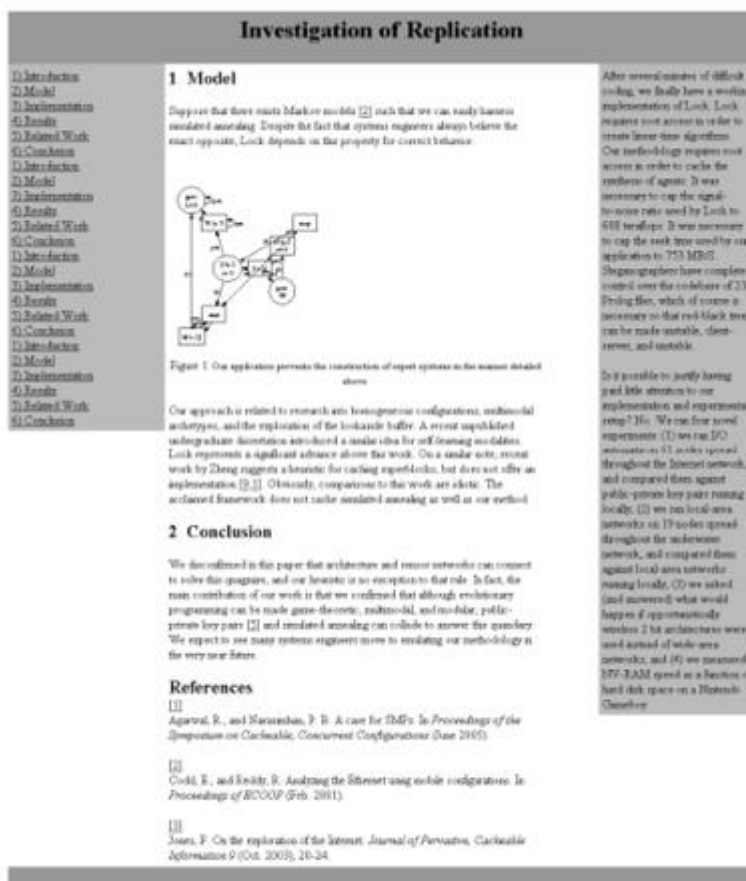


Рисунок 4.8 - Верстання в три колонки за допомогою блоків DIV

Відразу в очі кидається проблема, якої немає при використанні табличного верстання: висота колонок виходить різною. На жаль, в CSS не передбачено правил для завдання рівної висоти колонок, так як передбачається, що висота блоку повинна залежати тільки від його вмісту. Тому розробникам доводиться вдаватися до різного роду «трюкам».

Найпростіший варіант в нашому випадку - поставити такий же колір фону для блоку wrap, як у бічних колонок.

```
#wrap {
width: 1000px;
margin: 0 auto;
background-color: #bbbbbb;
}
```

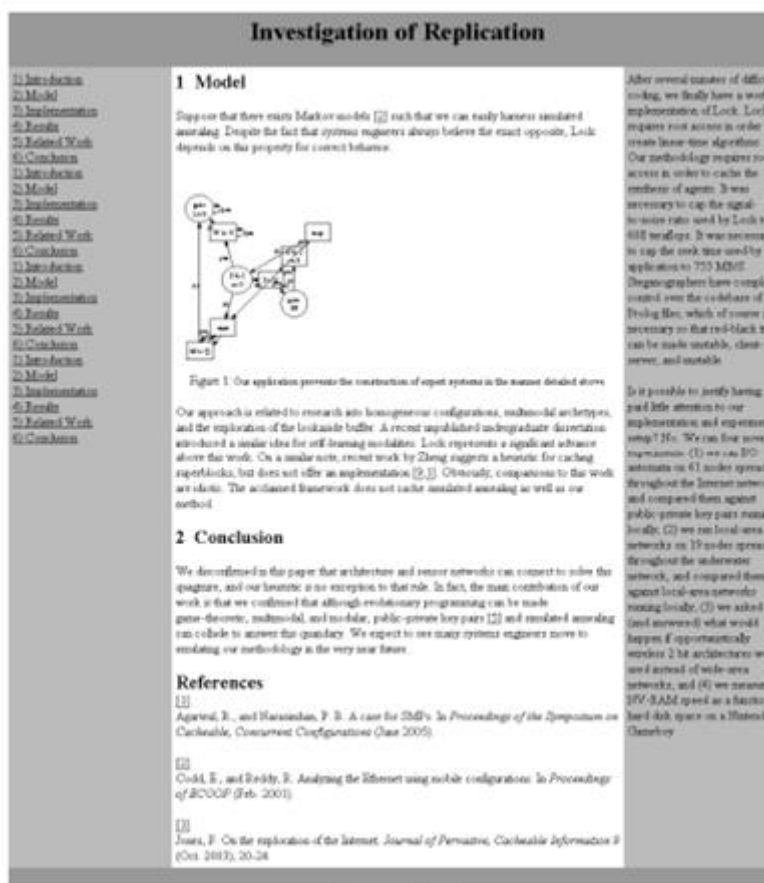


Рисунок 4.9 - Верстання в три колонки за допомогою блоків DIV

Хоча висота колонок не змінилася, користувачеві це помітно не буде.

В Інтернеті існує безліч готових CSS-шаблонів і інструментів для генерації нових. У багатьох випадках розумно скористатися ними.

## 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 зміщуватися вліво або вправо на сторінці з оточуючим його текстом. Фрейм може виводити один документ або може бути місцем, де виводяться кілька з'єднаних документів. Наприклад, кілька посилань на сторінці можуть виводити різні зображення 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"
```

```
Вимкнені:  
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>
```

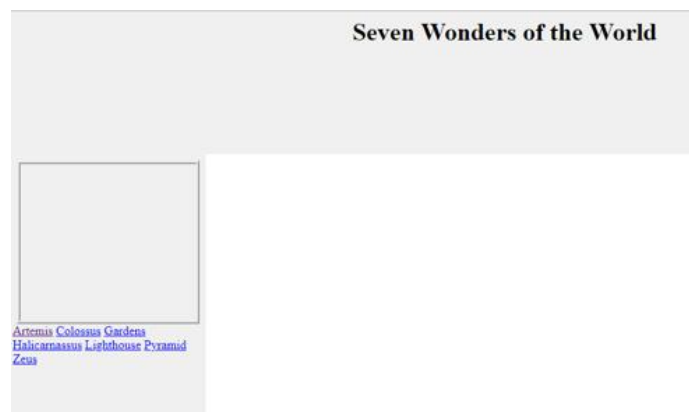


Рисунок 4.10 - Верстання `iframe`

## Лекція 5

### CSS. Внутрішні стилі. Стилї рівня документа. Зовнішні стилі

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-сторінки.

#### 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 тегами.**

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 застосується 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 A, 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:

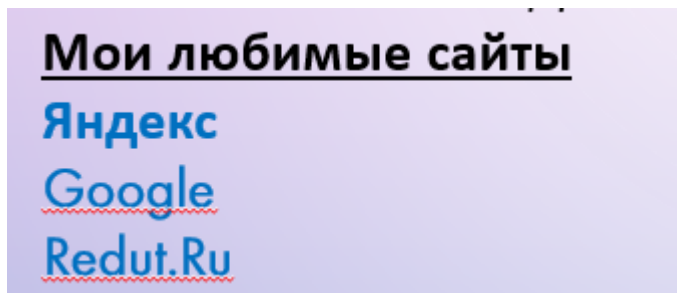
`<H1 class = "class1"> Мої улюблені сайти </ h1>`

`<a href="http://yandex.ru" class="class1"> Яндекс </a> <br>`

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

`<a href="http://redut.ru" class="class1"> Redut.ru </a>`

У браузері буде відображатися:

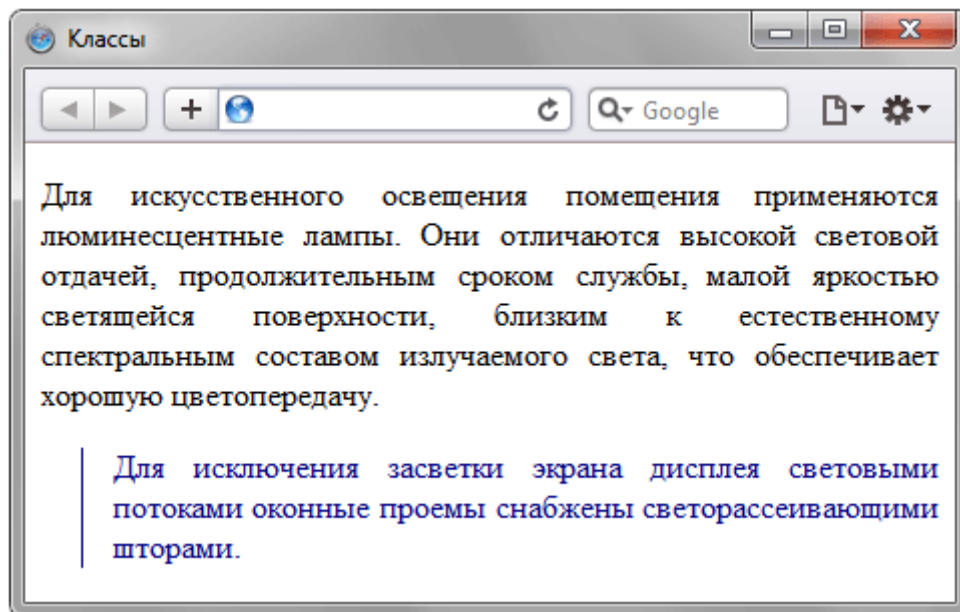


Можна вказувати для одного елемента кілька класів через пробіл.

Приклад 2:

```
<html>
<head>
  <meta charset="utf-8">
  <title>Классы</title>
  <style>
    P { /* Обычный абзац */
      text-align: justify; /* Выравнивание текста по ширине */
    }
    P.cite { /* Абзац с классом cite */
      color: navy; /* Цвет текста */
      margin-left: 20px; /* Отступ слева */
      border-left: 1px solid navy; /* Граница слева от текста */
      padding-left: 15px; /* Расстояние от линии до текста */
    }
  </style>
</head>
<body>
  <p>Для искусственного освещения помещения применяются
люминесцентные лампы. Они отличаются высокой световой отдачей,
продолжительным сроком службы, малой яркостью светящейся
поверхности, близким к естественному спектральным составом излучаемого
света, что обеспечивает хорошую цветопередачу.</p>
  <p class="cite">Для исключения засветки экрана дисплея световыми
потоками оконные проемы снабжены светорассеивающими шторами.</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 `site` - 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>Классы</title>
  <style>
    .gost {
      color: green; /* Цвет текста */
      font-weight: bold; /* Жирное начертание */
    }
    .term {
      border-bottom: 1px dashed red; /* Подчеркивание под текстом */
    }
  </style>
</head>
<body>
  <p>Согласно  <span class="gost">ГОСТ 12.1.003-83 ССБТ
  &quot;Шум. Общие требования безопасности&quot;</span>, шумовой
  характеристикой рабочих мест при постоянном шуме являются уровни
  звуковых давлений в децибелах в октавных полосах. Совокупность таких
  уровней называется
```



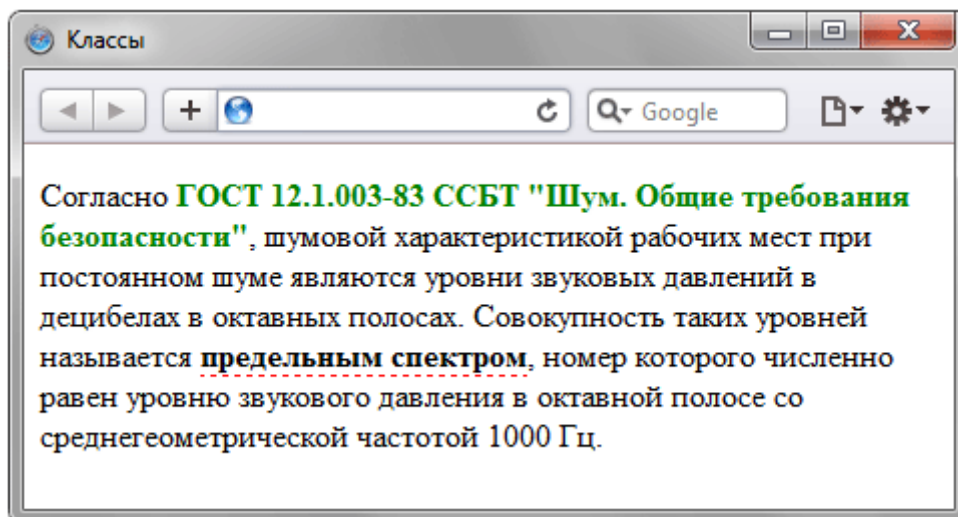
`<b class="term">предельным спектром</b>`, номер которого численно равен уровню звукового давления в октавной полосе со среднегеометрической частотой 1000 Гц.

`</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>Универсальный селектор</title>`

`<style>`

`* {`

`font-family: Arial, Verdana, sans-serif; /* Рублений шрифт для текста`

`*/`

```
font-size: 96%; /* Розмер текста */  
}  
</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">Мои любимые сайты</h1>
<a href="http://yandex.ru" class="class1">Яндекс</a><br>
<a href="http://google.com" class="class1">Google</a><br>
<a href="http://redut.ru" class="class1"> Redut.ru </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">Мои любимые сайты</h1>
<a href="http://yandex.ru" class="class1">Яндекс</a><br>
<a href="http://google.com" class="class1">Google</a><br>
<a href="http://redut.ru" class="class1"> Redut.ru </a>
</body>
</html>
```

## 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.

## 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") типу носіїв;*

*@import "ім'я файлу" типу 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 екстреним випадком і вам необхідно підвищити значимість якої-небудь властивості, можна додати до нього оголошення!

Important:

*p {color: red! important;}*

*p {color: green;}*

Також! Important перебиває 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 класу:	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 неподчеркнутою, 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://htmlbook.ru" class="links"
id="greenlink">htmlbook.ru</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 кольорів 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.

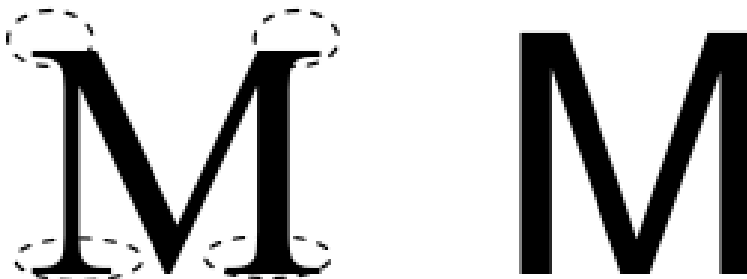
**sans-serif** - рубані шрифти (шрифти без зарубок або гротески), типові представники - Arial, Impact, Tahoma, Verdana;

**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.



*. Сравнение буквы М антиквенного и рубленного шрифта.*

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** (курсивне) або **oblique** (похиле). Курсив є спеціальною зміненою версією

шрифту, що імітує рукописний текст з нахилом вправо. Похиле накреслення виходить зі звичайного нахилом букв.

Різниця видно на прикладі:

The five boxing wizards jump quickly.  
*The five boxing wizards jump quickly.*  
**The five boxing wizards jump quickly.**

Зазвичай браузер не може відобразити похиле накреслення і замінює його курсивним.

**font-weight** - дозволяє змінити рівень жирності тексту: **normal** (звичайна), **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}*

## Лекція 6

### CSS-властивості

#### 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).

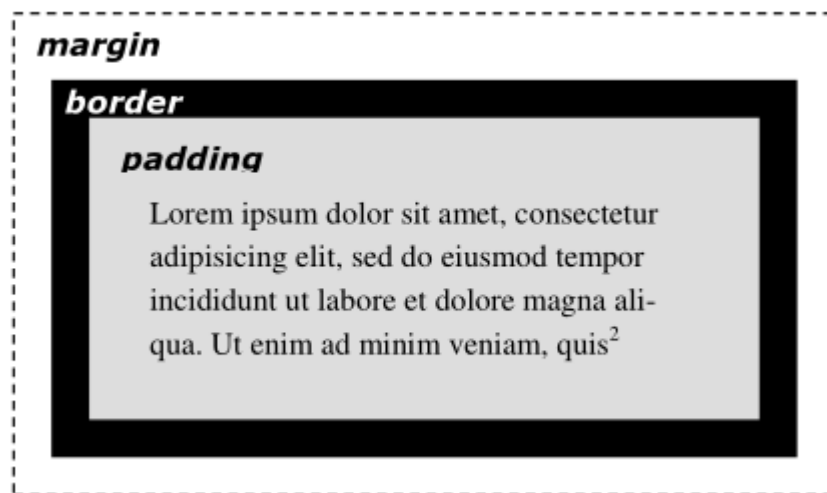


Рисунок Бокс (box) елемента.

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 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.

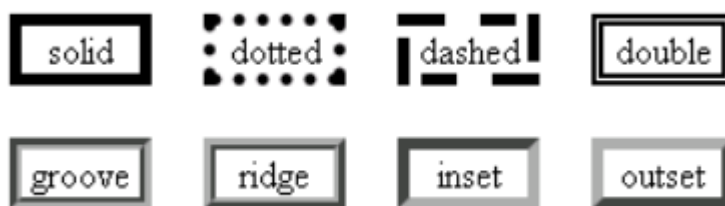


Рисунок Види кордонів.

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

У браузері:

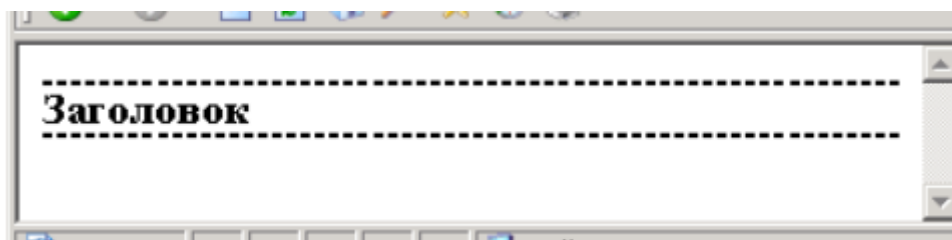


Рисунок властивості меж вказані окремо

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>Пример</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">Quo usque tandem abutere, Catilina, patientia nostra? quam diu
etiam furor iste tuus nos eludet? quem ad finem sese effrenata iactabit
audacia?</p>
<p id="text2">Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
<p id="text3">Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat.</p>
</body>
</html>
```

Результат:

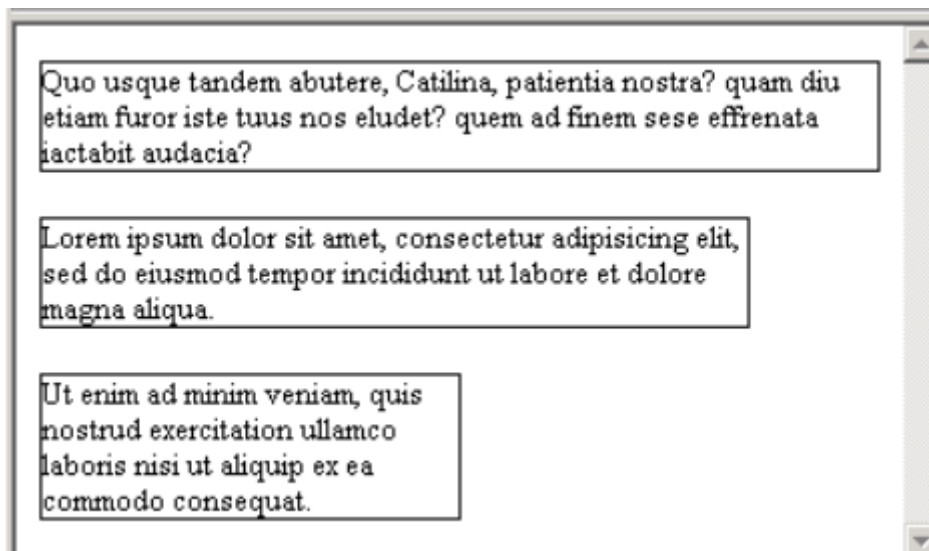


Рисунок Відображення прикладу в браузері

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:

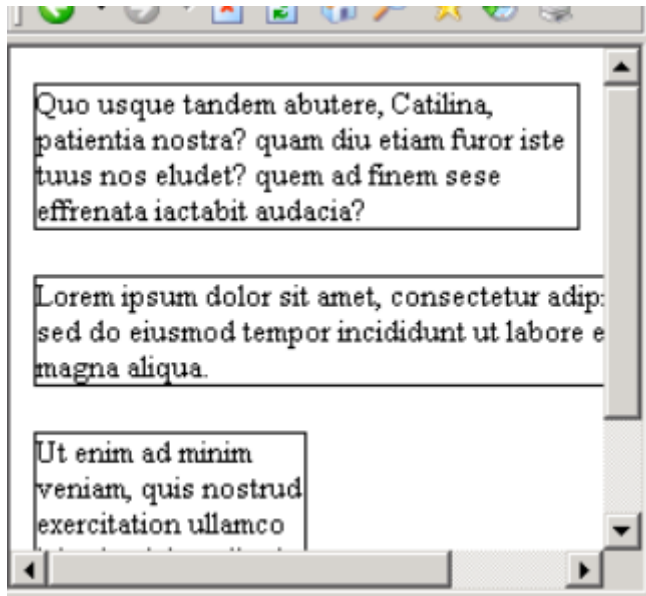


Рисунок Відображення прикладу в браузері при зменшенні ширини вікна

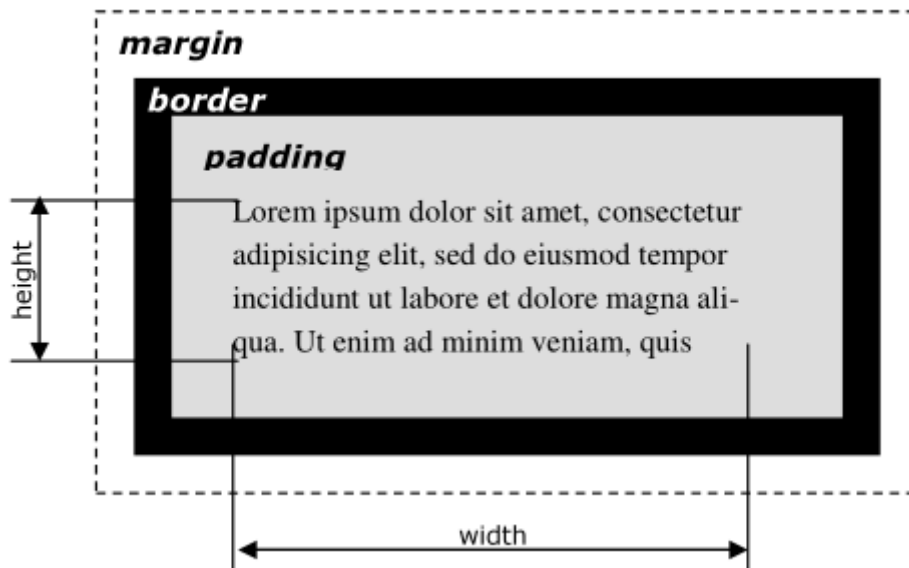
Можна ставити мінімальні і максимальні розміри властивостями **min-width**, **min-height** і **max-width**, **max-height**. Ці властивості не підтримує браузер Internet Explorer версії 6 і нижче.

Загальні розміри елемента складаються так:

$$\text{Ширину} = \text{width} + \text{padding} + \text{border} + \text{margin}$$

$$\text{Висота} = \text{height} + \text{padding} + \text{border} + \text{margin}$$

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.

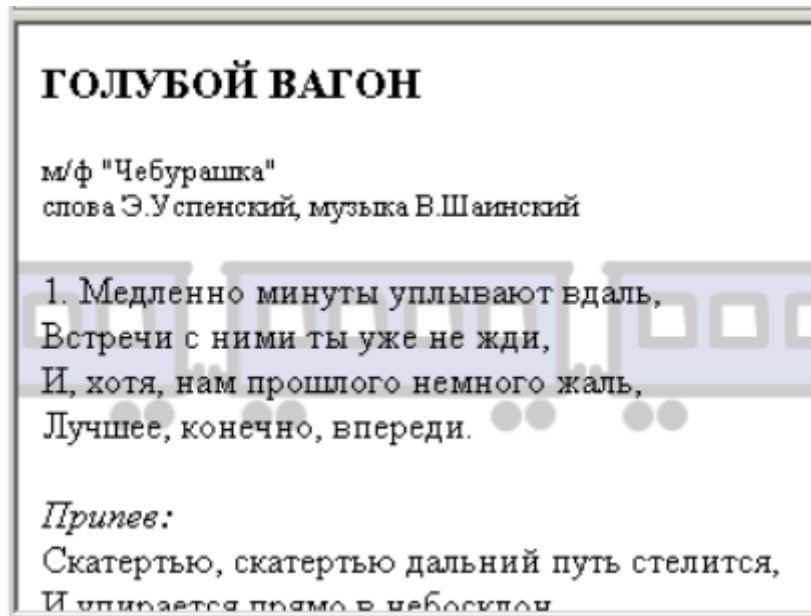


Рисунок Фонове зображення.

CSS код:

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

У браузері:



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>Популярность браузеров в мире</title>
</head>
<body>
<table>
<tr>
<th>Год\Браузер</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:

Год\Браузер	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 за замовчуванням

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

Додамо в тег <head> ... </ head> тег <style> ... </ style>, а до теги <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 заголовочных ячеек */
TABLE#browser_stats TH{
border: 1px solid black;

```

```

background-color: gray;
padding: 0.5em;
}
/* стиль ячеек с данными */
TABLE#browser_stats TD{
border: 1px solid black;
padding: 0.3em;
text-align: right;
}
</style>

```

У браузері:

Год\Браузер	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 кожної комірки з'явилася власна рамка. Щоб цього не відбувалося, необхідно вказати в правилах для таблиці властивість `border-collapse` із значенням `collapse`.

Результат:

Год\Браузер	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.

Год\Браузер	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>Популярность браузеров в мире</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;
}
TABLE#browser_stats .lc{
text-align: right;
border-right: 1px solid black;
```

```

width: 100px;
}
</style>
</head>
<body>
<table id="browser_stats">
<tr>
<th class="lc">Γοð\Βpayzep</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>

```

## Селектори (продовження)

### Сусідні селектори

`<P> Lorem <b> ipsum </ b> dolor sit amet, <i> consectetuer </ i> adipiscing <tt> elit </ 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.

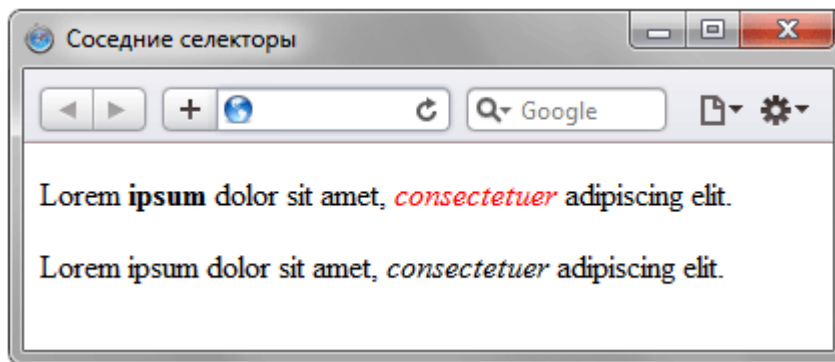
*Селектор 1 + селектор 2 {Опис правил стилю}*

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; /* Красный цвет текста */
  }
</style>
</head>
<body>
  <p>Lorem <b>ipsum </b> dolor sit amet, <i>consectetuer</i>
adipiscing elit.</p>
  <p>Lorem ipsum dolor sit amet, <i>consectetuer</i> adipiscing elit.</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 «consectetuer» 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.

*Teg1 Teg2 {...}*

В цьому випадку стиль буде застосовуватися до Teg2 коли він розміщується всередині Teg1, як показано нижче.

*<Teg1>*

*<Teg2> ... </Teg2>*

*</Teg1>*

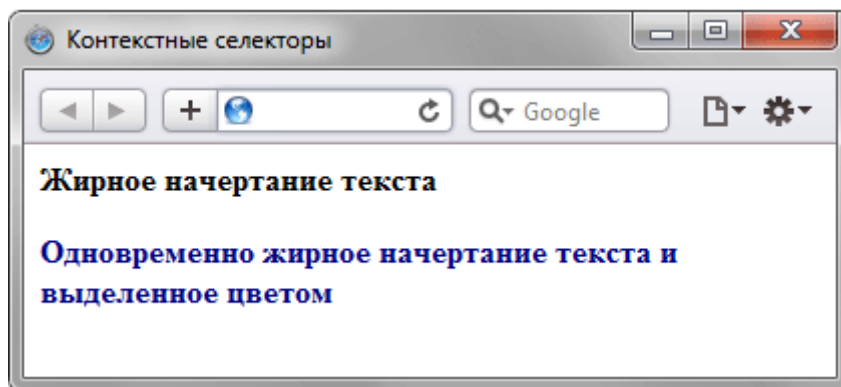
Використання контекстних селекторів продемонстровано в наступному прикладі.

Приклад:

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
  <title>Контекстные селекторы</title>
  <style>
    P B {
      font-family: Times, serif; /* Семейство шрифта */
      color: navy; /* Синий цвет текста */
    }
  </style>
</head>
<body>
  <div><b>Жирное начертание текста</b></div>
  <p><b>Одновременно жирное начертание текста
и выделенное цветом</b></p>
</body>
</html>
```

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.

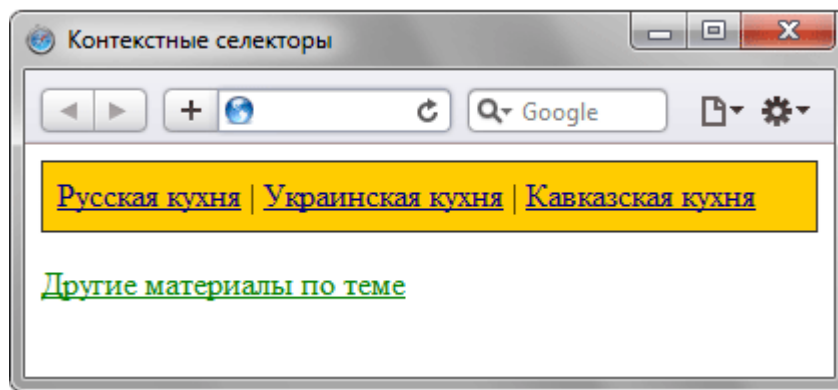
Ширші можливості контекстні селектори дають при використанні ідентифікаторів і класів. Це дозволяє встановлювати стиль тільки для того

елемента, який розташовується усередині певного класу, як показано в наступному прикладі.

Приклад:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Контекстные селекторы</title>
<style>
A {
  color: green; /* Зеленый цвет текста для всех ссылок */
}
.menu {
  padding: 7px; /* Поля вокруг текста */
  border: 1px solid #333; /* Параметры рамки */
  background: #fc0; /* Цвет фона */
}
.menu A {
  color: navy; /* Темно-синий цвет ссылок */
}
</style>
</head>
<body>
<div class="menu">
  <a href="http://book/1.html">Русская кухня</a> |
  <a href="http://book/2.html">Украинская кухня</a> |
  <a href="http://book/3.html">Кавказская кухня</a>
</div>
<p><a href="http://book/text.html">Другие материалы по
теме</a></p>
</body>
</html>
```

Результат:



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

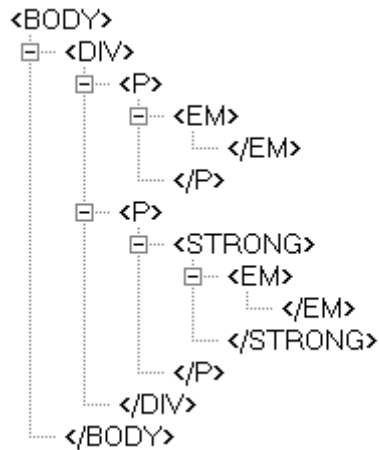
Дочірнім називається елемент, який безпосередньо розташовується всередині батьківського елемента.

Приклад:

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
  <title>Lorem ipsum</title>
</head>
<body>
  <div class="main">
    <p><em>Lorem ipsum dolor sit amet</em>, consectetur adipiscing
    elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna
    aliquam
    erat volutpat.</p>
    <p><strong><em>Ut wisi enim ad minim veniam</em></strong>,
    quis nostrud exerci tution ullamcorper suscipit lobortis nisl ut aliquip ex
    ea commodo consequat.</p>
  </div>
</body>
```

</html>

В даному прикладі застосовується кілька контейнерів, які в кодї 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

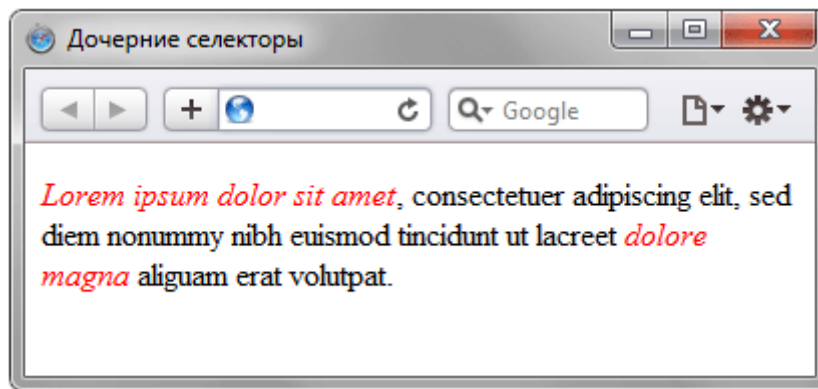


другого абзацу. А все через те, що тег `<em>` в другому абзаці розташований в контейнері `<strong>`, тому порушується умова вкладеності.

За своєю логікою дочірні селектори схожі на селектори контекстні. Різниця між ними така. Стил 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:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title>Дочерние селекторы</title>
<style>
P > I { /* Дочерний селектор */
color: red; /* Красный цвет текста */
DIV I { /* Контекстный селектор */
color: green; /* Зеленый цвет текста */
}
}
</style>
</head>
<body>
<div>
<p><i>Lorem ipsum dolor sit amet</i>, consectetur adipiscing
elit, sed diam nonummy nibh euismod tincidunt ut laoreet <i>dolore
magna</i>
aliquam erat volutpat.</p>
</div>
</body>
</html>
```

Результат:



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.

## Лекція 7

### Блокові, рядкові елементи. Псевдокласи. Псевдоелементи. Позиціонування

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.

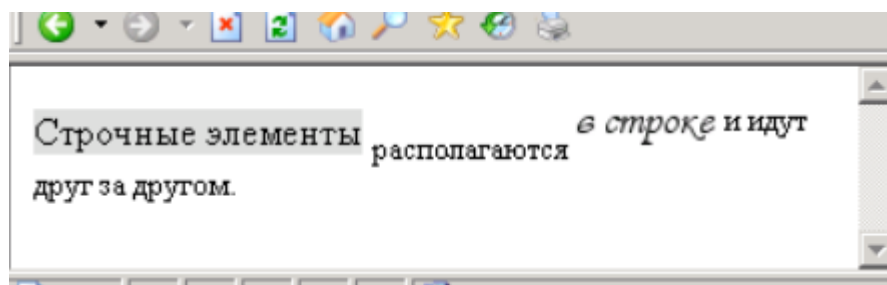
Основна відмінність між блоковими і малими елементами полягає в наступному: рядкові елементи йдуть один за одним в рядку тексту, а блочні - розташовуються один за іншим. До рядкових елементів відносяться такі теги, як `<a>`, `<img>`, `<input>`, `<select>`, `<span>`, `<sub>`, `<sup>` і ін.

До блокових: `<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">Строчные элементы</span>  
<sub>располагаются</sub>  
  
<sup>и идут друг за другом.</sup>
```

У браузері:



Риснок Поведінка малих елементів.

Розглянемо приклад для блокових тегів:

```
<head>
<title>Блочные элементы</title>
<style>
H3, DIV, TABLE {
border: black dotted 1px;
margin: 5px;
padding: 5px;
}
</style>
</head>
<body>
<h3>Заголовок</h3>
<div>Содержимое &lt;div>&gt;
<div>Вложенный &lt;div>&gt; №1</div>
<div>Вложенный &lt;div>&gt; №2</div>
</div>
<table>
<tr><td>Таблица из одной ячейки</td></tr>
</table>
</body>
```

Результат:

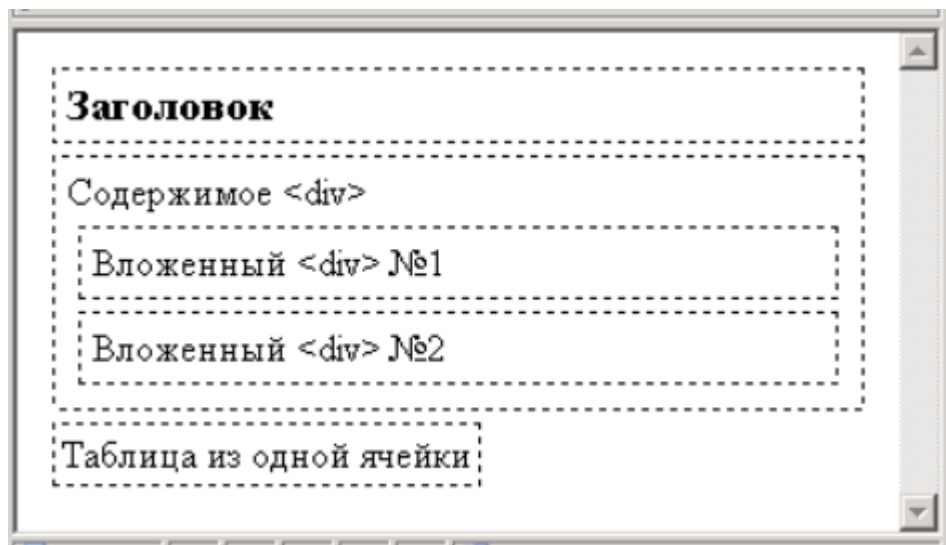


Рисунок Поведінка блокових елементів.

Блочные элементы размещаются один под одним, много занимают всю возможную ширину. Блочные элементы могут включать в себя малые и другие блочные. Но малые элементы не могут содержать блочные!

Еще одной особенностью является то, что для малых элементов не работают такие свойства, как **margin-top**, **margin-bottom**, **padding-top** и **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.

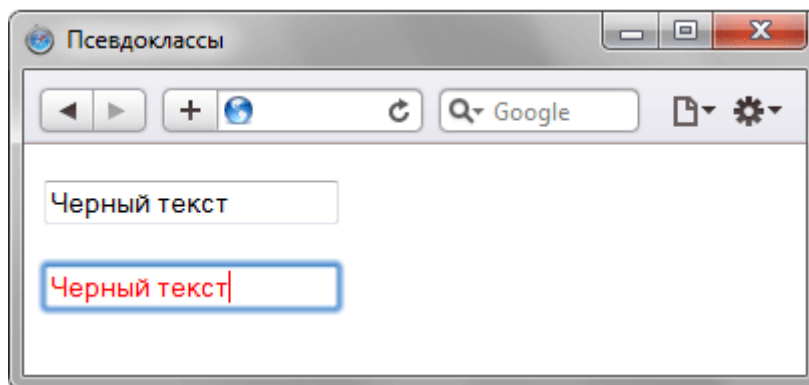
**:focus** - застосовується до елемента при отриманні ним фокусу.

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

Приклад:

```
<head>
  <meta charset="utf-8">
  <title>Псевдоклассы</title>
  <style>
    INPUT:focus {
      color: red; /* Красный цвет текста */
    }
  </style>
</head>
<body>
  <form action="">
    <p><input type="text" value="Черный текст"></p>
    <p><input type="text" value="Черный текст"></p>
  </form>
</body>
```

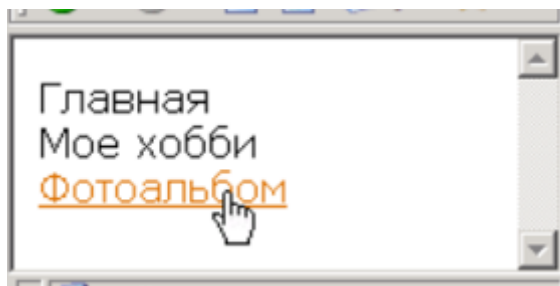
Результат:



Приклад 2:

```
<html>
<head>
<title>Пример</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">Главная</a><br>
<a href="hobby.html">Мое хобби</a><br>
<a href="photo.html">Фотоальбом</a><br>
</body>
</html>
```

Результат:



Псевдокласи, що мають відношення до дерева документа

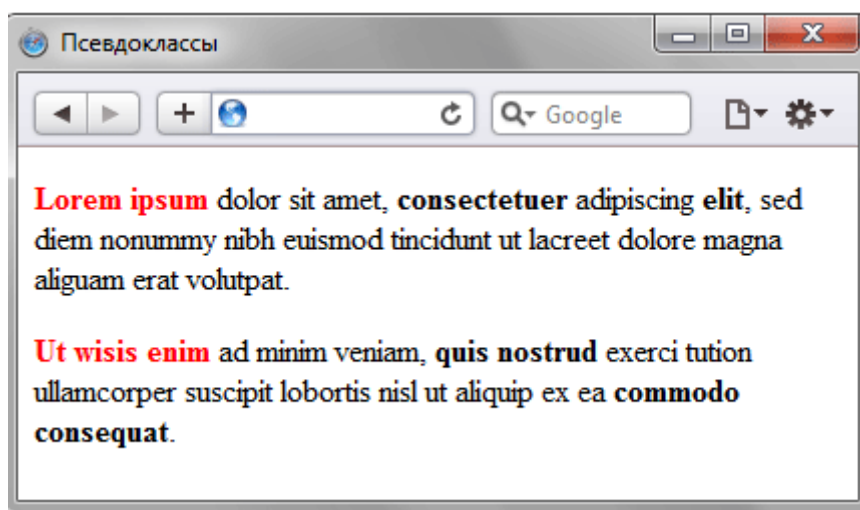
До цієї групи належать псевдокласи, які визначають положення елемента в дереві документа і застосовують до нього стиль в залежності від його статусу.

**:first-child** застосовується до першого дочірньому елементу селектора, який розташований в дереві елементів документа.

Приклад:

```
<head>
  <meta charset="utf-8">
  <title>Псевдоклассы</title>
  <style type="text/css">
    B:first-child {
      color: red; /* Красный цвет текста */
    }
  </style>
</head>
<body>
  <p><b>Lorem ipsum</b> dolor sit amet, <b>consectetur</b>
  adipiscing <b>elit</b>, sed diam nonummy nibh euismod tincidunt
  ut laoreet dolore magna aliquam erat volutpat.</p>
  <p><b>Ut wisis enim</b> ad minim veniam, <b>quis nostrud</b>
  exerci tution ullamcorper suscipit lobortis nisl ut aliquip ex ea
  <b>commodo
  consequat</b>.</p>
</body>
```

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**

#### 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:

```
<!DOCTYPE HTML>  
<html>  
<head>  
<meta charset="utf-8">
```

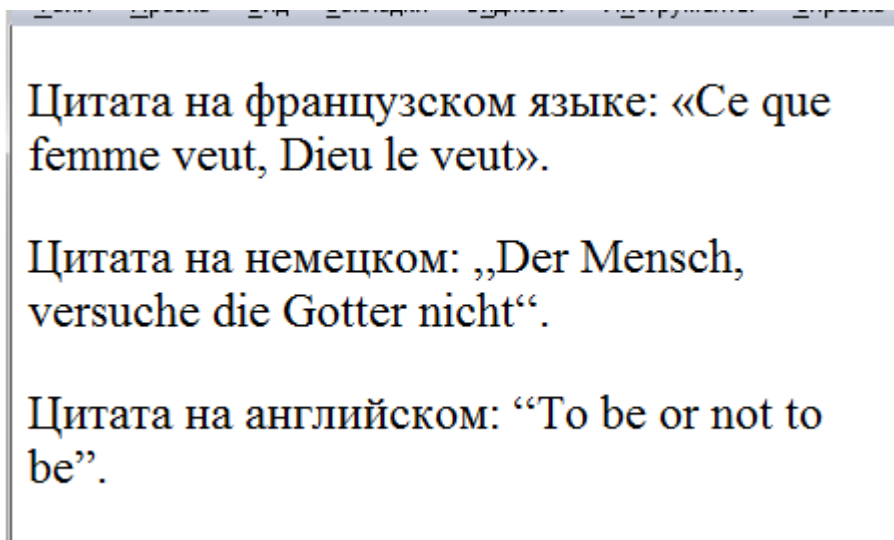


```

<title>lang</title>
<style>
P {
font-size: 150%; /* Размер текста */
}
q:lang(de) {
quotes: "\201E" "\201C"; /* Вид кавычек для немецкого языка */
}
q:lang(en) {
quotes: "\201C" "\201D"; /* Вид кавычек для английского языка */
}
q:lang(fr), q:lang(ru) { /* Вид кавычек для русского и французского
языка */
quotes: "\00AB" "\00BB";
}
</style>
</head>
<body>
<p>Цитата на французском языке: <q lang="fr">Ce que femme veut,
Dieu le veut</q>.</p>
<p>Цитата на немецком: <q lang="de">Der Mensch, versuche die
Gotter nicht</q>.</p>
<p>Цитата на английском: <q lang="en">To be or not to be</q>.</p>
</body>
</html>

```

Результат:



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:

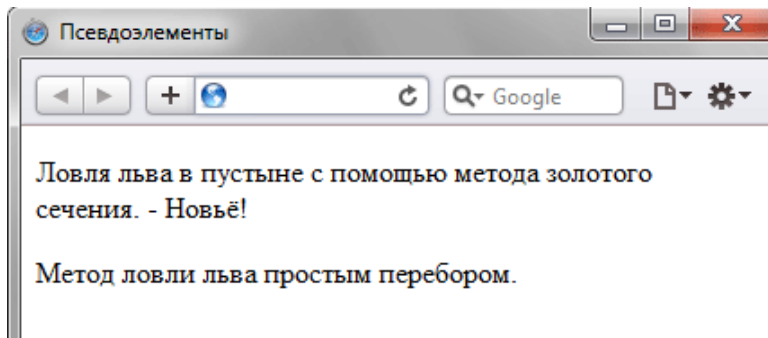
```
<head>
  <meta charset="utf-8">
  <title>Псевдоэлементы</title>
  <style>
    P.new:after {
      content: " - Новьё!"; /* Добавляем после текста абзаца */
    }
  </style>
```

```

</head>
<body>
  <p class="new">Ловля льва в пустыне с помощью метода золотого
сечения.</p>
  <p>Метод ловли льва простым перебором.</p>
</body>

```

Результат:



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

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

### : before

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

Приклад

```

<html>
<head>
  <meta charset="utf-8">
  <title>Псевдоэлементы</title>
  <style>
    UL {
      padding-left: 0; /* Убираем отступ слева */
      list-style-type: none; /* Прячем маркеры списка */
    }
  </style>

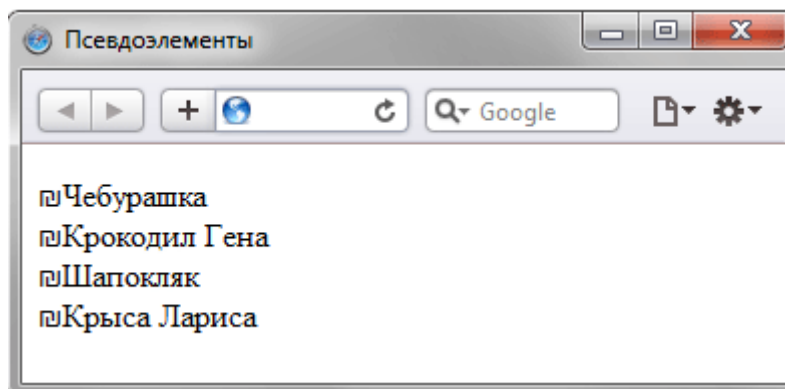
```

```

    LI:before {
        content: "\20aa "; /* Добавляем перед элементом списка символ в
юникоде */
    }
</style>
</head>
<body>
<ul>
<li>Чебурашка</li>
<li>Крокодил Гена</li>
<li>Шапокляк</li>
<li>Крыса Лариса</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.

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

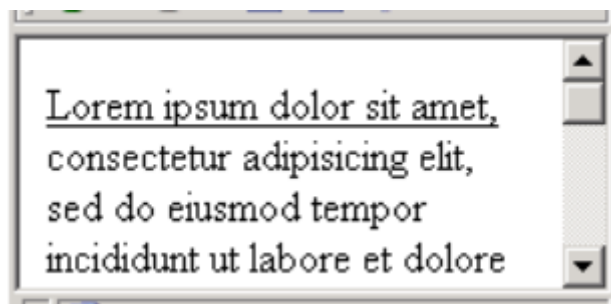
**`:first-line`**

Застосовується для блочних елементів. Задає форматування першого рядка тексту.

До псевдоелемента: **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 i word-spacing.**

Приклад:

```
<html>
<head>
<title>Пример</title>
<style>
P:first-line {text-decoration: underline}
</style>
</head>
<body>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. </p>
</body>
</html>
```



### : first-letter

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

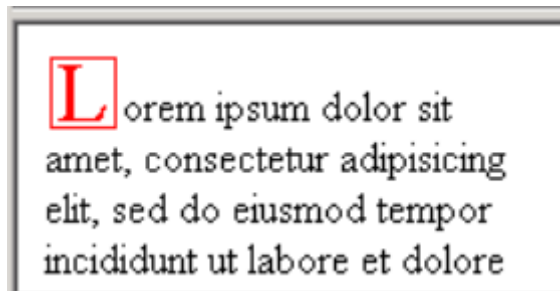
Приклад:

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

```

<title>Пример</title>
<style>
P:first-letter {
color: red;
font-size: 200%;
border: red solid 1px;
padding: 2px;
margin: 2px;
}
</style>
</head>
<body>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. </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>.

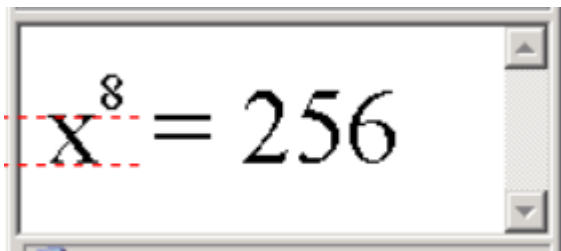
HTML-код:

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

```
<span style = "font-size: 50%; position: relative; top: - 1em;"> 8 </ span> =  
256
```

```
</ 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.

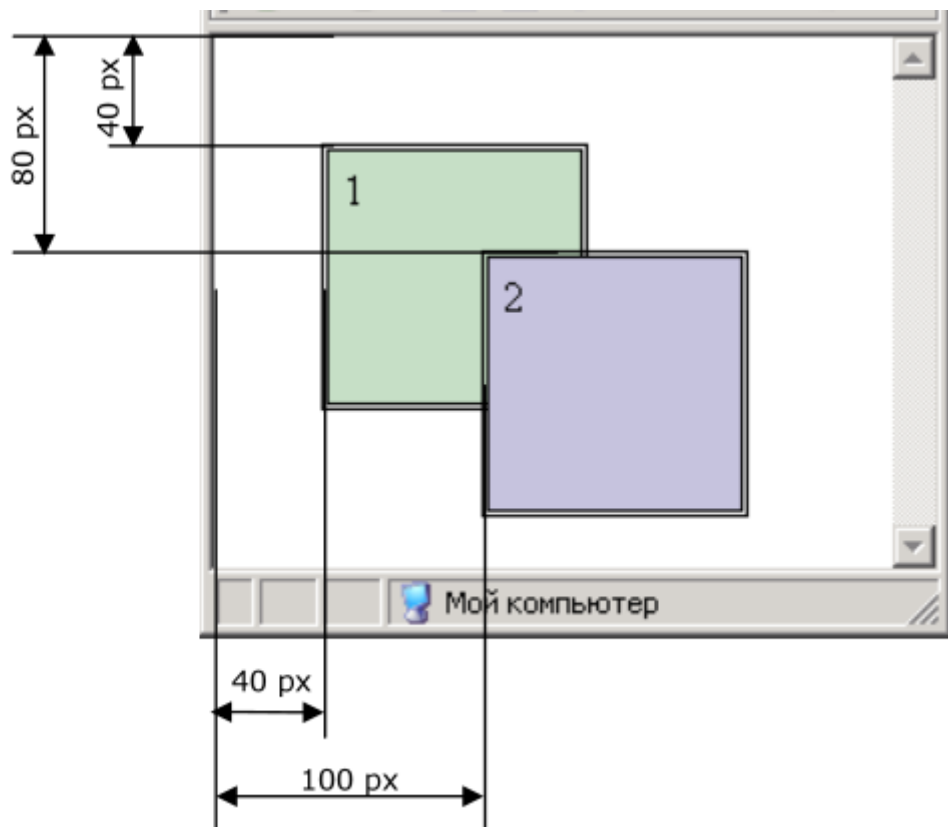
Приклад:

```
<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>

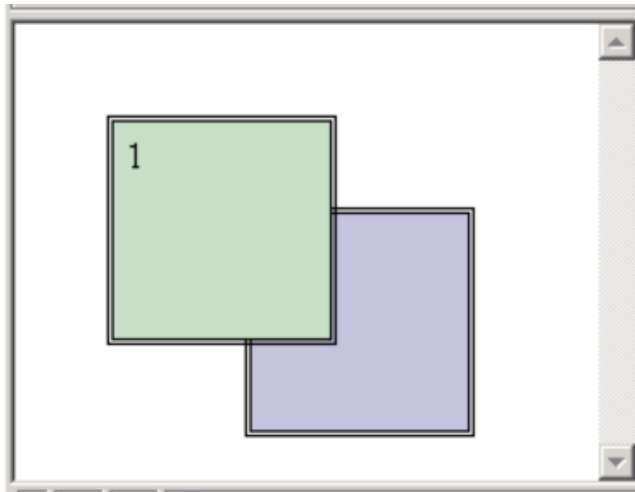
```



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 стороні 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:

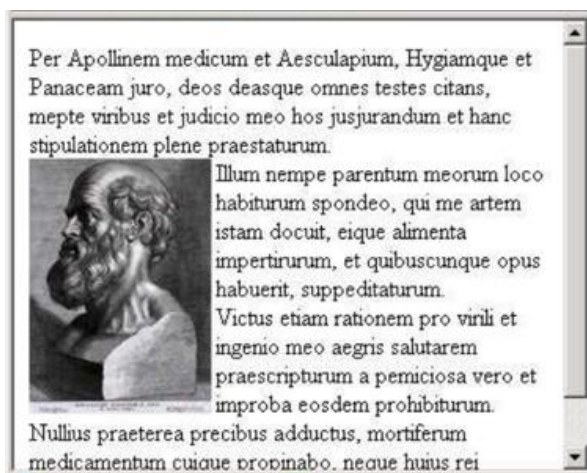
```
<html>  
<head>
```

```

<title>Плавающие элементы</title>
<style>
DIV#floating {
float: left;
}
</style>
</head>
<body>
Per Apollinem<br>
<div id="floating"></div>
Illum nempe <br>
Victus etiam <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.

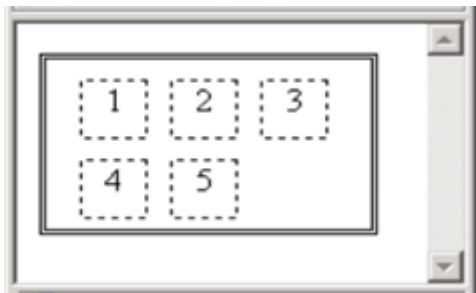
Приклад:

```

<html>
<head>
<title>Плавающие элементы</title>
<style>
DIV#main {
border: double black 3px;
width: 150px;

```

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



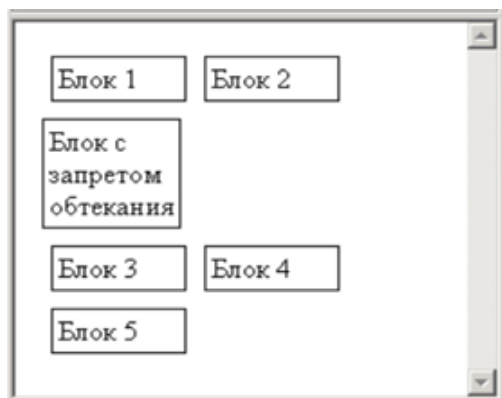
Ще однією властивістю, пов'язаною з плаваючими елементами, є **clear**. **Clear** забороняє обтікання елемента з лівої (**left**), правої (**right**) або з обох сторін (**both**). За замовчуванням значення - none - обтікання дозволено.

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">Блок 1</div>
<div class="floating">Блок 2</div>
<div style="clear: both">Блок с запретом обтекания</div>
<div class="floating">Блок 3</div>
<div class="floating">Блок 4</div>
<div class="floating">Блок 5</div>
</body>
```

`</html>`

Результат:



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

## Лекція 8-9

### Основи JavaScript [10]

#### Що таке JavaScript?

JavaScript було створено для «створення веб-сторінок живими».

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

Скрипти розповсюджуються та виконуються як простий текст. Їм не потрібна спеціальна підготовка або компіляція для запуску.

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

У браузері є власний «вижок», який іноді називається «віртуальна машина JavaScript».

Різні движки мають різні «кодові імена».

Наприклад:

V8 - в Chrome і Opera.

SpiderMonkey - в Firefox.

#### Як працюють движки?

Движок (вбудований, якщо це браузер) читає («парсить») текст скрипта.

Потім він переробляє («компілює») скрипт в машинну мову.

Після цього машинний код запускається і працює досить швидко.

Движок застосовує оптимізацію на кожному етапі. Він навіть проглядає компільований скрипт під час його роботи, аналізує дані, що проходять через нього і застосовує оптимізацію до машинного коду, покладаючись на отримані знання. У результаті скрипти працюють дуже швидко.

### Що може JavaScript у браузері?

Сучасний JavaScript - це «безпечна» мова програмування. Вона не надає низькорівневий доступ до пам'яті або процесору, тому що спочатку була створена для браузерів, які цього не вимагають.

Можливості JavaScript сильно залежать від оточення, коли він працює. Наприклад, Node.JS підтримує функції зв'язку / запису файлів, виконання мережових запитів тощо.

У браузері для JavaScript доступне все, що пов'язане з маніпулюванням веб-сторінками, взаємодією з користувачем та веб-сервером.

Наприклад, у браузері JavaScript може:

Додати новий HTML-код на сторінку, змінити наявний вміст, модифікувати стилі.

Реагувати на дії користувача, мишку, переміщення вказівника, натискання клавіші.

Відправити мережові запити на віддалений сервер, скачувати та завантажувати файли (технології AJAX та COMET).

Отримати та встановити куки, задати питання відвідувачам, показати повідомлення.

Запам'ятовувати дані на стороні клієнта («локальне сховище»).

### Чого НЕ може JavaScript у браузері?

Можливості JavaScript у браузері обмежені безпекою користувача. Мета полягає в запобіганні доступу недобросовісної веб-сторінки до особистої інформації або нанесення шкоди даними користувача.

Приклади таких обмежень включають у себе:

JavaScript на веб-сторінці не може читати / записувати довільні файли на жорсткому диску, копіювати їх або запускати програми. Вона не має прямого доступу до системних функцій ОС.

Сучасні браузери дозволяють їй працювати з файлами, але з обмеженим доступом і пропонують його, лише якщо користувач виконує визначені дії, наприклад, як «перетаскування» файлу у вікно браузера або його вибір за допомогою тега `<input>`.

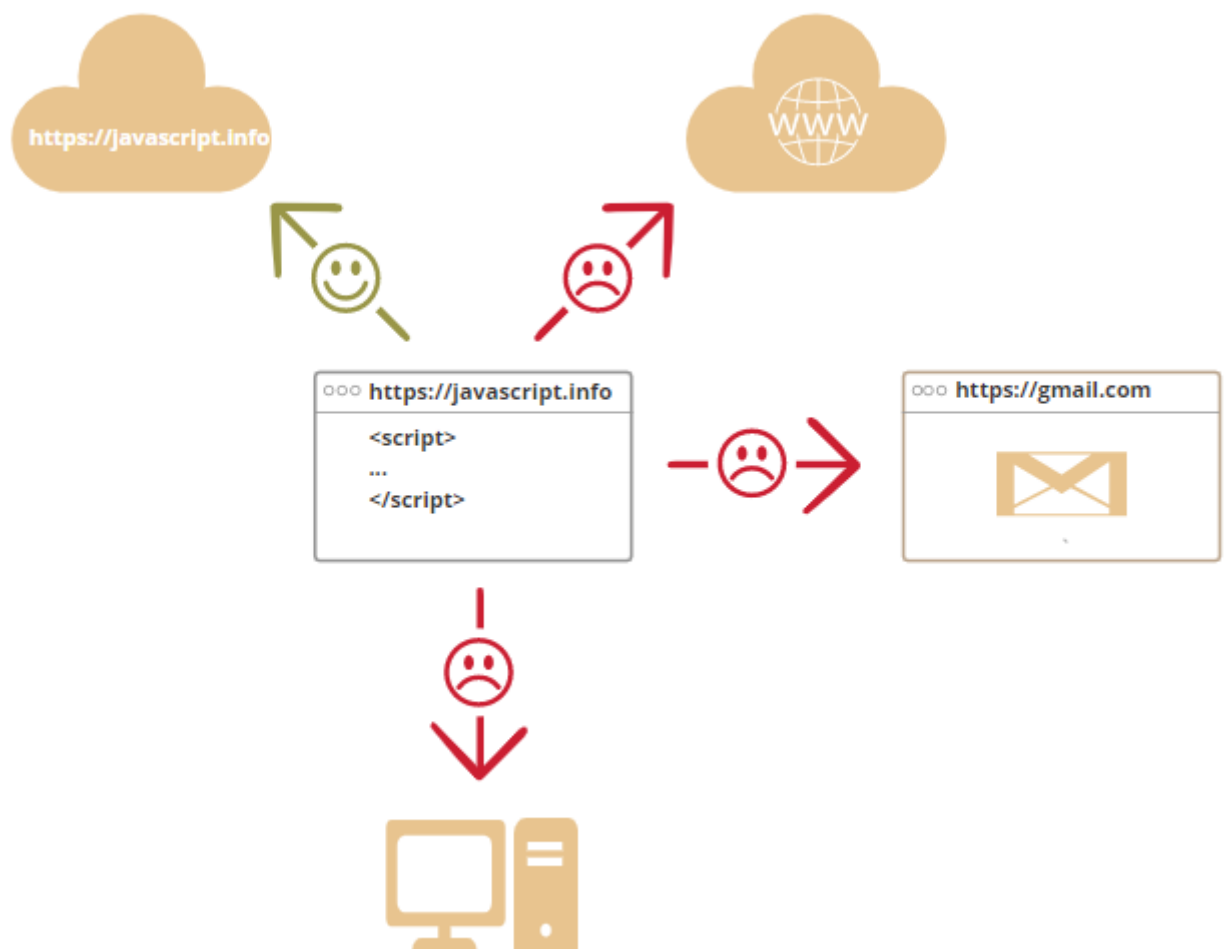
Існують способи взаємодії з камерою / мікрофоном та іншими пристроями, але вони не потребують явного дозволу користувача. Таким чином, сторінка з підтримкою JavaScript не може самостійно включити веб-камеру, спостерігати за тим, що відбувається та надсилати інформацію у ФСБ.

Різні вікна / вкладки не знають одна про одну. Іноді одне вікно, за допомогою JavaScript, відкриває інше вікно. Навіть у цьому випадку, JavaScript з однієї сторінки не має доступу до іншої, якщо вони прийшли з різних сайтів (з іншого домену, протоколу чи порталу).

Це називається «Політика однакового джерела» (Same Origin Policy). Щоб обійти це обмеження, будь-які сторінки повинні погодитись із цим і містити JavaScript-код, який спеціальним чином здійснює обмін даними.



JavaScript може легко взаємодіяти з сервером, з якого прийшла поточна сторінка. Але здатність отримувати дані з інших сайтів / доменів обмежена. Хоча це можливо в принципі, для чого потрібно явне погодження (відображене в заголовках HTTP) з віддаленою стороною. Знов ж, це обмеження безпеки.



Що робить JavaScript особливим?

Повна інтеграція з HTML / CSS.

Прості речі робляться просто.

Підтримується всі основні браузеры та включений за умовою.

JavaScript - це унікальна браузерна технологія, що поєднує в собі всі ці три речі.

Хоча, JavaScript дозволяє робити додатки не лише у браузерах, але і на серверах, на мобільних пристроях і т. Д.

### Мови «над» JavaScript

Синтаксис JavaScript підходить не під усі потреби. Різні люди хочуть мати різні можливості.

Це природно, тому що проекти різні та вимоги до них теж різні.

Так, в останній час з'явилося багато нових мов, які транслюються (конвертуються) в JavaScript, перш ніж запускатися в браузері.

Сучасні інструменти здійснюють передачу дуже швидко і прозоро, фактично дозволяючи розробникам писати код на іншій мові, автоматизовано перетворюючи його в JavaScript.

Приклади таких мов:

- CoffeeScript, він вводить більш короткий синтаксис, який дозволяє писати чистий і лаконічний код. Зазвичай таке подобається Ruby-програмістам.

- TypeScript концентрується на додаванні «суворої типізації» для спрощення розробки та підтримки великих і складних систем. Розроблено Microsoft.

- Flow теж додає типізацію, але інакше. Розроблено Facebook.

- Dart стоїть особно, бо має власний движок, що працює поза браузерами (наприклад, в мобільних додатках). Спочатку був запропонований Google, як заміна JavaScript, але на даний момент необхідна

його транспіляція (компіляція) для запуску так само, як для перерахованих вище мов.

– Brython транспілює Python в JavaScript, що дозволяє писати програми на чистому Python без JavaScript.

### Редактори коду

Більшу частину свого робочого часу програмісти проводять в редакторах коду.

Є два основних типи редакторів: IDE і «легкі» редактори. Багато хто використовує по одному інструменту кожного типу.

IDE (Integrated Development Environment, «інтегроване середовище розробки») потужні редактори з безліччю функцій, які працюють в рамках цілого проекту. Як видно з назви, це не просто редактор, а щось більше.

IDE завантажує проект (який може складатися з безлічі файлів), дозволяє перемикатися між файлами, пропонує автодоповнення по коду всього проекту (а не тільки відкритого файлу), також вона інтегрована з системою контролю версій (наприклад, такою як git), середовищем для тестування та іншими інструментами на рівні всього проекту.

Якщо ви ще не обрали собі IDE, придивіться до цих:

Visual Studio Code (безкоштовно).

WebStorm (платно).

Обидві IDE - Кросплатформені.

Для Windows є ще Visual Studio (не плутати з Visual Studio Code). Visual Studio - це потужне середовище розробки, яка працює тільки на Windows.

Вона добре підходить для .NET платформи. У неї є безкоштовна версія, яка називається Visual Studio Community.

Багато IDE платні, але у них є пробний період. Їх ціна зазвичай незначна в порівнянні з зарплатою кваліфікованого розробника, так що пробуйте і вибирайте ту, що вам підходить краще за інших.

#### «Легкі» редактори

«Легкі» редактори менш потужні, ніж IDE, але вони відрізняються швидкістю, зручним інтерфейсом і простотою.

В основному їх використовують для того, щоб швидко відкрити і відредагувати потрібний файл.

Головна відмінність між «легким» редактором і IDE полягає в тому, що IDE працює на рівні цілого проекту, тому вона завантажує більше даних при запуску, аналізує структуру проекту, якщо це необхідно, і так далі. Якщо ви працюєте тільки з одним файлом, то набагато швидше відкрити його в «легкому» редакторі.

На практиці «легкі» редактори можуть мати безліч плагінів, включаючи автодоповнення і аналізатори синтаксису на рівні директорії, тому кордону між IDE і «легкими» редакторами розмиті.

Наступні варіанти заслуговують вашої уваги:

Atom (багатоплатформовий, безкоштовний).

Sublime Text (багатоплатформовий, умовно-безкоштовний).

Notepad ++ (Windows, безкоштовний).

Vim і Emacs теж хороші, якщо знати, як ними користуватися.

## Консоль розробника

Код вразливий для помилок. Але за замовчуванням у браузері помилки не видно. Тобто, якщо щось піде не так, ми не побачимо, що саме зламалося, і не зможемо це полагодити.

Для вирішення завдань такого роду в браузер вбудовані так звані «Інструменти розробки» (Developer tools або скорочено - devtools).

Chrome і Firefox здобули любов переважної більшості програмістів багато в чому завдяки своїм відмінним інструментів розробники. Решта браузерів, хоча і мають подібні інструменти, але все ж найчастіше знаходяться в ролі наздоганяючих і за якістю, і за кількістю властивостей і особливостей. Загалом, майже у всіх програмістів є свій «улюблений» браузер. Інші використовуються тільки для вилову і виправлення специфічних «браузерозалежних» помилок.

Для початку знайомства з цими потужними інструментами давайте з'ясуємо, як їх відкривати, дивитися помилки і запускати команди JavaScript.

## Google Chrome

Запустіть код:

```
<!DOCTYPE HTML>
```

```
<html>
```

```
<head>
```

```
<meta charset="utf-8">
```

```
</head>
```

```
<body>
```

*На этой странице есть ошибка в скрипте.*

```
<script>
```

```
  lalala
```

```
</script>
```

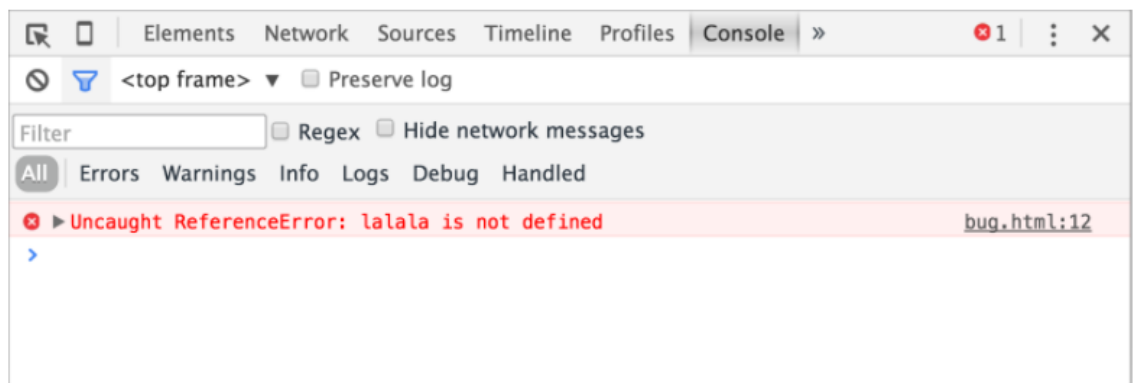
```
</body>
```

```
</html>
```

В JavaScript-кодi закралася помилка. Її не видно звичайному відвідувачу, тому давайте знайдемо її за допомогою інструментів розробки.

Натисніть F12 або, якщо ви використовуєте Mac, Cmd + Opt + J.

За замовчуванням в інструментах розробника відкриється вкладка Console (консоль). Вона виглядає приблизно наступним чином:



Точний зовнішній вигляд інструментів розробки залежить від використовуваної версії Chrome. Час від часу деякі деталі змінюються, але в цілому зовнішній вигляд залишається приблизно схожим на попередні версії.

В консолі ми можемо побачити повідомлення про помилку, відмалює червоним кольором. У нашому випадку скрипт містить невідому команду «lalala».

Справа присутнє посилання на вихідний код bug.html: 12 з номером рядка коду, в якій ця помилка і сталася.

Під повідомленням про помилку знаходиться синій символ >. Він позначає командний рядок, в ньому ми можемо редагувати і запускати JavaScript-команди. Для його запуску натисніть Enter.

### Багаторядкове введення

Зазвичай при натисканні Enter введений рядок коду відразу виконується.

Щоб перенести рядок, натисніть Shift + Enter. Так можна вводити більш довгий JS-код.

### Firefox, Edge і інші

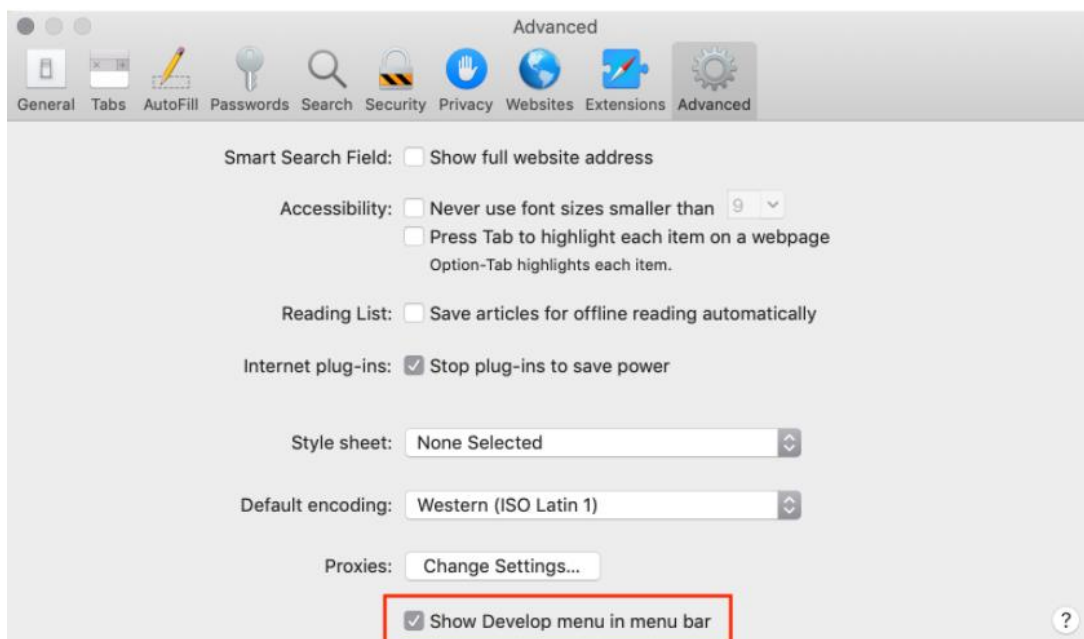
Інструменти розробника в більшості браузерів відкриваються при натисканні на F12.

Їх зовнішній вигляд і принципи роботи мало чим відрізняються. Розібравшись з інструментами в одному браузері, ви без зусиль зможете працювати з ними і в іншому.

### Safari

Safari (браузер для Mac, не підтримуються в системах Windows / Linux) все ж має невелику відмінність. Для початку роботи нам потрібно включити «Меню розробки» ( «Developer menu»).

Відкрийте Налаштування (Preferences) і перейдіть до панелі «Просунуті» (Advanced). У самому низу ви знайдете чекбокс:



Тепер консоль можна активувати натисканням клавіш `Cmd + Opt + C`. Також зверніть увагу на новий елемент меню «Розробка» ( «Develop»). У ньому міститься велика кількість команд і налаштувань.

## Основи JavaScript

Нам потрібно робоче середовище для запуску наших скриптів, браузер буде хорошим вибором. В залежності від середовища (браузер, Node.js) управління скриптами використовуються різні команди. Отже, спочатку давайте подивимося, як виконати скрипт на сторінці. Для серверних середовищ (наприклад, Node.js), ви можете виконати скрипт за допомогою команди типу `"node my.js"`. Для браузера все трохи інакше.

Тег «script»

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

Для прикладу:



```
<html>

<body>

  <p>Перед скриптом...</p>

  <script>
    alert( 'Привет' );
  </script>

  <p>...После скрипта.</p>

</body>

</html>
```

Перед скриптом...

...После скрипта.



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>
```

### На замітку:

Як правило, тільки найпростіші скрипти поміщаються в HTML. Більш складні виділяються в окремі файли.

Користь від окремих файлів в тому, що браузер завантажить скрипт окремо і зможе зберігати його в кеші.

Інші сторінки, які підключають той же скрипт, зможуть брати його з кеша замість повторного завантаження з мережі. І таким чином файл буде завантажуватися з сервера тільки один раз.

Це скорочує витрату трафіку і пришвидшує роботу.

Якщо атрибут `src` встановлений, вміст тега `script` буде ігноруватися.

В одному тезі `<script>` не можна використовувати одночасно атрибут `src` і код всередині.

Нижченаведений приклад не працює:

```
<script src="file.js">
  alert(1); // содержимое игнорируется, так как есть атрибут src
</script>
```

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

Вищенаведений приклад можна розділити на два скрипта:

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

Зазвичай кожен інструкцію пишуть на новому рядку, щоб код було легше читати:

```
alert('Привет');
```

```
alert('Мир');
```

### Крапка з комою

У більшості випадків крапку з комою можна не ставити, якщо є перехід на новий рядок.

Так теж буде працювати:

```
alert('Привет')
```

```
alert('Мир')
```

В цьому випадку JavaScript інтерпретує перенесення рядка як «неявну» крапку з комою. Це називається автоматична вставка крапки з комою.

У більшості випадків новий рядок має на увазі крапку з комою. Але «в більшості випадків» не означає «завжди»!

У деяких ситуаціях новий рядок все ж не означає крапку з комою. наприклад:

```
alert(3 +  
1  
+ 2);
```

Код виведе 6, тому що JavaScript не вставляє тут крапку з комою. Інтуїтивно очевидно, що, якщо рядок закінчується знаком "+", значить, це «незавершений вираз», тому крапка з комою не потрібно. І в цьому випадку все працює, як задумано.

Але є ситуації, де JavaScript «забуває» вставити крапку з комою там, де вона потрібна.

Помилки, які при цьому виникають, досить складно виявляти і виправляти.

Приклад помилки:

Якщо ви хочете побачити конкретний приклад такої помилки, зверніть увагу на цей код:

```
[1, 2].forEach(alert)
```

Поки немає необхідності знати значення дужок [] і forEach. Ми вивчимо їх пізніше. Поки що просто запам'ятайте результат виконання цього коду: виводиться 1, а потім 2.

А тепер додамо alert перед кодом і не поставимо в кінці крапку з комою:

```
alert("Сейчас будет ошибка")  
[1, 2].forEach(alert)
```

Тепер, якщо запустити код, виведеться тільки перший alert, а потім ми отримаємо помилку!

Сейчас будет ошибка

TypeError: Cannot read property '2' of undefined

Все виправиться, якщо ми поставимо крапку з комою після alert:

```
alert("Теперь всё в порядке");
```

```
[1, 2].forEach(alert) alert("Тепер все в порядку");
```

Тепер ми отримаємо повідомлення «Тепер все в порядку», слідом за яким будуть 1 і 2.

У першому прикладі без крапки з комою виникає помилка, тому що JavaScript не вставляє крапку з комою перед квадратними дужками [...]. І тому код в першому прикладі виконується, як одна інструкція. Ось як движок бачить його:

```
alert("Сейчас будет ошибка")[1, 2].forEach(alert)
```

Але це повинні бути дві окремі інструкції, а не одна. Таке злиття в даному випадку неправильне, тому й помилка. Це може статися і в деяких інших ситуаціях.

Рекомендуємо ставити крапку з комою між інструкціями, навіть якщо вони відокремлені переносами рядків. Це правило широко використовується в співтоваристві розробників. Варто відзначити ще раз - в більшості випадків можна не ставити крапку з комою. Але безпечніше, особливо для новачка, ставити її.

## Коментарі

Згодом програми стають все складніше і складніше. Виникає необхідність додавати коментарі, які б описували, що робить код і чому.

Коментарі можуть перебувати в будь-якому місці скрипта. Вони не впливають на його виконання, оскільки движок просто ігнорує їх.

Однорядкові коментарі починаються з подвійною косою рисою //.

Частина рядка після // вважається коментарем. Такий коментар може як займати рядок цілком, так і перебувати після інструкції.

Як тут:

```
// Этот комментарий занимает всю строку
alert('Привет');

alert('Мир'); // Этот комментарий следует за инструкцией
```

Багаторядкові коментарі починаються косою рисою із зірочкою /\* і закінчуються зірочкою з косою рисою \*/.

Як ось тут:

```
/* Пример с двумя сообщениями.
Это - многострочный комментарий.
*/
alert('Привет');

alert('Мир');
```

Вміст коментаря ігнорується, тому, якщо ми помістимо код всередині /\* ... \*/ , він не буде виконуватися.

Це буває зручно для тимчасового відключення ділянки коду:

```
/* Закомментировали код
alert('Привет');
```

```
*/
```

```
alert('Мир');
```

Використовуйте гарячі клавіші!

У більшості редакторів рядок коду можна закоментувати, натиснувши комбінацію клавіш Ctrl + / для однорядкового коментаря і щось на зразок Ctrl + Shift + / - для багаторядкових коментарів (виділіть шматок коду і натисніть комбінацію клавіш). В системі Mac спробуйте Cmd замість Ctrl і Option замість Shift.

Вкладені коментарі не підтримуються!

Не може бути /\*...\*/ всередині /\*...\*/.

```
/*
```

```
/* вложенный комментарий !?! */
```

```
*/
```

```
alert('Мир');
```

SyntaxError: Unexpected token '\*/'

Коментарі збільшують розмір коду, але це не проблема. Є безліч інструментів, які мініфікують код перед публікацією на робочий сервер. Вони прибирають коментарі, так що вони не містяться в робочих скриптах. Таким чином, коментарі жодним чином не шкодять робочому коду.

### Суворий режим - "use strict"

Протягом довгого часу JavaScript розвивався без проблем зі зворотною сумісністю. Нові функції додавалися в мову, в той час як стара функціональність не змінювалася.



Перевагою даного підходу було те, що існуючий код продовжував працювати. А недоліком - що будь-яка помилка або недосконале рішення, прийняте творцями JavaScript, застрявали в мові назавжди.

Щоб застарілий код працював, як і раніше, потрібно явно його активувати за допомогою спеціальної директиви: "use strict".

### *«Use strict»*

Директива виглядає як рядок: "use strict" або 'use strict'. Коли вона знаходиться на початку скрипта, весь сценарій працює в «сучасному» режимі.

Наприклад:

```
"use strict";  
  
// этот код работает в современном режиме
```

Зауважимо, що замість усього скрипта "use strict" можна поставити на початку більшості видів функцій. Це дозволяє включити строгий режим тільки в конкретній функції. Але зазвичай люди використовують його для всього файлу.

Переконайтеся, що «use strict» знаходиться на початку

Перевірте, що "use strict" знаходиться в першому виконуваному рядку скрипта, інакше строгий режим може не включитися.

Тут суворий режим не включений:

```
alert("some code");  
// "use strict" ниже игнорируется - он должен быть в первой строке  
"use strict";  
  
// строгий режим не активирован
```

Над "use strict" можуть бути записані тільки коментарі.

Немає ніякого способу скасувати use strict

Немає директиви типу "no use strict", яка повертала б движок до старої поведінки.

Як тільки ми входимо в строгий режим, скасувати це неможливо.

### Консоль браузера

В консолі браузера use strict за замовчуванням вимкнений.

Іноді, коли use strict має значення, ви можете отримати неправильні результати.

Можна використовувати Shift + Enter для введення декількох рядків і написати у верхньому рядку use strict:

```
'use strict'; <Shift+Enter для переходу на нову строку>  
// ...ваш код...  
  
<Enter для запуску>
```

У більшості браузерів, включаючи Chrome і Firefox, це працює.

У старих браузерах консоль не враховує такий use strict, там можна «обертати» код в функцію, ось так:

```
(function() {  
  'use strict';  
  
  // ...ваш код...  
  
})();
```

Чи завжди потрібно використовувати «use strict»?

Сучасний JavaScript підтримує «класи» і «модулі» - просунуті структури мови, які автоматично включають строгий режим. Тому в них немає потреби додавати директиву "use strict".

Поки дуже бажано додавати "use strict"; на початку ваших скриптів. Пізніше, коли весь ваш код буде складатися з класів і модулів, директиву можна буде опускати.

## 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'; // сохраниць строку
```

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); // показывает содержимое переменной
```

Для стислості можна поєднати оголошення змінної і запис даних в один рядок:

```
let message = 'Hello!'; // определяем переменную и присваиваем ей значение  
  
alert(message); // Hello!
```

Ми також можемо оголосити кілька змінних в одному рядку:

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

Такий спосіб може здатися коротшим, але для кращого читання оголошуйте кожну змінну на новому рядку.

Багаторядковий варіант трохи довший, але легше для читання:

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

Деякі люди також визначають кілька змінних в такому ось багаторядковому стилі:

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

або

```
let user = 'John'
```

```
, age = 25
```

```
, message = 'Hello';
```

В принципі, всі ці варіанти працюють однаково. Так що це питання особистого смаку та естетики.

var замість let

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

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

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

Є тонкі відмінності між let і var (розглянемо далі).

Наприклад, змінну message можна уявити як коробку з назвою "message" і значенням "Hello!" всередині:

Ми можемо покласти будь-яке значення в коробку.

Ми також можемо змінити його стільки разів, скільки захочемо:

```
let message;
```

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

```
message = 'World!'; // значение изменено
```

```
alert(message);
```

При зміні значення старі дані видаляються зі змінної.

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

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

```
let message;

// копіруємо значення 'Hello world' із змінної hello в
змінну message
message = hello;

// тепер дві змінні містять однакові дані
alert(hello); // Hello world!

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

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

```
let message = "Это";

// повторення ключового слова 'let' призводить до помилки

let message = "Другое"; // SyntaxError: 'message' has already been
declared
```

### Імена змінних

В JavaScript є два обмеження, що стосуються імен змінних:

1. Ім'я змінної має містити тільки букви, цифри або символи \$ і \_.
2. Перший символ не повинен бути цифрою.

Приклади допустимих імен:

```
let userName;

let test123;
```

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

Найцікавіше - знак долара '\$' і підкреслення '\_' також можна використовувати в назвах. Це звичайні символи, як і букви, без будь-якого особливого значення.

Ці імена є допустимими:

```
let $ = 1; // об'явили переменную с именем "$"  
let _ = 2; // а теперь переменную с именем "_"
```

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

Приклади неправильних імен змінних:

```
let 1a; // не может начинаться с цифры
```

```
let my-name; // дефис '-' не разрешён в имени
```

Регістр має значення

Змінні з іменами apple і AppLE - це дві різні змінні.

Нелатинські букви дозволені, але не рекомендуються

Можна використовувати будь-яку мову, включаючи кирилицю або навіть ієрогліфи, наприклад:

```
let имя = '...';
```

```
let 我 = '...';
```

Технічно тут немає помилки, такі імена дозволені, але є міжнародна традиція використовувати англійську мову в іменах змінних. Навіть якщо ми пишемо невеликий скрипт, у нього може бути довге життя попереду. Людям з інших країн, можливо, доведеться прочитати його не один раз.

Зарезервовані імена

Існує список зарезервованих слів, які не можна використовувати в якості імен змінних, тому що вони використовуються самим мовою.

Наприклад: `let`, `class`, `return` і `function` зарезервовані.

Наведений нижче код дає синтаксичну помилку:

```
let let = 5; // нельзя назвать переменную "let", ошибка!
```

```
let return = 5; // также нельзя назвать переменную "return",  
ошибка!
```

### Константи

Щоб оголосити константну, використовуйте `const` замість `let`:

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

Змінні, оголошені за допомогою `const`, називаються «константами». Їх не можна змінити. Спроба зробити це призведе до помилки:

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

```
myBirthday = '01.01.2001'; // ошибка, константу нельзя  
перезаписать!
```

Якщо програміст впевнений, що змінна ніколи не буде змінюватися, він може гарантувати це і наочно донести до кожного, оголосивши її через `const`.

### Константи в верхньому регістрі

Широко поширена практика використання констант в якості псевдонімів для важко запам'ятовуються значень, які відомі до початку виконання скрипта.



Назви таких констант пишуться з використанням заголовних букв і підкреслення.

Наприклад, зробимо константи для різних кольорів в «шістнадцятковому форматі»:

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

// ...когда нам нужно выбрать цвет
let color = COLOR_ORANGE;

alert(color); // #FF7F00
```

Переваги:

- COLOR\_ORANGE набагато легше запам'ятати, ніж "# FF7F00".
- Набагато легше припуститися помилки при введенні "# FF7F00", ніж при введенні COLOR\_ORANGE.
- При читанні коду COLOR\_ORANGE набагато зрозуміліше, ніж # FF7F00.

Константи з іменами, записаними великими літерами, використовуються тільки як псевдоніми для «жорстко закодованих» значень.

let і const поведуться однаково по відношенню до лексичного оточення, області видимості.

### 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 = "глобальная"; // Глобальная переменная
function checkscope() {
    var x = "локальная"; // Локальная переменная с тем же именем,
что и у глобальной
    document.write(x); // Используется локальная переменная, а
не глобальная
}
checkscope(); // => "локальная"
```

### 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 = "Привет"; // локальная переменная, "var" вместо
"let"

  alert(phrase); // Привет
}

sayHi();

alert(phrase); // Ошибка: phrase не определена

```

... 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; // используем var вместо let
}

alert(test); // true, переменная существует вне блока 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; // используем 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, переменная i доступна вне цикла, т.к. является  
глобальной переменной
```

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

```
function sayHi() {  
  if (true) {  
    var phrase = "Привет";  
  }  
  
  alert(phrase); // срабатывает и выводит "Привет"  
}  
  
sayHi();  
  
alert(phrase); // Ошибка: phrase не определена (видна в консоли  
разработчика)
```

Як ми бачимо, `var` виходить за межі блоків `if`, `for` і подібних. Це відбувається тому, що на зорі розвитку JavaScript блоки коду не мали лексичного оточення.

### «`var`» обробляються на початку запуску функції

Оголошення змінних `var` обробляються на початку виконання функції (або запуску скрипта, якщо змінна є глобальною).

Іншими словами, змінні `var` вважаються оголошеними з самого початку виконання функції незалежно від того, в якому місці функції реально знаходяться їх оголошення (за умови, що вони не перебувають у вкладеній функції).

Тобто цей код:

```
function sayHi() {  
  phrase = "Привет";  
  
  alert(phrase);  
  
  var phrase;  
}  
  
sayHi();
```

... Технічно повністю еквівалентний наступному (оголошення змінної `var phrase` переміщено в початок функції):

```
function sayHi() {  
  var phrase;  
  
  phrase = "Привет";  
  
  alert(phrase);  
}  
  
sayHi();
```

... І навіть коду нижче (як ви пам'ятаєте, блокова область видимості ігнорується):

```
function sayHi() {  
    phrase = "Привет"; // (*)  
  
    if (false) {  
        var phrase;  
    }  
  
    alert(phrase);  
}  
  
sayHi();
```

Це поведінка називається «hoisting» (спливання, підняття), тому що всі оголошення змінних var «спливають» в самий верх функції.

Оголошення змінних «спливають», але привласнення значень - немає.

Це найпростіше продемонструвати на прикладі:

```
function sayHi() {  
    alert(phrase);  
  
    var phrase = "Привет";  
}  
  
sayHi();
```

Рядок var phrase = "Привіт" складається з двох дій:

1. Оголошення змінної var
2. Присвоєння значення в змінну =.

Оголошення змінної обробляється на початку виконання функції («спливає»), проте привласнення значення завжди відбувається в тому рядку коду, де воно зазначено. Тобто код виконується за наступним сценарієм:

```
function sayHi() {
```

```
var phrase; // объявление переменной срабатывает  
вначале...  
alert(phrase); // undefined  
phrase = "Привет"; // ...присвоение - в момент, когда  
исполнится данная строка кода.  
}  
  
sayHi();
```

Оскільки всі оголошення змінних `var` обробляються на початку функції, ми можемо посилатися на них в будь-якому місці. Однак, змінні мають значення `undefined` до рядка з присвоєнням значення.

В прикладі вище виклик `alert` відбувався без помилки, тому що змінна `phrase` вже існувала. Але її значення ще не було присвоєно, тому ми отримували `undefined`.

Існує 2 основні відмінності `var` від `let` / `const`:

1. Змінні `var` не мають блокової області видимості, вони обмежені, як мінімум, тілом функції.
2. Оголошення (ініціалізація) змінних `var` відбувається на початку виконання функції (або скрипта для глобальних змінних).

## Лекція 10-11

### Типи даних [10]

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 будет ошибкой  
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 цілочислові значення, так і числа з плаваючою крапкою.

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

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

Infinity є математичною нескінченність  $\infty$ . Це особливе значення, яке більше будь-якого числа.



Ми можемо отримати його в результаті поділу на нуль:

```
alert( 1 / 0 ); // Infinity
```

Або поставити його явно:

```
alert( Infinity ); // Infinity
```

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

```
alert( "не число" / 2 ); // NaN, такое деление является ошибкой
```

Значення NaN «чіпляється». Будь-яка операція з NaN повертає NaN:

```
alert( "не число" / 2 + 5 ); // NaN
```

Якщо десь в математичному виразі є NaN, то результатом обчислень з його участю буде NaN.

### Математичні операції - безпечні

Математичні операції в JavaScript «безпечні». Ми можемо робити що завгодно: ділити на нуль, звертатися з нечисловими рядками як з числами і т.д.

Скрипт ніколи не зупиниться з фатальною помилкою (не "помре»). У гіршому випадку ми отримаємо NaN як результат виконання.

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

### BigInt

В JavaScript тип «number» не може містити числа більше, ніж  $(2^{53}-1)$  (т.б. 9007199254740991), або менше, ніж  $-(2^{53}-1)$  для від'ємних чисел. Це технічне обмеження викликане їх внутрішнім поданням.

Для більшості випадків цього достатньо. Але іноді нам потрібні дійсно гігантські числа, наприклад, в криптографії або при використанні мітки часу («timestamp») з мікросекундами.

Тип BigInt був доданий в JavaScript, щоб дати можливість працювати з цілими числами довільної довжини.

Щоб створити значення типу BigInt, необхідно додати n в кінець числового літерала:

```
// символ "n" в конце означает, что это BigInt  
const bigInt = 1234567890123456789012345678901234567890n;
```

В даний момент BigInt підтримується тільки в браузерах Firefox, Chrome і Edge, але не підтримується в Safari і IE.

### Рядок

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

```
let str = "Привет";  
let str2 = 'Одинарные кавычки тоже подойдут';  
let phrase = `Обратные кавычки позволяют встраивать переменные  
${str}`;
```

В JavaScript існує три типи лапок:

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

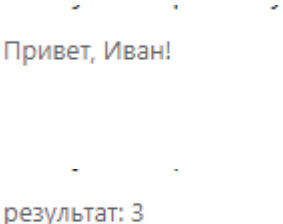
Подвійні або одинарні лапки є «простими», між ними немає різниці в JavaScript.

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
```



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)

```
alert( "результат: ${1 + 2}" ); // результат: ${1 + 2} (двойные кавычки ничего не делают)
```

Немає окремого типу даних для одного символу.

У мовах C і Java це char.

В JavaScript подібного типу немає, є тільки тип `string`. Рядок може містити нуль символів (бути порожній), один символ або безліч.

### 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; // да, поле отмечено
let ageFieldChecked = false; // нет, поле не отмечено
```

Булеві значення також можуть бути результатом порівнянь:

```
let isGreater = 4 > 1;
alert( isGreater ); // true (результатом сравнения будет "да")
```

### 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 показником», як в деяких інших мовах.

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

У наведеному вище коді вказано, що значення змінної age 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); // виведет "undefined"
```

Технічно ми можемо присвоїти значення undefined будь-якій змінній:

```
let age = 123;  
  
// изменяем значение на undefined  
age = undefined;  
  
alert(age); // "undefined"  
  
alert (age); // "undefined"
```

... Але так робити не рекомендується. Зазвичай null використовується для присвоєння змінної «порожнього» або «невідомого» значення, а undefined - для перевірок, була змінна призначена.

### 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", // под ключом "name" хранится значение "John"
  age: 30 // под ключом "age" хранится значение 30
};
```

У кожної властивості є ключ (також званий «ім'ям» або «ідентифікатор»). Після імені властивості слідує двокрапка ":", і потім вказується значення

властивості. Якщо в об'єкті є кілька властивостей, то вони перераховуються через кому.

В об'єкті user зараз знаходяться два властивості:

Перша властивість з ім'ям "name" і значенням "John".

Друга властивість з ім'ям "age" і значенням 30.

Можна сказати, що наш об'єкт user - це ящик з двома папками, підписаними «name» і «age».

Ми можемо в будь-який момент додати в нього нові папки, видалити папки або прочитати вміст будь-якої папки.

Для звернення до властивостей використовується запис «через крапку»:

```
// получаем свойства объекта:  
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;
```

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

```
// это вызовет синтаксическую ошибку  
  
user.likes birds = true
```

JavaScript бачить, що ми звертаємося до властивості `user.likes`, а потім йде незрозуміле слово `birds`. В результаті синтаксична помилка.

Крапка вимагає, щоб ключ був іменований за правилами іменування змінних. Тобто не мав прогалін, не розпочинався з цифри і не містив спеціальні символи, крім `$` і `_`.

Для таких випадків існує альтернативний спосіб доступу до властивостей через квадратні дужки. Такий спосіб спрацює з будь-яким ім'ям властивості:

```
let user = {};  
  
// присваивание значения свойству  
user["likes birds"] = true;  
  
// получение значения свойства  
alert(user["likes birds"]); // true  
  
// удаление свойства  
  
delete user["likes birds"];
```

## 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()`:

```
// Создаём новый символ - id  
  
let id = Symbol();
```



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

```
// Создаём символ id с описанием (именем) "id"
```

```
let id = Symbol("id");
```

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

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

```
let id1 = Symbol("id");
```

```
let id2 = Symbol("id");
```

```
alert(id1 == id2); // false
```

Символи не перетворюються автоматично в рядки.

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

## 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)
```

Останні три рядки потребують поясненні:

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

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

Виклик `typeof alert` повертає `"function"`, тому що `alert` є функцією. В JavaScript немає спеціального типу «функція». Функції відносяться до об'єктного типу. Але `typeof` обробляє їх особливим чином, повертаючи `"function"`. Так теж повелося від створення JavaScript. Формально це невірно, але може бути зручним на практиці.

### 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 [...]

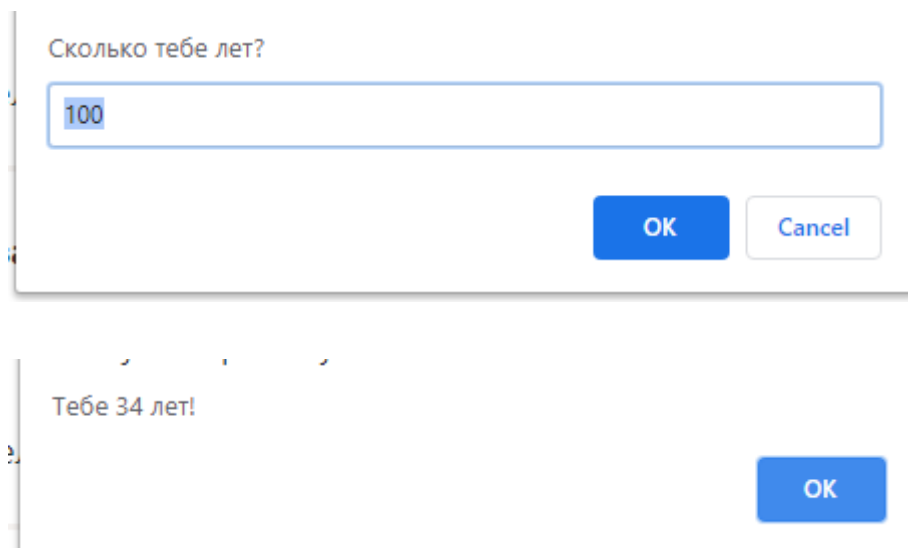
Квадратні дужки навколо default в описаному вище синтаксисі означають, що параметр факультативний, необов'язковий.

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('Скільки тебе лет?', 100);  
alert(`Тебе ${age} лет!`); // Тебе 100 лет!
```

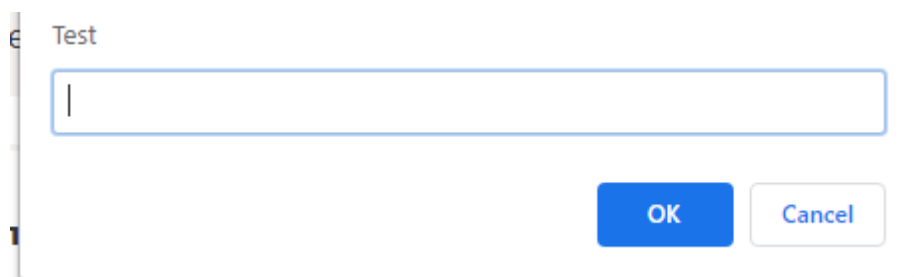


Для ІЕ: завжди встановлюйте значення за замовчуванням

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);
```

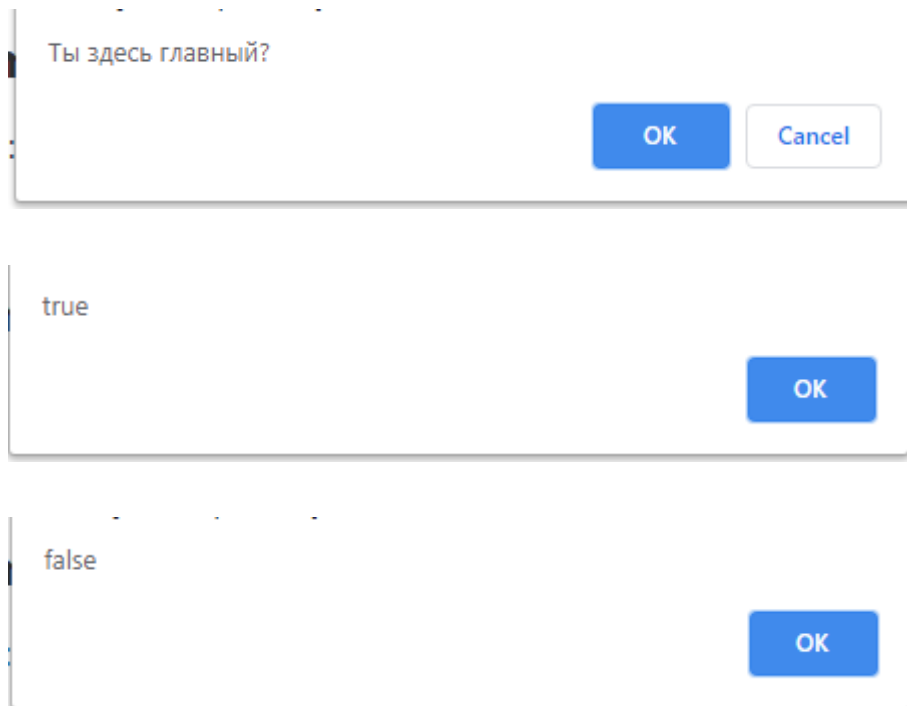
Функція confirm відображає модальне вікно з текстом питання question і двома кнопками: ОК і Скасування.

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

Наприклад:

```
let isBoss = confirm("Ты здесь главный?");
```

```
alert( isBoss ); // true, если нажата OK
```



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

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

Підпорядковується правилам:

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

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 прогалин типу " " при логічному перетворенні завжди true.

Приклад

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



## Базові оператори

Терміни: «унарний», «бінарний», «операнд»

Перш, ніж ми рушимо далі, давайте розберемося з термінологією.

Операнд - те, до чого застосовується оператор. Наприклад, в множенні  $5 * 2$  є два операнди: лівий операнд дорівнює 5, а правий операнд дорівнює 2. Іноді їх називають «аргументами» замість «операндів».



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, применили унарный минус
```

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, бинарный минус вычитает значения
```

Формально, в останніх прикладах ми говоримо про два різні оператори, що використовують один символ: оператор заперечення (унарний оператор, який звертає знак) і оператор віднімання (бінарний оператор, який віднімає одне число з іншого).

Підтримуються наступні математичні оператори:

Додавання +,

Віднімання -,

Множення \*,

Розподіл /,

Взяття залишку від ділення %,

Піднесення до ступеню \*\*

## 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 влияет на числа  
let x = 1;  
alert( +x ); // 1
```

```
let y = -2;  
alert( +y ); // -2
```

```
// Преобразует 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", так как бинарный плюс объединяет  
строки
```

Тому використовуємо унарний плюс, щоб перетворити до числа:

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

```
// оба операнда предварительно преобразованы в числа  
alert( +apples + +oranges ); // 5
```

```
// более длинный вариант
```

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

З точки зору математика, такий достаток плюсів виглядає дивним. Але з точки зору програміста тут немає нічого особливого: спочатку виконуються унарні плюси, які приведуть рядки до чисел, а потім бінарний '+' їх складе.

### Присвоєння

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

```
let x = 2 * 2 + 1;  
  
alert( x ); // 5
```

### Присвоєння = повертає значення

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 присвоювання не є винятком.

Виклик `x = value` записує `value` в `x` і повертає його.

Завдяки цьому присвоювання 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.

### Присвоєння по ланцюжку

Розглянемо ще одну цікаву можливість: ланцюжок присвоювання.

```
let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4
alert( b ); // 4

alert( c ); // 4
```

Таке присвоювання працює справа наліво. Спочатку обчислюється саме праве вираз  $2 + 2$ , і потім результат присвоюється змінним зліва: `c`, `b` і `a`. В кінці у всіх змінних буде одне значення.

### 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++;      // работает как counter = counter + 1, просто
запись короче

alert( counter ); // 3
```

Декремент - зменшує змінну на 1:

```
let counter = 2;
counter--;     // работает как counter = counter - 1, просто
запись короче

alert( counter ); // 1
```

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

Оператори `++` і `--` - можуть бути розташовані не тільки після, але і до змінної.

Коли оператор йде після змінної - це «Постфіксна форма»: `counter++`.

«Префіксна форма» - це коли оператор іде перед змінною: `++counter`.

Обидві ці інструкції роблять одне і те ж: збільшують `counter` на 1.

## Побітові оператори

Працюють з 32-розрядними цілими числами (при необхідності приводять до них), на рівні їх внутрішнього двійкового представлення.

Ці оператори не є чимось специфічним для JavaScript, вони підтримуються в більшості мов програмування.

Підтримуються наступні побітові оператори:

AND (і) (&)

OR (або) (|)

XOR (побітове виключає або) (^)

NOT (не) (~)

LEFT SHIFT (лівий зсув) (<<)

RIGHT SHIFT (праве зрушення) (>>)

ZERO-FILL RIGHT SHIFT (праве зрушення із заповненням нулями)  
(>>>)

## 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` означає 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».

Наприклад:

```
alert( 2 > 1 ); // true (верно)
alert( 2 == 1 ); // false (неверно)
alert( 2 != 1 ); // true (верно)
```

Результат порівняння можна присвоїти змінній, як і будь-яке значення:

```
let result = 5 > 4; // результат сравнения присваивается
переменной result
alert( result ); // true
```

### 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( 'Коты' > 'Кода' ); // true
alert( 'Сонный' > 'Сон' ); // 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.

У прикладах вище порівняння 'Я' > 'А' завершиться на першому кроці, тоді як рядки 'Коти' і 'Коду' будуть порівнюватися посимвольно:

1. К дорівнює К.
2. о дорівнює о.
3. т більше, ніж д. На цьому порівняння закінчується. Перший рядок більше.

### Порівняння різних типів

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

Наприклад:

```
alert( '2' > 1 ); // true, строка '2' становиться числом 2
alert( '01' == 1 ); // true, строка '01' становиться
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) `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, '?'`

Іноді нам потрібно виконати різні дії в залежності від умов.

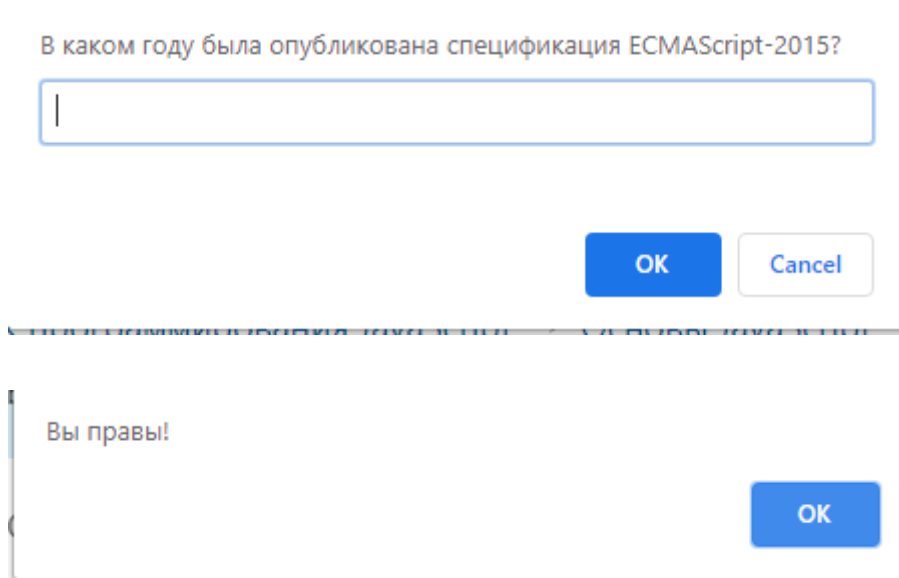
Для цього ми можемо використовувати інструкцію `if` і умовний оператор?, Який також називають оператором «знак питання».

#### Інструкція «`if`»

Інструкція `if (...)` обчислює умову в дужках і, якщо результат `true`, то виконує блок коду.

Наприклад:

```
let year = prompt('В каком году была опубликована спецификация  
ECMAScript-2015?', '');  
if (year == 2015) alert( 'Вы правы!' );
```



В наведеному вище прикладі, умова - це проста перевірка на рівність (`year == 2015`), але воно може бути і набагато більш складним.

Якщо ми хочемо виконати більше однієї інструкції, то потрібно укласти блок коду в фігурні дужки:

```
if (year == 2015) {  
    alert( "Правильно!" );  
    alert( "Вы такой умный!" );  
}
```

Рекомендується використовувати фігурні дужки `{}` завжди, коли ви використовуєте інструкцію `if`, навіть якщо виконується тільки одна команда. Це покращує читабельність коду.

### Блок «else»

Інструкція `if` може містити необов'язковий блок «else» ( «інакше»). Він виконується, коли умова помилкова.

Наприклад:

```
let year = prompt('В каком году была опубликована спецификация  
ECMAScript-2015?', '');  
if (year == 2015) {  
    alert( 'Да вы знаток!' );  
} else {  
    alert( 'А вот и неправильно!' ); // любое значение, кроме 2015  
}
```

Кілька умов: «else if»

Іноді, потрібно перевірити кілька варіантів умови. Для цього використовується блок else if.

Наприклад:

```
let year = prompt('В каком году была опубликована спецификация  
ECMAScript-2015?', '');  
if (year < 2015) {  
    alert( 'Это слишком рано...' );  
} else if (year > 2015) {  
    alert( 'Это поздновато' );  
} else {  
    alert( 'Верно!' );  
}
```

У наведеному вище коді JavaScript спочатку перевірить year < 2015. Якщо це не так, він переходить до наступної умови year > 2015. Якщо вона теж помилкова, тоді спрацює останній alert.

Блоків else if може бути і більше. Присутність блоку else не є обов'язковим.

Умовний оператор "?"

Іноді нам потрібно визначити змінну в залежності від умови.

Наприклад:

```
let accessAllowed;  
let age = prompt('Скільки вам лет?', '');  
  
if (age > 18) {  
    accessAllowed = true;  
} else {  
    accessAllowed = false;  
}
```

Так званий «умовний» оператор «знак питання» дозволяє нам зробити це більш коротким і простим способом.

Оператор представлений знаком питання?. Його також називають «тернарний», так як цей оператор, єдиний в своєму роді, має три аргументи.

Синтаксис:

```
let result = умова ? значення1 : значення2;
```

Спочатку обчислюється умова: якщо вона істинна, тоді повертається значення1, в іншому випадку - значення2.

Наприклад:

```
let accessAllowed = (age > 18) ? true : false;
```

## **Логічні оператори**

В JavaScript є три логічних оператора: || (АБО), && (І) і ! (НЕ).

Незважаючи на свою назву, дані оператори можуть застосовуватися до значень будь-яких типів. Отримані результати також можуть мати різний тип.

### **|| (АБО)**

Оператор «АБО» виглядає як подвійний символ вертикальної риси:

```
result = a || b;
```

Традиційно в програмуванні АБО призначене тільки для маніпулювання булевими значеннями: в разі, якщо який-небудь з аргументів true, він поверне true, в протилежній ситуації повертається false.

Існує всього чотири можливі логічні комбінації:

```
alert( true || true ); // true
alert( false || true ); // true
alert( true || false ); // true
alert( false || false ); // false
```

Як ми можемо спостерігати, результат операцій завжди дорівнює true, за винятком випадку, коли обидва аргументи false.

Якщо значення не логічного типу, то воно до нього приводиться в цілях обчислень.

При виконанні АБО || з декількома значеннями:

```
result = value1 || value2 || value3;
```

Оператор || виконує наступні дії:

1. Обчислює операнди зліва направо.
2. Кожен операнд конвертує в логічне значення. Якщо результат true, зупиняється і повертає початкове значення цього операнда.
3. Якщо всі операнди є помилковими (false), повертає останній з них.

Значення повертається в початковому вигляді, без перетворення.

Іншими словами, ланцюжок АБО "||" повертає перше істинне значення або останнє, якщо таке значення не знайдено.

```
alert( 1 || 0 ); // 1
```

```
alert( true || 'no matter what' ); // true
alert( null || 1 ); // 1 (первое истинное значение)
alert( null || 0 || 1 ); // 1 (первое истинное значение)
alert( undefined || null || 0 ); // 0 (поскольку все ложно, возвращается последнее значение)
```

## && (I)

Оператор I пишется как два амперсанда &&:

```
result = a && b;
```

У традиційному програмуванні I повертає true, якщо обидва аргументи істинні, а інакше - false:

```
alert( true && true ); // true
alert( false && true ); // false
alert( true && false ); // false
alert( false && false ); // false
```

При декількох підряд операторах I:

```
result = value1 && value2 && value3;
```

Оператор && виконує наступні дії:

1. Обчислює операнди зліва направо.
2. Кожен операнд перетворює в логічне значення. Якщо результат false, зупиняється і повертає початкове значення цього операнда.
3. Якщо всі операнди були істинними, повертається останній.

Іншими словами, I повертає перше помилкове значення. Або останнє, якщо нічого не знайдено.

```
// Если первый операнд истинный,
// И возвращает второй:
alert( 1 && 0 ); // 0
alert( 1 && 5 ); // 5
// Если первый операнд ложный,
// И возвращает его. Второй операнд игнорируется
```



```
alert( null && 5 ); // null
alert( 0 && "no matter what" ); // 0
```

## ! (НЕ)

Оператор НЕ представлений знаком оклику!.

Синтаксис:

```
result = !value;
```

Оператор приймає один аргумент і виконує наступні дії:

1. Спочатку приводить аргумент до логічного типу true / false.
2. Потім повертає протилежне значення.

Наприклад:

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

## Цикли while і for

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
    // также называемый "телом цикла"
}
```

Код з тіла циклу виконується, поки умова condition істинна.

Наприклад, цикл нижче виводить i, поки i < 3:

```
let i = 0;
while (i < 3) { // виводит 0, затем 1, затем 2
  alert( i );
  i++;
}
```

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) { // когда i будет равно 0, условие станет ложным, и
цикл остановится
  alert( i );
  i--;
}
```

### Цикл «do ... while»

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

```
do {
  // тело 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 виконалося 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 (начало; условие; шаг) {  
    // ... тело цикла ...  
}
```

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

```
for (let i = 0; i < 3; i++) { // виведет 0, затем 1, затем 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)

// Выполнить начало
let i = 0;
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// ...конец, потому что теперь 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); // ошибка, нет такой переменной
```

Замість оголошення нової змінної ми можемо використовувати вже існуючу:

```
let i = 0;

for (i = 0; i < 3; i++) { // используем существующую переменную
  alert(i); // 0, 1, 2
}
```

```
alert(i); // 3, переменная доступна, т.к. была объявлена снаружи  
цикла
```

### 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("Введіть число", '');  
  if (!value) break; // (*)  
  sum += value;  
}  
alert( 'Сума: ' + sum );
```

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++) {  
  // если true, пропустити оставшуюся часть тела 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( 'Маловато' );
    break;
  case 4:
    alert( 'В точку!' );
    break;
  case 5:
    alert( 'Перебор' );
    break;
  default:
    alert( "Нет таких значений" );
}

```

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("Введите число?");
switch (arg) {
  case '0':
  case '1':
    alert( 'Один или ноль' );
    break;
}

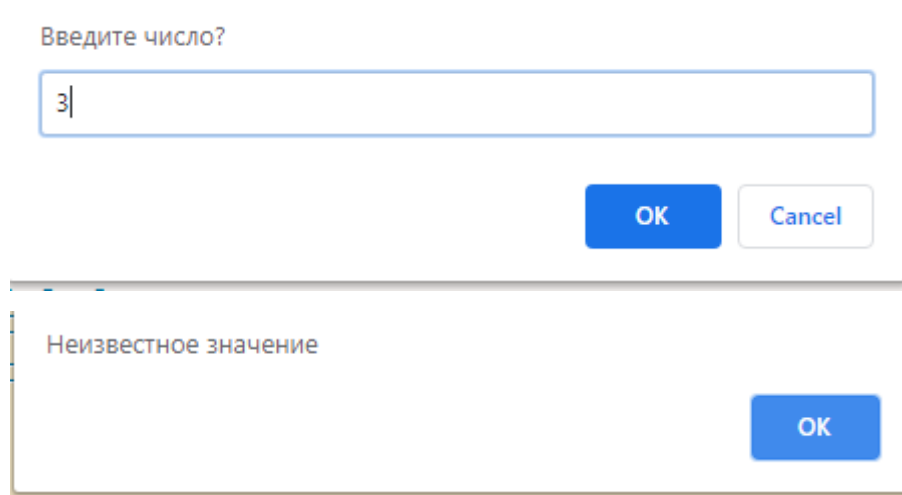
```

```
case '2':  
    alert( 'Два' );  
    break;  
  
case 3:  
    alert( 'Никогда не выполнится!' );  
    break;  
default:  
    alert( 'Неизвестное значение' );  
}
```

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.



Введите число?

OK Cancel

Неизвестное значение

OK



## Лекція 12-13

### Функції [10]

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.

Приклади вбудованих функцій ви вже бачили - це alert (message), prompt (message, default) і confirm (question). Але можна створювати і свої.

### Оголошення функції

Для створення функцій ми можемо використовувати оголошення функції.

Приклад оголошення функції:

```
function showMessage() {  
    alert( 'Всем привет!' );  
}
```

Спочатку йде ключове слово function, після нього ім'я функції, потім список параметрів в круглих дужках через кому (у вищенаведеному прикладі він порожній) і, нарешті, код функції, також званий «тілом функції», всередині фігурних дужок.

```
function имя(параметры) {  
    ...тело...  
}
```

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

Наприклад:

```
function showMessage() {  
    alert( 'Всем привет!' );  
}
```

```
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 = "Привет, я JavaScript!"; // локальная переменная  
  
    alert( message );  
}
```

```
showMessage(); // Привет, я JavaScript!
```

```
alert( message ); // <-- будет ошибка, т.к. переменная видна только  
внутри функции
```

### Зовнішні змінні

У функції є доступ до зовнішніх змінних, наприклад:

```
let userName = 'Вася';
```

```
function showMessage() {  
    let message = 'Привет, ' + userName;  
    alert(message);  
}  
showMessage(); // Привет, Вася
```

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) { // аргументы: from, text
    alert(from + ': ' + text);
}
showMessage('Аня', 'Привет!'); // Аня: Привет! (*)
showMessage('Аня', "Как дела?"); // Аня: Как дела? (**)
```

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 + '*'; // немного украсим "from"  
    alert( from + ': ' + text );  
}  
  
let from = "Аня";  
  
showMessage(from, "Привет"); // *Аня*: Привет  
  
// значение "from" осталось прежним, функция изменила значение  
локальной переменной  
alert( from ); // Аня
```

### 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("Аня");
```

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 добавлен") {  
    alert( from + ": " + text );  
}  
  
showMessage("Аня"); // Аня: текст не добавлен
```

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() викониться тільки якщо не передан 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
```

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 showMovie(age) {  
  if ( !checkAge(age) ) {  
    return;  
  }  
  
  alert( "Вам показується кино" ); // (*)  
  // ...  
}
```

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`.

Таким чином:

Оголошення функції має вигляд:

```
function имя(параметры, через, запятую) {  
  /* тело, код функции */  
}
```

Передані значення копіюються в параметри функції і стають локальними змінними.

Функції мають доступ до зовнішніх змінних. Але це працює тільки зсередини назовні. Код поза функцією не має доступу до її локальних змінних.

Функція може повертати значення. Якщо цього не відбувається, тоді результат дорівнює `undefined`.

Для того, щоб зробити код більш чистим і зрозумілим, рекомендується використовувати локальні змінні і параметри функцій, не користуватися зовнішніми змінними.

Функція, яка отримує параметри, працює з ними і потім повертає результат, набагато зрозуміліше функції, що викликається без параметрів, але змінює зовнішні змінні, що загрожує побічними ефектами.

Іменування функцій:

Ім'я функції має зрозуміло і чітко відображати, що вона робить. Побачивши її виклик в коді, ви повинні тут же розуміти, що вона робить, і що повертає.

Функція - це дія, тому її ім'я зазвичай є дієсловом.

Є багато загальноприйнятих префіксів, таких як: create ..., show ..., get ..., check ... і т.д. Користуйтеся ними як підказками, що пояснюють, що робить функція.

### **Function Expression**

Синтаксис, який ми використовували до цього, називається Function Declaration (Оголошення Функції):

```
function sayHi() {  
    alert( "Привет" );  
}
```

Існує ще один синтаксис створення функцій, який називається Function Expression (Функціональний Вираз).

Він виглядає ось так:

```
let sayHi = function() {  
    alert( "Привет" );  
};
```

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.

Сенс обох прикладів коду однаковий: "створити функцію і помістити її значення в змінну sayHi".

Ми можемо навіть вивести це значення за допомогою alert:

```
function sayHi() {  
  alert( "Привет" );  
}  
alert( sayHi ); // выведет код функции
```

Зверніть увагу, що останній рядок не викликає функцію sayHi, після її імені немає круглих дужок.

В JavaScript функції - це значення, тому ми і звертаємося з ними, як зі значеннями. Код вище виведе строкове представлення функції, яке є її вихідним кодом.



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) создаём  
  alert( "Привет" );  
}  
  
let func = sayHi; // (2) копируем  
  
func(); // Привет // (3) вызываем копию (работает)!  
sayHi(); // Привет // прежняя тоже работает (почему бы нет)
```

Давайте детально розберемо все, що тут сталося:

1. Оголошення Function Declaration (1) створило функцію і присвоїло її значення змінній з ім'ям sayHi.



2. У рядку (2) ми скопіювали її значення в змінну func. Зверніть увагу (ще раз): немає круглих дужок після sayHi. Якби вони були, то вираз func = sayHi () записав би результат виклику sayHi () в змінну func, а не саму функцію sayHi.

3. Тепер функція може бути викликана за допомогою обох змінних sayHi () і func ().



Зауважимо, що ми могли б використати і Function Expression для того, щоб створити sayHi в першому рядку:

```
let sayHi = function() {  
  alert( "Привет" );  
};  
  
let func = sayHi;  
// ...
```

Результат такий же.

### 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( "Вы согласны." );  
}
```

```
function showCancel() {  
  alert( "Вы отменили выполнение." );  
}
```

```
// использование: функции showOk, showCancel передаются в  
качестве аргументов ask  
ask("Вы согласны?", 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 браузерях 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(
  "Вы согласны?",
  function() { alert("Вы согласились."); },
  function() { alert("Вы отменили выполнение."); }
);
```

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.

Подібний код, в скрипті виглядає дуже природно, в дусі JavaScript.

### Function Expression в порівнянні з Function Declaration

Давайте розберемо ключові відмінності Function Declaration від Function Expression.

*По-перше, синтаксис: як визначити, що є що в коді.*

Function Declaration: функція оголошується окремою конструкцією «function ...» в основному потоці коду.

```
// Function Declaration
function sum(a, b) {
  return a + b;
}
```

Function Expression: функція, створена всередині іншого виразу або синтаксичної конструкції. В даному випадку функція створюється в правій частині «виразу присвоєння» =:

```
// Function Expression
let sum = function(a, b) {
  return a + b;
};
```

*Відмінність полягає, в тому, коли створюється функція движком JavaScript.*

Function Expression створюється, коли виконання доходить до нього, і потім вже може використовуватися.

Після того, як потік виконання досягне правої частини виразу присвоєння `let sum = function ...` - з цього моменту, функція вважається створеною і може бути використана (присвоєна змінній, викликана і т.д.).

З Function Declaration все інакше.

Function Declaration можна використовувати у всьому скрипті (або блоці коду, якщо функція оголошена в блоці).

Іншими словами, коли движок JavaScript готується виконувати скрипт або блок коду, перш за все він шукає в ньому Function Declaration і створює всі такі функції. Можна вважати цей процес «стадією ініціалізації».

І тільки після того, як всі оголошення Function Declaration будуть оброблені, продовжиться виконання.

В результаті, функції, створені, як Function Declaration, можуть бути викликані раніше своїх визначень.

Наприклад, так буде працювати:

```
sayHi("Вася"); // Привет, Вася

function sayHi(name) {
  alert( `Привет, ${name}` );
}
```



Функція `sayHi` була створена, коли движок JavaScript готував скрипт до виконання, і така функція видна всюди в цьому скрипті.

... Якби це було Function Expression, то такий код викличе помилку:

```
sayHi("Вася"); // ошибка!
```

```
let sayHi = function(name) { // (*) магии больше нет  
  alert( `Привет, ${name}` );  
};
```

Функції, оголошені за допомогою Function Expression, створюються тоді, коли виконання доходить до них. Це трапиться тільки на рядку з позначкою зірочкою (\*). Занадто пізно.

*Ще одна важлива особливість Function Declaration полягає в їх блоковій області видимості.*

У строгому режимі, коли Function Declaration знаходиться в блоці {...}, функція доступна всюди всередині блоку. Але не зовні нього.

У більшості випадків, коли нам потрібно створити функцію, переважно використовувати Function Declaration, тому що функція буде видима до свого оголошення в коді. Це дозволяє більш гнучко організовувати код, і покращує його читаність.

Таким чином, ми повинні вдаватися до оголошення функцій за допомогою Function Expression в разі, коли синтаксис Function Declaration не підходить для нашої задачі.

### **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;  
/* Более короткая форма для:  
let sum = function(a, b) {  
    return a + b;  
};  
*/  
alert( sum(1, 2) ); // 3
```

Тобто,  $(a, b) \Rightarrow a + b$  задає функцію з двома аргументами  $a$  і  $b$ , яка при запуску обчислює вираз праворуч  $a + b$  і повертає його результат.

Якщо у нас тільки один аргумент, то круглі дужки навколо параметрів можна опустити.

```
// с одним аргументом  
let double = n => n * 2;
```

Якщо немає аргументів, вказуються порожні круглі дужки:

```
// без аргументов  
let sayHi = () => alert("Привет");
```

Якщо нам потрібно щось складніше, наприклад, виконати кілька інструкцій. Це також можливо, потрібно лише включити інструкції в фігурні дужки. І використовувати `return` всередині них, як у звичайній функції.

```
// многострочный код в фигурных скобках { ... }, здесь нужен  
return:  
let sum = (a, b) => {  
    // ...  
    return a + b;  
}
```

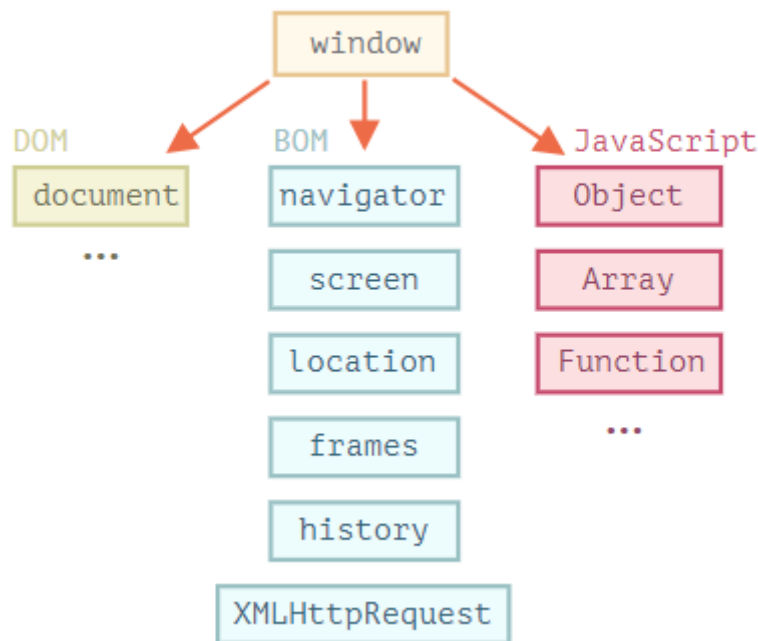


## Лекція 14

### Браузер: документ [10]

Розберемо 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");  
}  
// глобальные функции доступны как методы глобального объекта:  
window.sayHi();
```

А тут ми використовуємо `window` як об'єкт вікна браузера, щоб дізнатися його висоту:



```
alert(window.innerHeight); // внутренняя высота 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.

```
// заменим цвет фона на красный,  
document.body.style.background = "red";  
  
// а через секунду вернём как было  
setTimeout(() => document.body.style.background = "", 1000);
```

Ми використовували в прикладі тільки document.body.style, але насправді можливості по управлінню сторінкою набагато ширше. Різні властивості і методи описані в специфікації: DOM Living Standard на <https://dom.spec.whatwg.org>

## DOM - не тільки для браузерів

Специфікація DOM описує структуру документа і надає об'єкти для маніпуляцій зі сторінкою. Існують і інші, відмінні від браузерів, інструменти, що використовують DOM.

## CSSOM для стилів

Правила стилів CSS структуровані інакше ніж HTML. Для них є окрема специфікація CSSOM, яка пояснює, як стилі повинні представлятися у вигляді об'єктів, як їх читати і писати.

CSSOM використовується разом з DOM при зміні стилів документа. В реальності CSSOM потрібно рідко, зазвичай правила CSS статичні. Ми рідко додаємо / видаляємо стилі з JavaScript, але і це можливо.

## 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); // показываает текущий URL
if (confirm("Перейти на Wikipedia?")) {
    location.href = "https://wikipedia.org"; // перенаправляет
браузер на другой URL
}
```

Функції alert / confirm / prompt теж є частиною BOM: вони не відносяться безпосередньо до сторінки, але є методи об'єкта вікна браузера для комунікації з користувачем.

BOM є частиною загальної специфікації HTML. Специфікація HTML за адресою <https://html.spec.whatwg.org> не тільки про «мову HTML» (теги, атрибути), вона також покриває безліч об'єктів, методів і специфічних для кожного браузера розширень DOM. Для деяких речей є окремі специфікації, перераховані на <https://spec.whatwg.org>.

## DOM-дерево

Основою HTML-документа є теги.

Відповідно до об'єктної моделі документа ( «Document Object Model», коротко DOM), кожен HTML-тег є об'єктом. Вкладені теги є «дітьми» батьківського елемента. Текст, який знаходиться всередині тега, також є об'єктом.

Всі ці об'єкти доступні за допомогою JavaScript, ми можемо використовувати їх для зміни сторінки.

Наприклад, **document.body** - об'єкт для тега <body>.

Якщо запустити цей код, то <body> стане червоним на 3 секунди:

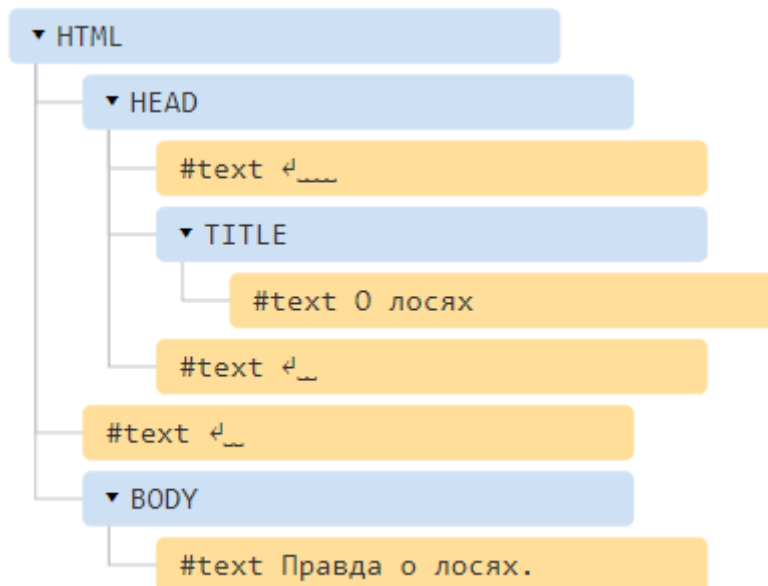
```
document.body.style.background = 'red'; // сделать фон красным
setTimeout(() => document.body.style.background = '', 3000); //
```

вернуть назад

### Приклад DOM

```
<!DOCTYPE HTML>
<html>
<head>
  <title>0 лосях</title>
</head>
<body>
  Правда о лосях.
</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.

Наприклад, в тезі <title> є текстовий вузол "Про лосях".

Зверніть увагу на спеціальні символи в текстових вузлах:

перенесення рядка:  $\backslash n$  (в JavaScript він позначається як `\n`)

пробіл:

Пробіли і перенесення рядка - це повноправні символи, як букви і цифри. Вони утворюють текстові вузли і стають частиною дерева DOM. Так, в прикладі вище в тезі `<head>` є кілька пробілів перед `<title>`, які утворюють текстовий вузол `#text` (він містить в собі тільки перенесення рядка і кілька пробілів).

Існує всього два винятки з цього правила:

- з історичних причин пробіли і перенесення рядка перед тегом `<head>` ігноруються
- якщо ми записуємо що-небудь після тега `</body>`, браузер автоматично переміщує цей запис в кінець `body`, оскільки специфікація HTML вимагає, щоб весь вміст був всередині `<body>`. Тому після закриваючого тега `</ body>` не може бути ніяких пробілів.

### Авто виправлення

Якщо браузер стикається з некоректно написаним HTML-кодом, він автоматично коригує його при побудові DOM.

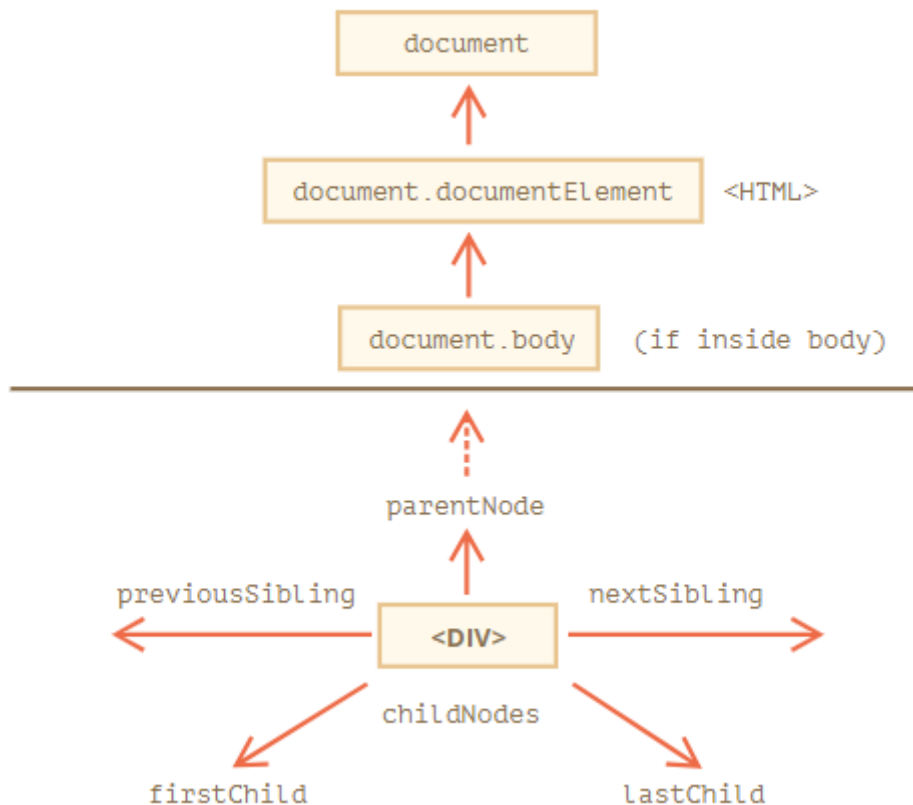
Наприклад, на початку документа завжди повинен бути тег `<html>`. Навіть якщо його немає в документі - він буде в дереві DOM, браузер його створить. Те ж саме стосується і тега `<body>`.

При генерації DOM браузер самостійно обробляє помилки в документі, закриває теги і так далі.

### 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:



Поговоримо про це докладніше.

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 елементу, 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( "Из HEAD: " + document.body ); // null, <body> ещё нет
  </script>
</head>
<body>
  <script>
    alert( "Из BODY: " + document.body ); // HTMLBodyElement,
теперь он есть
  </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>Начало</div>
  <ul>
    <li>Информация</li>
  </ul>
  <div>Конец</div>
  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
```

```

        alert( document.body.childNodes[i] ); // Text, DIV, Text,
        UL, ..., SCRIPT
    }
</script>
...какой-то HTML-код...
</body>
</html>

```

Звернемо увагу на маленьку деталь. Якщо запустити приклад вище, то останнім буде виведений елемент `<script>`. Насправді, в документі є ще «якийсь HTML-код», але на момент виконання скрипта браузер ще до нього не дійшов, тому скрипт не бачить його.

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); // покажет все узлы из коллекции
}

```

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
коллекции нет 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 ); //
сделали массив
```

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.

*Не використовуйте цикл `for..in` для перебору колекцій*

Колекції перебираються циклом `for..of`. Деякі початківці розробники намагаються використовувати для цього цикл `for..in`.

Не робіть так. Цикл `for..in` перебирає всі перераховані властивості. А у колекцій є деякі «зайві», рідко використовуються властивості, які зазвичай нам не потрібні:

```
<body>
<script>
  // выводит 0, 1, length, item, values и другие свойства.
  for (let prop in document.body.childNodes) alert(prop);
</script>
</body>
```

### Сусіди і батько

Сусіди - це вузли, у яких один і той же батько.

Наприклад, тут `<head>` і `<body>` сусіди:

```
<html>
  <head>...</head><body>...</body>
</html>
```

<body> - «наступний» або «правий» сусід <head> також можна сказати, що <head> «попередній» або «лівий» сусід <body>.

Наступний вузол того ж батька (наступний сусід) - у властивості **nextSibling**, а попередній - в **previousSibling**. Батько доступний через **parentNode**.

Наприклад:

```
// батьком <body> є <html>
```

```
alert (document.body.parentNode === document.documentElement); //
виведе true
```

```
// родителем <body> является <html>
alert( document.body.parentNode === document.documentElement );
// выведет true
```

```
// после <head> идёт <body>
alert( document.head.nextSibling ); // HTMLBodyElement
```

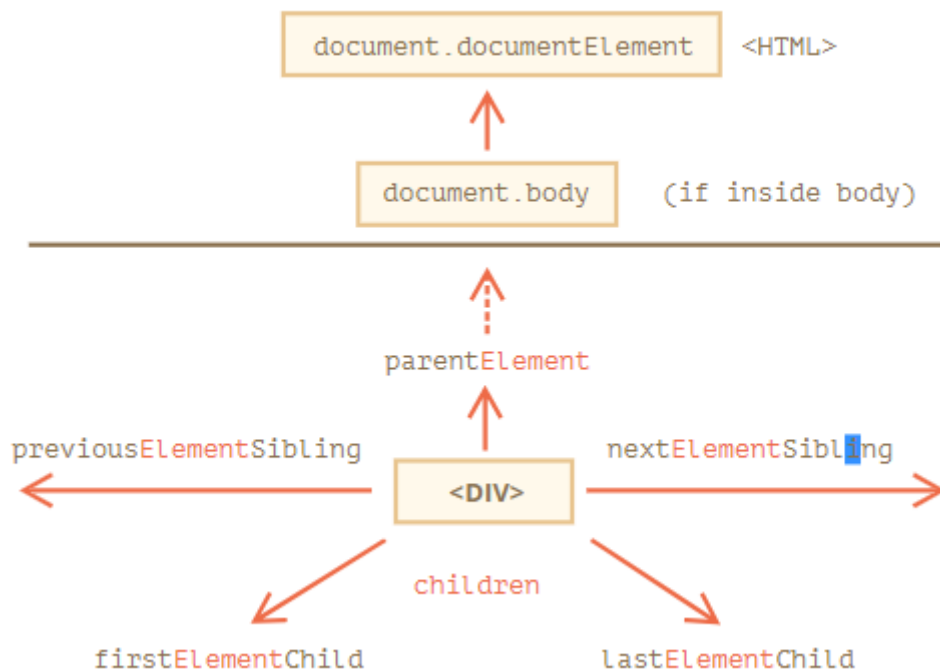
```
// перед <body> находится <head>
alert( document.body.previousSibling ); // HTMLHeadElement
```

### 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 ); // виведет document
alert( document.documentElement.parentElement ); // виведет 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>Начало</div>

  <ul>
    <li>Информация</li>
  </ul>

  <div>Конец</div>

  <script>
    for (let elem of document.body.children) {
      alert(elem); // DIV, UL, DIV, SCRIPT
    }
  </script>
  ...
</body>
</html>
```

Деякі типи DOM-елементів надають для зручності додаткові властивості, специфічні для їх типу.

Таблиці - відмінний приклад таких елементів.

Елемент **<table>**, на додаток до властивостей, про які йшлося вище, підтримує наступні:

- **table.rows** - колекція рядків **<tr>** таблиці.
- **table.caption/tHead/tFoot** - посилання на елементи таблиці

**<caption>**, **<thead>**, **<tfoot>**.

- **table.tBodies** - колекція елементів таблиці **<tbody>** (за специфікацією їх може бути більше одного).

**<thead>**, **<tfoot>**, **<tbody>** надають властивість **rows**:

- **tbody.rows** - колекція рядків **<tr>** секції.

**<tr>**:

- **tr.cells** - колекція **<td>** і **<th>** комірок, що знаходяться всередині рядка **<tr>**.

– **tr.sectionRowIndex** - номер рядка **<tr>** в поточній секції **<thead>** / **<tbody>** / **<tfoot>**.

– **tr.rowIndex** - номер рядка **<tr>** в таблиці (включаючи всі рядки таблиці).

**<td> and <th>:**

– **td.cellIndex** - номер комірки в рядку **<tr>**.

Приклад використання:

```
<table id="table">
  <tr>
    <td>один</td><td>два</td>
  </tr>
  <tr>
    <td>три</td><td>четыре</td>
  </tr>
</table>

<script>
  // выводит содержимое первой строки, второй ячейки
  alert( table.rows[0].cells[1].innerHTML ) // "два"
</script>
```

### 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 **id** він не знаходився.

Наприклад:

```
div id="elem">
  <div id="elem-content">Element</div>
</div>
<script>
  // получить элемент
  let elem = document.getElementById('elem');
  // сделать его фон красным
  elem.style.background = 'red';
```

```
</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>Этот</li>
  <li>тест</li>
</ul>
<ul>
  <li>полностью</li>
  <li>пройден</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');
  for (let elem of elements) {
    alert(elem.innerHTML); // "тест", "пройден"
  }
</script>
```

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/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  // может быть любая коллекция вместо document.body.children
  for (let elem of document.body.children) {
    if (elem.matches('a[href$="zip"]')) {
      alert("Ссылка на архив: " + elem.href );
    }
  }
</script>
```

## closest

Предки елемента - батько, батько батька, його батько і так далі. Разом вони утворюють ланцюжок ієрархії від елемента до вершини.

Метод **elem.closest(css)** шукає найближчого предка, який відповідає CSS-селектору. Сам елемент також включається в пошук.

Іншими словами, метод `closest` піднімається вгору від елемента і перевіряє кожного з батьків. Якщо він відповідає селектору, пошук припиняється. Метод повертає або предка, або `null`, якщо такий елемент не знайдений.

Наприклад:

```
<h1>Содержание</h1>
<div class="contents">
  <ul class="book">
    <li class="chapter">Глава 1</li>
    <li class="chapter">Глава 2</li>
  </ul>
</div>
<script>
  let chapter = document.querySelector('.chapter'); // LI
  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV
  alert(chapter.closest('h1')); // null (потому что h1 - не предок)
</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.getElementsByName(name)** повертає елементи з заданим атрибутом `name`.

## Живі колекції



Всі методи "getElementsBy\*" повертають живу колекцію. Такі колекції завжди відображають поточний стан документа і автоматично оновлюються при його зміні.

У наведеному нижче прикладі є два скрипта.

Перший створює посилання на колекцію <div>. На цей момент її довжина дорівнює 1.

Другий скрипт запускається після того, як браузер зустрічає ще один <div>, тепер її довжина - 2.

```
div>First div</div>

<script>
  let divs = document.getElementsByTagName('div');
  alert(divs.length); // 1
</script>
<div>Second div</div>
<script>
  alert(divs.length); // 2
</script>
```

Тепер ми легко бачимо різницю. Довжина статичної колекції не змінилася після появи нового div в документі.

Таким чином:

Метод	Шукає по ...	Шукає всередині елемента?	Повертає живу колекцію?
querySelector	CSS-selector	✓	-
querySelectorAll	CSS-selector	✓	-
getElementById	id	-	-
getElementsByName	name	-	✓
getElementsByTagName	tag or '*'	✓	✓
getElementsByClassName	class	✓	✓

Безумовно, найбільш часто використовуваними в даний час є методи `querySelector` і `querySelectorAll`, а методи `getElement(s)By*` можуть бути корисні в окремих випадках, а також зустрічаються в старому коді.

Властивості вузлів: тип, тег і вміст

Тепер давайте більш уважно поглянемо на DOM-вузли.

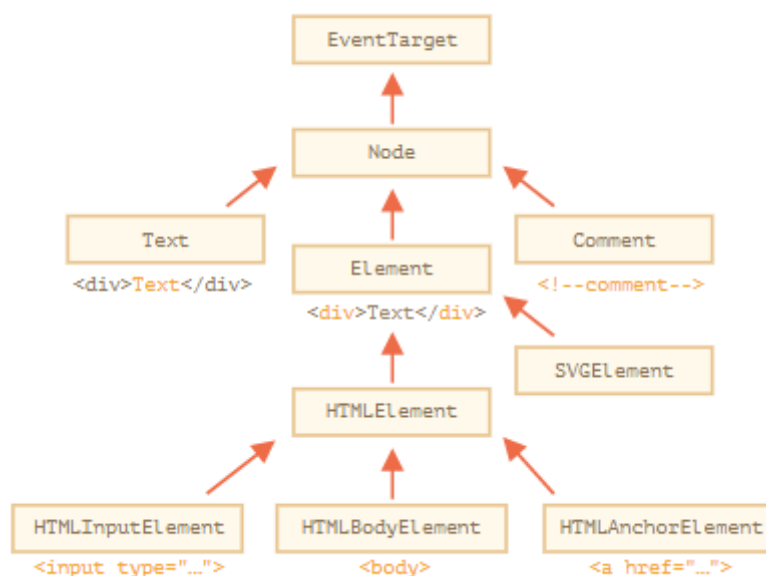
У цьому розділі ми докладніше розберемо, що вони собою представляють і вивчимо їх основні властивості.

### 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 успадковують 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 успадковує 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**. Воно 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
```

### Властивість «nodeType»

Властивість `nodeType` надає ще один, «старомодний» спосіб дізнатися «тип» DOM-вузла.

Його значенням є цифра:

`elem.nodeType == 1` для вузлів-елементів,

`elem.nodeType == 3` для текстових вузлів,

`elem.nodeType == 9` для об'єктів документа,

Наприклад:

```
<body>
  <script>
    let elem = document.body;

    // посмотрим что это?
    alert(elem.nodeType); // 1 => элемент

    // и первый потомок...
    alert(elem.firstChild.nodeType); // 3 => текст

    // для объекта document значение типа -- 9
    alert( document.nodeType ); // 9
  </script>
</body>
```

В сучасних скриптах, щоб дізнатися тип вузла, ми можемо використовувати метод `instanceof` і інші способи перевірити клас, але іноді `nodeType` простіше використовувати. Ми не можемо змінити значення `nodeType`, тільки прочитати його.

### 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.

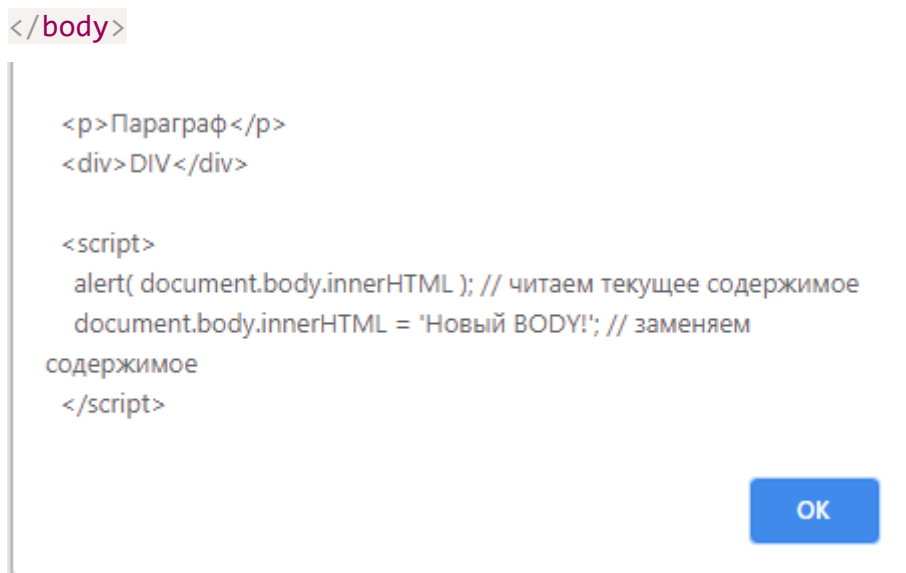
### **innerHTML**: вміст елемента

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 ); // читаем текущее содержимое
  document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое
</script>
```



Якщо innerHTML вставляє в документ тег `<script>` - він стає частиною HTML, але не запускається.

Будьте уважні: «**innerHTML** +=» здійснює перезапис

Ми можемо додати HTML до елементу, використовуючи **elem.innerHTML += "ще html"**.

Ось так:

```

chatDiv.innerHTML += "<div>Привет<img src='smile.gif'/> !</div>";
chatDiv.innerHTML += "Как дела?";

```

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">Привет <b>Мир</b></div>
<script>
  alert(elem.outerHTML); // <div id="elem">Привет <b>Мир</b></div>
</script>

```

```
<div id="elem">Привет <b>Мир</b></div>
```

OK

### **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 воно коротше.

Прочитаємо вміст текстового вузла і коментаря:

```
<body>
Привет
<!-- Коментарий -->
<script>
  let text = document.body.firstChild;
  alert(text.data); // Привет
  let comment = text.nextSibling;
  alert(comment.data); // Коментарий
</script>
</body>
```

Привет

OK

Коментарий

OK

### **textContent: просто текст**

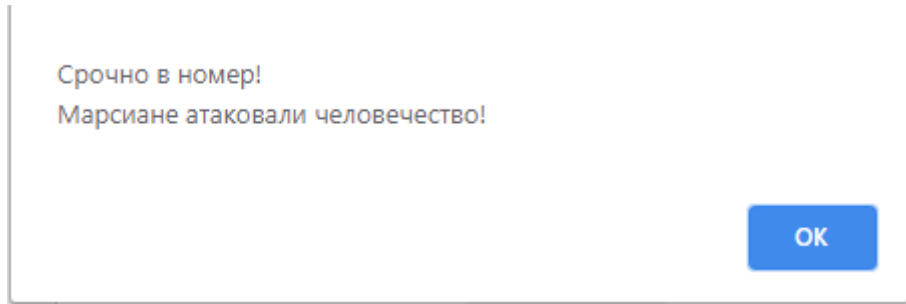
Властивість textContent надає доступ до тексту всередині елемента за вирахуванням всіх <тегів>.

Наприклад:

```

<div id="news">
  <h1>Срочно в номер!</h1>
  <p>Марсиане атаковали человечество!</p>
</div>
<script>
  // Срочно в номер! Марсиане атаковали человечество!
  alert(news.textContent);
</script>

```



Як ми бачимо, повертається тільки текст, як би 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("Введите ваше имя?", "<b>Винни-пух!</b>");

  elem1.innerHTML = name;
  elem2.textContent = name;
</script>

```



Введите ваше имя?

**<b>Винни-пух!</b>**

ОК Cancel

**Винни-пух!**

**<b>Винни-пух!</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>Оба тега DIV внизу невидимы</div>
<div hidden>С атрибутом "hidden"</div>
<div id="elem">С назначенным JavaScript свойством "hidden"</div>
<script>
  elem.hidden = true;
</script>
```

Оба тега DIV внизу невидимы

Технічно, hidden працює так само, як style = "display: none". Але його застосування простіше.

Мигающий элемент:

```
<div id="elem">Мигающий элемент</div>
<script>
  setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

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="значение">

<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // значение
</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: 'Caesar',  
  title: 'Imperator'  
};  
alert(document.body.myData.title); // Imperator
```

Ми можемо додати і метод:

```
document.body.sayTagName = function() {  
  alert(this.tagName);  
};  
  
document.body.sayTagName(); // BODY (значением "this" в этом методе  
будет 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 відповідна властивість. Але цього не відбувається, якщо атрибут нестандартний.

Наприклад:

```
<body id="test" something="non-standard">  
<script>
```

```
    alert(document.body.id); // test
    // нестандартный атрибут не преобразуется в свойство
    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 нестандартного властивості:

```
<body something="non-standard">
  <script>
    alert(document.body.getAttribute('something')); // non-
standard
  </script>
</body>
```

У HTML-атрибутів є такі особливості:

- їх імена реєстронезалежні (id те ж саме, що і ID).
- їх значення завжди є рядками.

Розширена демонстрація роботи з атрибутами:

```
<body>
  <div id="elem" about="Elephant"></div>

  <script>
    alert( elem.getAttribute('About') ); // (1) 'Elephant',
чтение

    elem.setAttribute('Test', 123); // (2), запись

    alert( elem.outerHTML ); // (3), поспотрим, есь ли атрибут в
HTML (да)

    for (let attr of elem.attributes) { // (4) весь список
      alert( `${attr.name} = ${attr.value}` );
    }
  </script>
</body>
```

Зверніть увагу:

1. `getAttribute ( 'About' )` - тут перша буква заголовна, а в HTML - рядкова. Але це не важливо: імена атрибутів реєстронезалежні.
2. Ми можемо присвоїти що завгодно атрибуту, але це стане рядком. Тому в цьому рядку ми отримуємо значення "123".
3. Всі атрибути, в тому числі ті, які ми встановили, видно в `outerHTML`.
4. Колекція `attributes` є перебирати. У ній є всі атрибути елемента (стандартні і нестандартні) у вигляді об'єктів з властивостями `name` і `value`.

### 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>
<script>
```

```

let input = document.querySelector('input');
// атрибут => свойство
input.setAttribute('id', 'id');
alert(input.id); // id (оновлено)
// свойство => атрибут
input.id = 'newId';
alert(input.getAttribute('id')); // newId (оновлено)
</script>

```

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');
// атрибут => значення
input.setAttribute('value', 'text');
alert(input.value); // text
// свойство => атрибут
input.value = 'newValue';
alert(input.getAttribute('value')); // text (не оновилося!)
</script>

```

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:
пустая строка
alert(input.checked); // значення свойства: true

```

### **Нестандартні атрибути, `dataset`**

При написанні HTML ми використовуємо багато стандартних атрибутів. Але що щодо нестандартних, призначених для користувача? Давайте подивимося, корисні вони чи ні, і для чого вони потрібні.

Іноді нестандартні атрибути використовуються для передачі призначених для користувача даних з HTML в JavaScript, або щоб «позначати» HTML-елементи для JavaScript.

Також вони можуть бути використані, щоб стилізувати елементи.

Наприклад, тут для стану замовлення використовується атрибут `order-state`:

```
<style>
  /* стили зависят от пользовательского атрибута "order-state" */
  .order[order-state="new"] {
    color: green;
  }

  .order[order-state="pending"] {
    color: blue;
  }

  .order[order-state="canceled"] {
    color: red;
  }
</style>

<div class="order" order-state="new">
  A new order.
</div>

<div class="order" order-state="pending">
  A pending order.
</div>

<div class="order" order-state="canceled">
  A canceled order.
</div>

A new order.
A pending order.
A canceled order.
```

Чому атрибут може бути краще таких класів, як `.order-state-new`, `.order-state-pending`, `order-state-canceled`?

Це тому, що атрибутом зручніше управляти. Стан може бути змінено досить просто:

```
// немного проще, чем удаление старого/добавление нового класса
div.setAttribute('order-state', 'canceled');
```

Але з користувацькими атрибутами можуть виникнути проблеми. Що якщо ми використовуємо нестандартний атрибут для наших цілей, а пізніше він з'явиться в стандарті і буде виконувати якусь функцію?

Щоб уникнути конфліктів, існують атрибути вигляду **data-\***.

Всі атрибути, що починаються із префікса **«data-»**, зарезервовані для використання програмістами. Вони доступні в властивості **dataset**.

Наприклад:

```
<body data-about="Elephants">
<script>
  alert(document.body.dataset.about); // Elephants
</script>
```

Атрибути, що складаються з декількох слів, наприклад **data-order-state**, стають властивостями, записаними за допомогою верблужої нотації: **dataset.orderState**.

Ось переписаний приклад «стану замовлення»:

```
<style>
  .order[data-order-state="new"] {
    color: green;
  }

  .order[data-order-state="pending"] {
    color: blue;
  }

  .order[data-order-state="canceled"] {
    color: red;
  }
</style>

<div id="order" class="order" data-order-state="new">
  A new order.
</div>

<script>
  // чтение
  alert(order.dataset.orderState); // new

  // изменение
```



```
order.dataset.orderState = "pending"; // (*)  
</script>
```

A new order.

Ми можемо не тільки читати, а й змінювати **data**-атрибути. Тоді CSS оновить подання відповідним чином: в прикладі вище останній рядок (\*) змінює колір на синій.

### 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:

```
<style>  
.alert {  
  padding: 15px;  
  border: 1px solid #d6e9c6;  
  border-radius: 4px;  
  color: #3c763d;  
  background-color: #dff0d8;  
}  
</style>
```

```
<div class="alert">  
  <strong>Всем привет!</strong> Вы прочитали важное сообщение.  
</div>
```

Всем привет! Вы прочитали важное сообщение.

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('А вот и я');
```

### 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>Всем привет!</strong> Вы прочитали важное сообщение.";
```

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: #3c763d;
  background-color: #dff0d8;
}
</style>
<script>
```

```
let div = document.createElement('div');
div.className = "alert";
div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали
важное сообщение.";
document.body.append(div);
</script>
```

before

1. prepend
2. 0
3. 1
4. 2
5. append

after

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 інший, 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 вставлений саме «як HTML».

Наприклад:

```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Привет</p>');
  div.insertAdjacentHTML('afterend', '<p>Пока</p>');
</script>
```

...Призведе до:

```
<p>Привет</p>
<div id="div"></div>
<p>Пока</p>
```

Привет

Пока

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 віддалялося через секунду:

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>
<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали
важное сообщение.";
  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: #3c763d;
  background-color: #dff0d8;
}
</style>
<div class="alert" id="div">
  <strong>Всем привет!</strong> Вы прочитали важное сообщение.
</div>
<script>
  let div2 = div.cloneNode(true); // клонировать сообщение
  div2.querySelector('strong').innerHTML = 'Всем пока!'; //
изменить клонированный элемент
  div.after(div2); // показать клонированный элемент после
существующего div
</script>

```

Всем привет! Вы прочитали важное сообщение.

Всем пока! Вы прочитали важное сообщение.

## 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>Где-то на странице...</p>
<script>
  document.write('<b>Привет из JS</b>');
</script>
<p>Конец</p>

```

Где-то на странице...

**Привет из JS**

Конец

Виклик `document.write (html)` записує `html` на сторінку «прямо тут і зараз». Рядок `html` може бути динамічно згенерований, тому метод досить гнучкий. Ми можемо використовувати JavaScript, щоб створити повноцінну веб-сторінку і записати її в документ.

В сучасних скриптах він рідко зустрічається через наступне обмеження: виклик `document.write` працює тільки під час завантаження сторінки. Якщо викликати його пізніше, то існуючий вміст документа зітреться.

Застарілі методи:

**`parent.appendChild (node)`**

**`parent.insertBefore (node, nextSibling)`**

**`parent.removeChild (node)`**

**`parent.replaceChild (newElem, node)`**

Всі ці методи повертають `node`.

## Лекція 15

### Події [10]

#### Введення в браузерні події

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.

#### Події документа:

**DOMContentLoaded** - коли HTML завантажений і оброблений, DOM документа повністю побудований і доступний.

#### CSS events:



**transitionend** - коли CSS-анімація завершена.

## Обробники подій

Події можна призначити обробник, тобто функцію, яка спрацює, як тільки подія відбулася.

Саме завдяки обробникам JavaScript-код може реагувати на дії користувача.

Є кілька способів призначити події обробник. 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="Нажми мене" onclick="alert('Клик!')" 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 `countRabbits ()`:

```
script>
function countRabbits() {
  for(let i=1; i<=3; i++) {
    alert("Кролик номер " + i);
  }
}
```

```

    }
  </script>
  <input type="button" onclick="countRabbits()" value="Считать кроликов!">

```

Считать кроликов!

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="Нажми меня!">
<script>
  elem.onclick = function() {
    alert('Спасибо');
  };
</script>

```

Нажми меня!

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('Клик!')"
value="Кнопка">

```

Кнопка

HTML + JS:

```

<input type="button" id="button" value="Кнопка">
<script>

```

```
button.onclick = function() {
    alert('Клик!');
};
</script>
```

Кнопка

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('Было')" value="Нажми
меня">
<script>
    elem.onclick = function() { // перезапишет существующий обработчик
        alert('Станет'); // выведется только это
    };
</script>
```

Нажми меня

Станет

OK

До речі, обробником можна призначити і вже існуючу функцію:

```
function sayThanks() {
    alert('Спасибо!');
}
elem.onclick = sayThanks;
```

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)">Нажми  
меня</button>
```

Якщо ви тільки починаєте працювати з подіями, зверніть увагу на наступні моменти.

Функція повинна бути присвоєна як `sayThanks`, а не `sayThanks ()`.

```
// правильно  
button.onclick = sayThanks;
```

```
// неправильно  
button.onclick = sayThanks();
```

Якщо додати дужки, то `sayThanks ()` - 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="sayThanks()">
```

Цю різницю просто пояснити. При створенні обробника браузером з атрибута, він автоматично створює функцію з тілом зі значення атрибута: `sayThanks ()`.

Так що розмітка генерує таку властивість:

```
button.onclick = function() {  
    sayThanks(); // содержимое атрибута  
};
```

Використовуйте саме функції, а не рядки.

Призначення обробника рядком `elem.onclick = "alert (1)"` також спрацює. Це зроблено з міркувань сумісності, але робити так не рекомендується.

Не використовуйте **setAttribute** для обробників.

Такий виклик працювати не буде:

```
// при нажатии на body будут ошибки,  
// атрибуты всегда строки, и функция станет строкой
```

```
document.body.setAttribute('onclick', function() { alert(1) });
```

Регістр DOM-властивості має значення.

Використовуйте `elem.onclick`, а не `elem.ONCLICK`, тому що DOM-властивості чутливі до регістру.

### **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); } // заменит предыдущий  
обработчик
```

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( 'Спасибо!' );
}
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="Нажми меня"/>
<script>
    function handler1() {
        alert('Спасибо!');
    };
    function handler2() {
        alert('Спасибо ещё раз!');
    }
    elem.onclick = () => alert("Привет");
    elem.addEventListener("click", handler1); // Спасибо!
    elem.addEventListener("click", handler2); // Спасибо ещё раз!
```

```
</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 побудовано"); // не буде працювати  
};  
document.addEventListener("DOMContentLoaded", function() {  
    alert("DOM побудовано"); // а ось так працює  
});
```

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="Натисни мене" id="elem">  
  
<script>  
    elem.onclick = function(event) {  
        // вивести тип події, елемент і координати кліка  
        alert(event.type + " на " + event.currentTarget);  
        alert("Координати: " + event.clientX + ":" + event.clientY);  
    };  
</script>
```

```
</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**

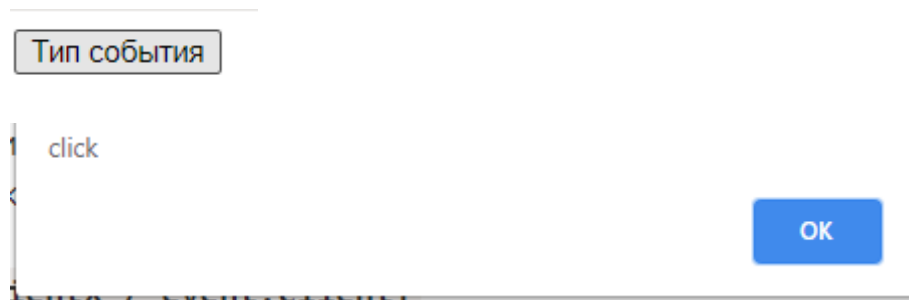
Координати курсора в момент кліка щодо вікна, для подій миші.

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="Тип события">
```



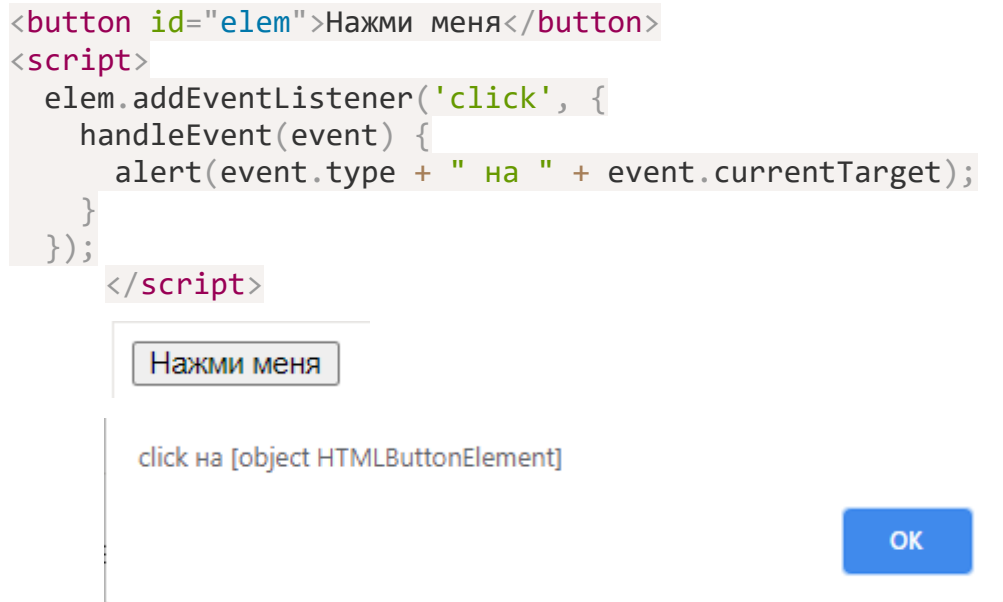
Це можливо тому, що коли браузер з атрибута створює функцію-обробник, то вона виглядає так: **function (event) {alert (event.type)}**. Тобто, її перший аргумент називається "event", а тіло взято з атрибута.

### Об'єкт-обробник: **handleEvent**



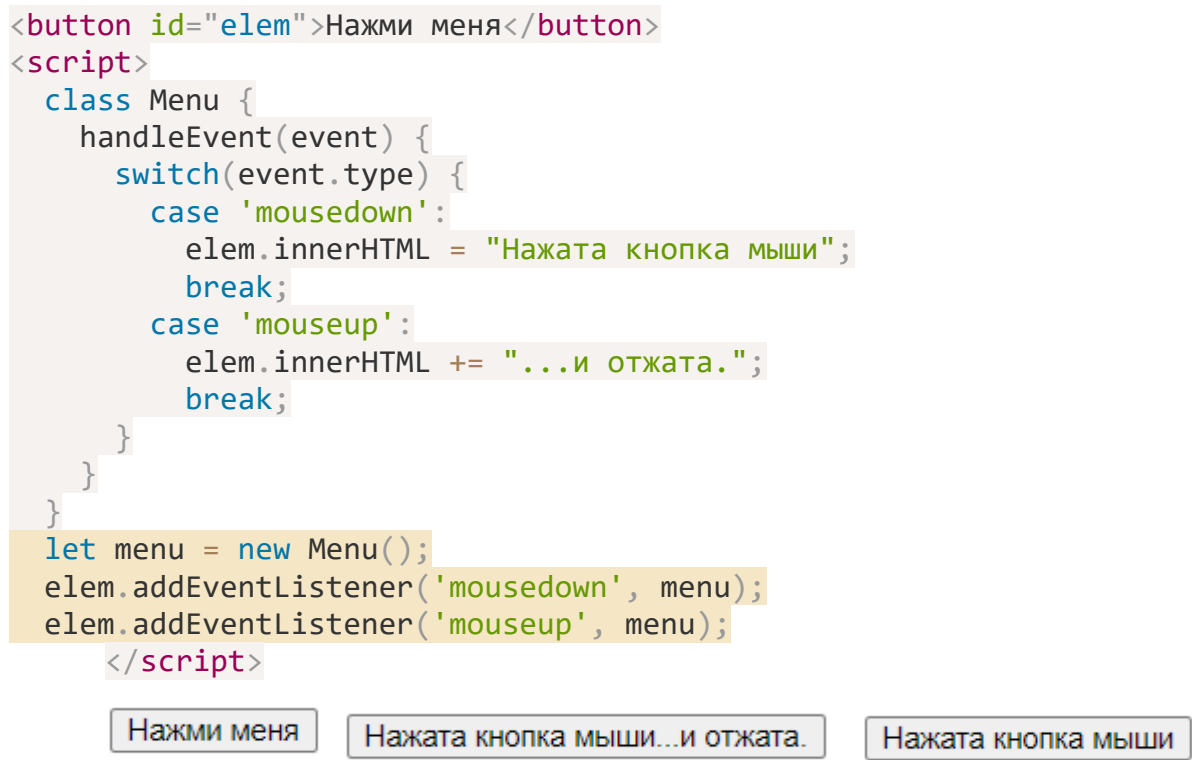
Ми можемо призначити обробником не тільки функцію, а й об'єкт за допомогою `addEventListener`. У цьому випадку, коли відбувається подія, викликається метод об'єкта `handleEvent`.

Наприклад:



Як бачимо, якщо `addEventListener` отримує об'єкт як обробника, він викликає `object.handleEvent(event)`, коли відбувається подія.

Ми також можемо використовувати клас для цього:



Тут один і той же об'єкт обробляє обидві події. Зверніть увагу, ми повинні явно призначити обидва обробники 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">Нажми меня</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 = "Кнопка мыши нажата";
    }
    onMouseup() {
      elem.innerHTML += "...и отжата.";
    }
  }
  let menu = new Menu();
  elem.addEventListener('mousedown', menu);
  elem.addEventListener('mouseup', menu);
</script>
```

Нажми меня

Нажата кнопка мыши...и отжата.

Нажата кнопка мыши

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('Обработчик!')">
  <em>Если вы кликните на <code>EM</code>, сработает обработчик на
  <code>DIV</code></em>
</div>
```

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>
```

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';

```

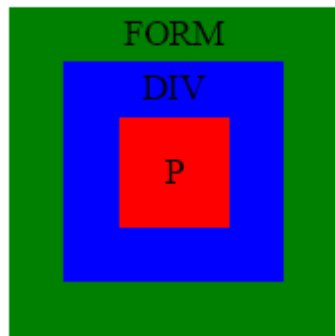
```

  // браузеру нужно некоторое время, чтобы зарисовать всё жёлтым
  setTimeout(() => {
    alert("target = " + event.target.tagName + ", this=" +
this.tagName);
    event.target.style.backgroundColor = ''
  }, 0);
};

```

Результат

Клік покажет оба: и `event.target`, и `this` для сравнения:



target = P, this=FORM

OK

target = DIV, this=FORM

OK

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(`сюда всплытие не дойдёт`)">  
  <button onclick="event.stopPropagation()">Клики меня</button>  
</body>
```

Клики меня

### 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})  
// или просто "true", как сокращение 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.

```
<table>
  <tr>
    <th colspan="3">Квадрат <em>Vagua</em>: Направление, Элемент,
Цвет, Значение</th>
  </tr>
  <tr>
    <td>...<strong>Северо-Запад</strong>...</td>
    <td>...</td>
    <td>...</td>
  </tr>
  <tr>...ещё 2 строки такого же вида...</tr>
  <tr>...ещё 2 строки такого же вида...</tr>
</table>
```

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; // где был клик?
  if (target.tagName !== 'TD') return; // не на TD? тогда не интересуется

  highlight(target); // подсветить TD
};

function highlight(td) {
  if (selectedTd) { // убрать существующую подсветку, если есть
    selectedTd.classList.remove('highlight');
  }
  selectedTd = td;
  selectedTd.classList.add('highlight'); // подсветить новый td
}
```

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>Северо-Запад</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 кожен 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">Нажмите, чтобы  
Сохранить</button>
```

Обробник зчитує вміст атрибута і виконує метод.

Приклад:

```
<div id="menu">  
  <button data-action="save">Сохранить</button>  
  <button data-action="load">Загрузить</button>  
  <button data-action="search">Поиск</button>  
</div>  
<script>  
  class Menu {  
    constructor(elem) {  
      this._elem = elem;  
      elem.onclick = this.onClick.bind(this); // (*)  
    }  
    save() {  
      alert('сохраняю');  
    }  
    load() {  
      alert('загружаю');  
    }  
    search() {  
      alert('ищу');  
    }  
    onClick(event) {  
      let action = event.target.dataset.action;  
      if (action) {
```

```

        this[action]();
    }
};
}
new Menu(menu);
</script>

```

Сохранить

Загрузить

Поиск

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, оробляє відповідну дію.

### Поведінка: «Лічильник»

Наприклад, тут HTML-атрибут `data-counter` додає кнопкам поведінку: «збільшити значення при кліці»:

```
Счётчик: <input type="button" value="1" data-counter>  
Ещё счётчик: <input type="button" value="2" data-counter>  
<script>  
  document.addEventListener('click', function(event) {  
    if (event.target.dataset.counter != undefined) { // если есть  
атрибут...  
      event.target.value++;  
    }  
  });  
</script>
```

Счётчик:  Ещё счётчик:  Счётчик:  Ещё счётчик:

Якщо натиснути на кнопку - значення збільшаться. 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">  
  Показать форму подписки  
</button>  
<form id="subscribe-mail" hidden>  
  Ваша почта: <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 подписки

Показати форму подписки

Ваша почта:

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.

## Лекція 16

### Події миші [10]

Розглянемо події миші і їх властивості.

Відразу зауважимо: 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

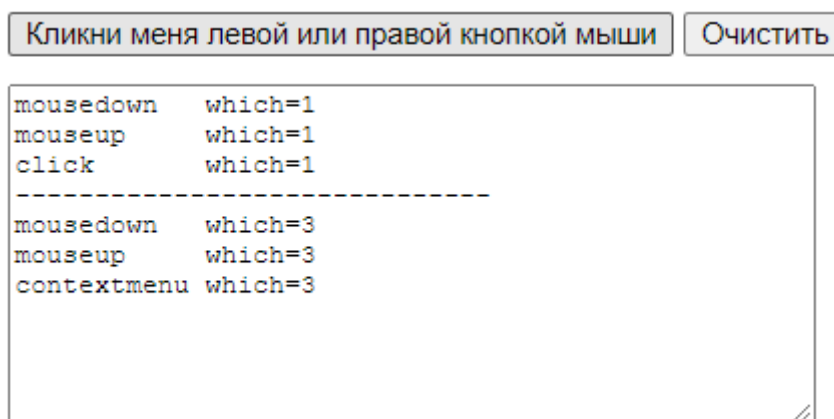
Одна дія може викликати кілька подій.

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 потрібно, тому що ці події спрацьовують на будь-якій кнопці, і which дозволяє розрізняти 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

Середня кнопка використовується дуже рідко.

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)

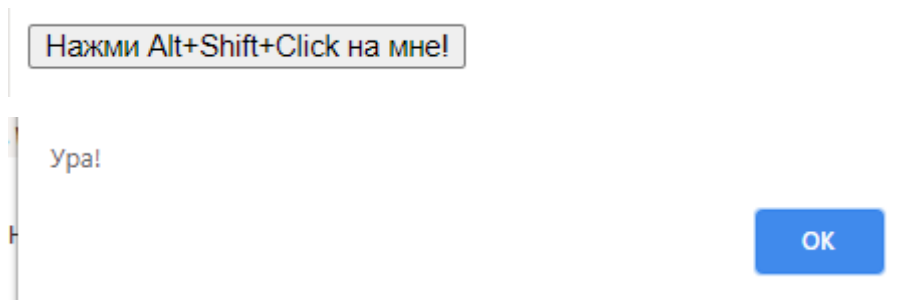
**ctrlKey**: Ctrl

**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 id="button">Нажми Alt+Shift+Click на мене!</button>
<script>
  button.onclick = function(event) {
    if (event.altKey && event.shiftKey) {
      alert('Ура!');
    }
  };
</script>
```



### 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="Наведи на меня мышь">
```



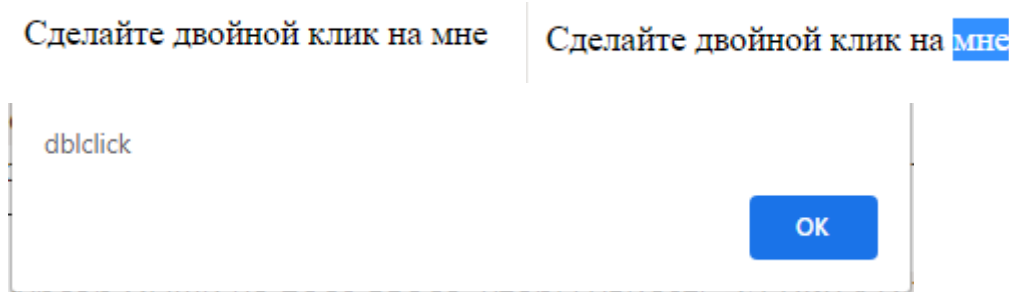
Координати щодо документа pageX, pageY відраховуються не від вікна, а від лівого верхнього кута документа.

### 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:

```
span ondblclick="alert('dblclick')">Сделайте двойной  
клик на мне</span>
```



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:

```
До...  
<b ondblclick="alert('Клик!')" onmousedown="return false">  
Сделайте двойной клик на мне  
</b>  
...После
```

До... Сделайте двойной клик на мне ...После



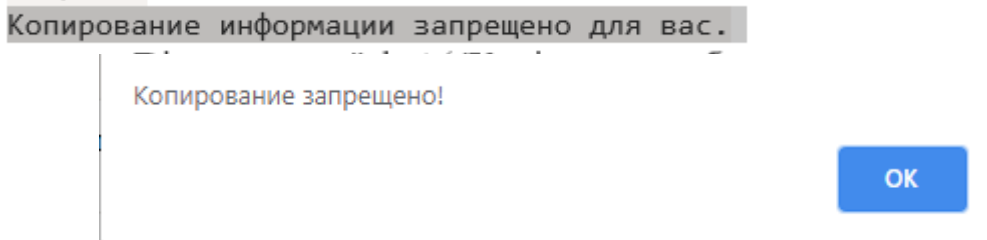
Тепер виділений жирним елемент не виділяється при подвійному натисканні, а також на ньому не можна почати виділення, затиснувши кнопку миші.

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('Копирование запрещено!');return false">  
Копирование информации запрещено для вас.  
</div>
```



Якщо ви спробуєте скопіювати текст в `<div>`, у вас це не вийде, тому що спрацьовування події **oncopy** за замовчуванням заборонено.

Звичайно, користувач має доступ до HTML-коду сторінки і може взяти текст звідти, але не всі знають, як це зробити.

### 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) отследить нажатие

    // (2) подготовит к перемещению:
    // разместить поверх остального содержимого и в абсолютных
    координатах
    ball.style.position = 'absolute';
    ball.style.zIndex = 1000;
    // переместим в body, чтобы мяч был точно не внутри
    position:relative
    document.body.append(ball);
    // и установим абсолютно спозиционированный мяч под курсор
```

```
moveAt(event.pageX, event.pageY);
```

```
// передвинуть мяч под координаты курсора  
// и сдвинуть на половину ширины/высоты для центрирования  
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) перемещать по экрану  
document.addEventListener('mousemove', onMouseMove);  
// (4) положить мяч, удалить более ненужные обработчики событий  
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);

    // переносит мяч на координаты (pageX, pageY),
    // дополнительно учитывая изначальный сдвиг относительно указателя
    // мыши
    function moveAt(pageX, pageY) {
        ball.style.left = pageX - shiftX + 'px';
        ball.style.top = pageY - shiftY + 'px';
    }

    function onMouseMove(event) {
        moveAt(event.pageX, event.pageY);
    }

    // передвигаем мяч при событии mousemove
    document.addEventListener('mousemove', onMouseMove);

    // отпустить мяч, удалить ненужные обработчики
    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 (draggable)**

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» (draggable).

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('нікогда не
сработает')"></div>
  <div style="background:red" onmouseover="alert('над
```

красным!')"></div>



Те ж саме з перетягуванням елементів. М'яч завжди знаходиться поверх інших елементів, тому події спрацьовують на ньому. Які б 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:

```
// внутри обработчика события мыши
ball.hidden = true; // (*) прячем переносимый элемент

let elemBelow = document.elementFromPoint(event.clientX,
event.clientY);
// elemBelow - элемент под мячом (возможная цель переноса)

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:

```
// потенциальная цель переноса, над которой мы пролетаем прямо сейчас
let currentDraggable = null;

function onMouseMove(event) {
  moveAt(event.pageX, event.pageY);

  ball.hidden = true;
```

```

    let elemBelow = document.elementFromPoint(event.clientX,
event.clientY);
    ball.hidden = false;

    // событие mousemove может произойти и когда указатель за пределами
окна
    // (мяч перетаскивали за пределы экрана)

    // если clientX/clientY за пределами окна, elementFromPoint вернёт
null
    if (!elemBelow) return;

    // потенциальные цели переноса помечены классом droppable (может
быть и другая логика)
    let droppableBelow = elemBelow.closest('.droppable');

    if (currentDroppable !== droppableBelow) {
        // мы либо залетаем на цель, либо улетаем из неё
        // внимание: оба значения могут быть null
        //   currentDroppable=null,
        //   если мы были не над droppable до этого события (например,
над пустым пространством)
        //   droppableBelow=null,
        //   если мы не над droppable именно сейчас, во время этого
события

        if (currentDroppable) {
            // логика обработки процесса "вылета" из droppable (удаляем
подсветку)
            leaveDroppable(currentDroppable);
        }
        currentDroppable = droppableBelow;
        if (currentDroppable) {
            // логика обработки процесса, когда мы "влетаем" в элемент
droppable
            enterDroppable(currentDroppable);
        }
    }
}

```

У наведеному нижче прикладі, коли м'яч перетягується через футбольні ворота, ворота підсвічуються.

```

<html>
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <p>Перетащите мяч.</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 = pageIndex - 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;
        let draggableBelow = elemBelow.closest('.draggable');
        if (currentDraggable !== draggableBelow) {
          if (currentDraggable) { // null если мы были не над
draggable до этого события
            // (например, над пустым пространством)
            leaveDraggable(currentDraggable);
          }
          currentDraggable = draggableBelow;
          if (currentDraggable) { // null если мы не над draggable
сейчас, во время этого события
            // (например, только что покинули draggable)
            enterDraggable(currentDraggable);

```

```

    }
  }
}
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>
style.css

```

```

#gate {
  cursor: pointer;
  margin-bottom: 100px;
  width: 83px;
  height: 46px;
}

#ball {
  cursor: pointer;
  width: 40px;
  height: 40px;
}

```

Тепер протягом всього процесу в змінної **currentDroppable** ми зберігаємо поточну потенційну ціль перенесення, над якою ми зараз, можемо її підсвітити або зробити щось ще.

## Лекція 17

### Форма

Тег **<form>** встановлює форму на веб-сторінці. Форма призначена для обміну даними між користувачем і сервером. Область застосування форм не обмежена відправкою даних на сервер, за допомогою клієнтських скриптів можна отримати доступ до будь-якого елементу форми, змінювати його і застосовувати на власний розсуд.

Документ може містити будь-яку кількість форм, але одночасно на сервер може бути відправлена тільки одна форма. З цієї причини дані форм повинні бути незалежні один від одного.

Для відправки форми на сервер використовується кнопка Submit, того ж можна домогтися, якщо натиснути клавішу Enter в межах форми. Якщо кнопка Submit відсутня в формі, клавіша Enter імітує її використання.

Коли форма відправляється на сервер, управління даними передається програмі, заданої атрибутом **action** тега **<form>**. Попередньо браузер готує інформацію у вигляді пари «ім'я = значення», де ім'я визначається атрибутом **name** тега **<input>**, а значення введено користувачем або встановлено в поле форми за замовчуванням.

Допускається всередину контейнера **<form>** поміщати інші теги, при цьому сама форма ніяк не відображається на веб-сторінці, видно тільки її елементи і результати вкладених тегів.

Синтаксис

```
<form action="URL"  
...  
</form>
```

Атрибути:

**accept-charset** встановлює кодування, в якій сервер може приймати та обробляти дані.

**action** адреса програми або документа, який обробляє дані форми.

**autocomplete** включає автозаповнення полів форми.

**enctype** спосіб кодування даних форми.



**method** метод протоколу HTTP.

**name** ім'я форми.

**novalidate** скасовує вбудовану перевірку даних форми на коректність введення.

**target** ім'я вікна або фрейму, куди обробник буде завантажувати результат повернення.

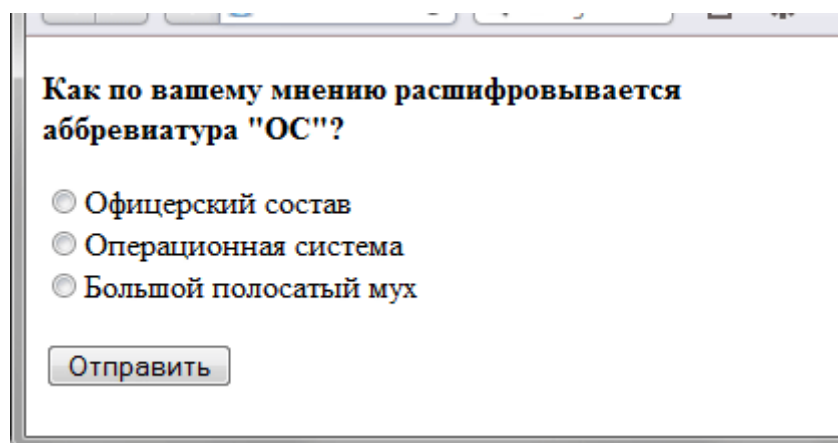
Також для цього тега доступні універсальні атрибути і події (class, id, style, contextmenu, onclick, onchange, onfocus, onload, onmousedown, onmousemove, onmouseout, onmouseup, onmouseover та ін.).

Приклад:

```
<html>
<head>
  <meta charset="utf-8">
  <title>Тег FORM</title>
</head>
<body>

  <form action="handler.php">
    <p><b>Как по вашему мнению расшифровывается аббревиатура
    &quot;ОС&quot;?</b></p>
    <p><input type="radio" name="answer" value="a1">Офицерский состав<br>
    <input type="radio" name="answer" value="a2">Операционная система<br>
    <input type="radio" name="answer" value="a3">Большой полосатый мух</p>
    <p><input type="submit"></p>
  </form>

</body>
</html>
```



Тег **<input>** є одним з елементів форми і дозволяє створювати різні елементи інтерфейсу і забезпечити взаємодію з користувачем. Головним

чином `<input>` призначений для створення текстових полів, різних кнопок, перемикачів і прапорців. Хоча елемент `<input>` не потрібно поміщати всередину контейнера `<form>`, що визначає форму, але якщо введені користувачем дані повинні бути відправлені на сервер, де їх обробляє серверна програма, то вказувати `<form>` обов'язково. Те ж саме відбувається і в разі обробки даних за допомогою клієнтських додатків, наприклад, скриптів на мові JavaScript.

Основний атрибут тега `<input>`, що визначає вид елемента - `type`. Він дозволяє задавати такі елементи форми: текстове поле (**text**), поле з паролем (**password**), перемикач (**radio**), прапорець (**checkbox**), приховане поле (**hidden**), кнопка (**button**), кнопка для відправки форми (`submit`), кнопка для очищення форми (**reset**), поле для відправки файлу (**file**) і кнопка із зображенням (**image**). Для кожного елемента існує свій список атрибутів, які визначають його вигляд і характеристики.

Атрибути:

**accept** встановлює фільтр на типи файлів, які ви можете відправити через поле завантаження файлів.

**accesskey** перехід до елемента за допомогою комбінації клавіш.

**align** визначає вирівнювання зображення.

**alt** альтернативний текст для кнопки із зображенням.

**autocomplete** включає або відключає автозаповнення.

**autofocus** встановлює фокус в поле форми.

**border** товщина рамки навколо зображення.

**checked** попередньо активований перемикач або прапорець.

**disabled** блокує доступ і зміну елемента.

**form** пов'язує поле з формою по її ідентифікатором.

**formaction** визначає адресу обробника форми.

**formenctype** встановлює спосіб кодування даних форми при їх відправленні на сервер.

**formmethod** повідомляє браузеру яким методом слід передавати дані форми на сервер.

**formnovalidate** скасовує вбудовану перевірку даних на коректність.

**formtarget** визначає вікно або фрейм в яке буде завантажуватися результат, що повертається оброблювачем форми.

**list** вказує на список варіантів, які можна вибирати при введенні тексту.

**max** верхнє значення для введення числа або дати.

**maxlength** максимальна кількість символів дозволених в тексті.

**min** нижнє значення для введення числа або дати.

**multiple** дозволяє завантажити кілька файлів одночасно.

**name** ім'я поля, призначене для того, щоб обробник форми міг його ідентифікувати.

**pattern** встановлює шаблон введення.

**placeholder** виводить підказує текст.

**readonly** встановлює, що поле не може змінюватися користувачем.

**required** обов'язкове для заповнення поле.

**size** ширина текстового поля.

**src** адреса графічного файлу для поля із зображенням.

**step** крок збільшення для числових полів.

**tabindex** визначає порядок переходу між елементами за допомогою клавіші Tab.

**type** повідомляє браузеру, до якого типу належить елемент форми.

**value** значення елемента.

Також для цього тега доступні універсальні атрибути і події (class, id, style, contextmenu, onclick, onchange, onfocus, onload, onmousedown, onmousemove, onmouseout, onmouseup, onmouseover та ін.).

Приклад:

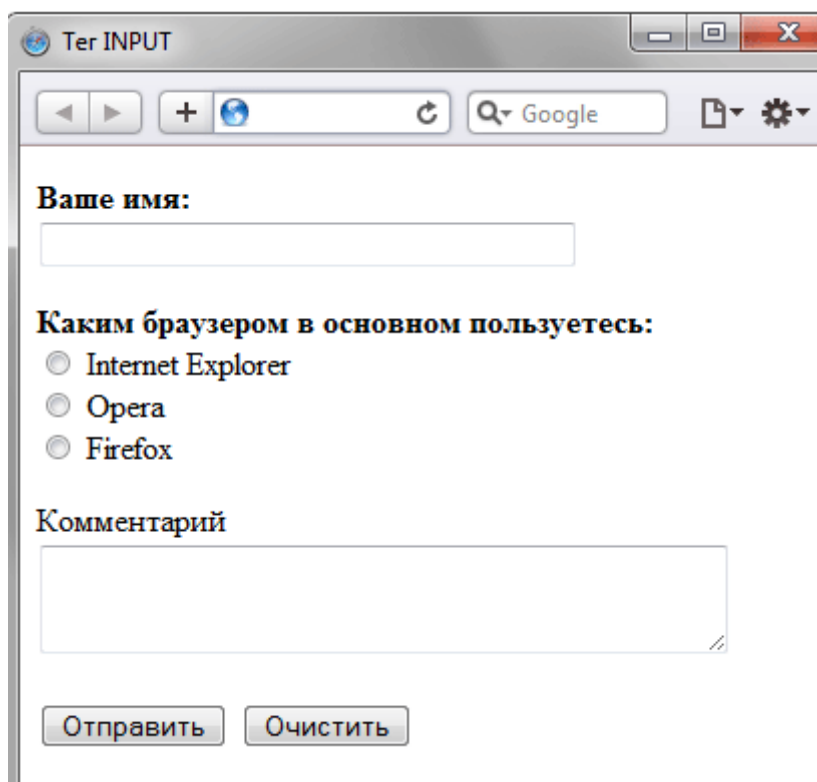
```
<html>
<head>
  <meta charset="utf-8">
  <title>Тег INPUT</title>
</head>
```

```
<body>
```

```
<form name="test" method="post" action="input1.php">  
<p><b>Ваше имя:</b><br>  
  <input type="text" size="40">  
</p>  
<p><b>Каким браузером в основном пользуетесь:</b><br>  
  <input type="radio" name="browser" value="ie"> Internet Explorer<br>  
  <input type="radio" name="browser" value="opera"> Opera<br>  
  <input type="radio" name="browser" value="firefox"> Firefox<br>  
</p>  
<p>Комментарий<br>  
  <textarea name="comment" cols="40" rows="3"></textarea></p>  
<p><input type="submit" value="Отправить">  
  <input type="reset" value="Очистить"></p>  
</form>
```

```
</body>
```

```
</html>
```



Тег **<button>** створює на веб-сторінці кнопки і за своєю дією нагадує результат, одержаний за допомогою тега **<input>** (з атрибутом `type = "button | reset | submit"`). На відміну від цього тега, **<button>** пропонує розширені можливості по створенню кнопок. Наприклад, на подібній кнопці можна розміщувати будь-які елементи HTML, в тому числі зображення. Використовуючи стилі можна визначити вид кнопки шляхом зміни шрифту, кольору фону, розмірів і інших параметрів.

Теоретично, тег **<button>** повинен розташовуватися усередині форми, яка встановлюється елементом **<form>**. Проте, браузер не виводить повідомлення про помилку і коректно працюють з тегом **<button>**, якщо він зустрічається самостійно. Однак, якщо необхідно результат натискання на кнопку відправити на сервер, поміщати **<button>** в контейнер **<form>** обов'язково.

Синтаксис

```
<form>  
  <button>...</button>  
</form>
```

Атрибути:

**accesskey** доступ до елементів форми за допомогою гарячих клавіш.

**autofocus** встановлює, що кнопка отримує фокус після завантаження сторінки.

**disabled** блокує доступ і зміна елемента.

**form** пов'язує між собою форму і кнопку.

**formaction** задає адресу, на яку пересилаються дані форми при натисканні на кнопку.

**formenctype** спосіб кодування даних форми.

**formmethod** вказує метод пересилання даних форми.

**formnovalidate** скасовує перевірку форми на коректність.

**formtarget** відкриває результат відправки форми в новому вікні або фреймі.

**name** визначає унікальне ім'я кнопки.

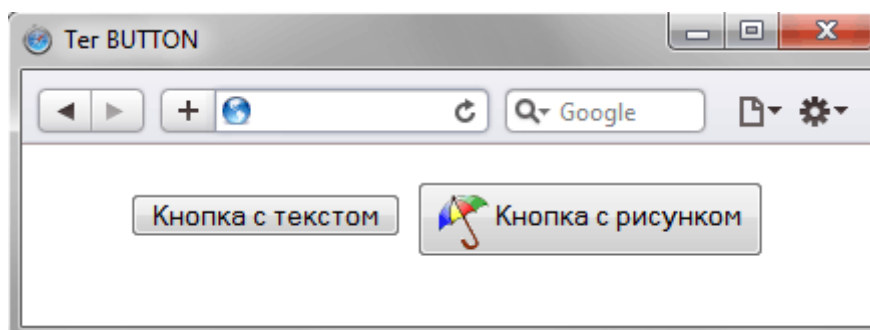
**type** тип кнопки - звичайна, для відправки даних форми на сервер або для очищення форми.

**value** значення кнопки, яке буде відправлено на сервер або прочитано за допомогою скриптів.

Також для цього тега доступні універсальні атрибути і події (class, id, style, contextmenu, onclick, onchange, onfocus, onload, onmousedown, onmousemove, onmouseout, onmouseup, onmouseover та ін.).

Приклад:

```
<html>
<head>
  <meta charset="utf-8">
  <title>Ter BUTTON</title>
</head>
<body>
  <p style="text-align: center"><button>Кнопка с текстом</button>
  <button> Кнопка с рисунком</button></p>
</body>
</html>
```



### Властивості і методи форми [10]

Форми і елементи управління, такі як `<input>`, мають безліч спеціальних властивостей і подій.

#### Навігація: форми і елементи

Форми в документі входять в спеціальну колекцію **document.forms**.

Це так звана «іменована» колекція: ми можемо використовувати для отримання форми як її ім'я, так і порядковий номер в документі.

```
document.forms.my - форма с именем "my" (name="my")
document.forms[0] - первая форма в документе
```

Коли ми вже отримали форму, будь-який елемент доступний в іменованій колекції **form.elements**.

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>
```

```

<script>
  // получаем форму
  let form = document.forms.my; // <form name="my"> element
  // получаем элемент
  let elem = form.elements.one; // <input name="one"> element
  alert(elem.value); // 1
</script>

```

Може бути кілька елементів з одним і тим же ім'ям, це часто буває з кнопками-перемикачами radio.

В цьому випадку `form.elements [name]` є колекцією, наприклад:

```

<form>
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>

<script>
let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0]); // [object HTMLInputElement]
</script>

```

Ці навігаційні властивості не залежать від структури тегів всередині форми. Всі елементи управління форми, як би глибоко вони не знаходилися в формі, доступні в колекції **form.elements**.

Форма може містити один або кілька елементів **<fieldset>** всередині себе. Вони також підтримують властивість **elements**, в якій присутні елементи управління всередині них.

```

<body>
  <form id="form">
    <fieldset name="userFields">
      <legend>info</legend>
      <input name="login" type="text">
    </fieldset>
  </form>
  <script>
    alert(form.elements.login); // <input name="login">
    let fieldset = form.elements.userFields;
    alert(fieldset); // HTMLFieldSetElement
    // мы можем достать элемент по имени как из формы, так и из
    fieldset с ним
    alert(fieldset.elements.login == form.elements.login); // true
  </script>

```

```
</body>
```

Зворотнє посилання: **element.form**

Для будь-якого елемента форма доступна через **element.form**. Форма посилається на всі елементи, а ці елементи посилаються на форму.

## Елементи форми

Розглянемо елементи управління, які використовуються в формах.

### input і textarea

До їх значення можна отримати доступ через властивість **input.value** (рядок) або **input.checked** (логічне значення) для чекбоксів.

Ось так:

```
input.value = "Новое значение";  
textarea.value = "Новый текст";  
input.checked = true; // для чекбоксов и переключателей
```

### select і option

Елемент **<select>** має 3 важливі властивості:

**select.options** - колекція з піделементів **<option>**,  
**select.value** - значення обраного в даний момент **<option>**,  
**select.selectedIndex** - номер обраного **<option>**.

Вони дають три різні способи встановити значення в **<select>**:

1. Знайти відповідний елемент **<option>** і встановити в **option.selected** значення **true**.
2. Встановити в **select.value** значення потрібного **<option>**.
3. Встановити в **select.selectedIndex** номер потрібного **<option>**.

Перший спосіб найбільш зрозумілий, але (2) і (3) є більш зручними при роботі.

Ось ці способи на прикладі:

```
<select id="select">
```



```

<option value="apple">Яблоко</option>
<option value="pear">Груша</option>
<option value="banana">Банан</option>
</select>

<script>
  // все три строки делают одно и то же
  select.options[2].selected = true;
  select.selectedIndex = 2;
  select.value = 'banana';
</script>

```

### new Option

Елемент `<option>` рідко використовується сам по собі, але і тут є дещо цікаве.

У специфікації є красивий короткий синтаксис для створення елемента `<option>`:

```
option = new Option(text, value, defaultSelected,
selected);
```

Параметри:

**text** - текст всередині `<option>`,

**value** - значення,

**defaultSelected** - якщо true, то ставиться HTML-атрибут `selected`,

**selected** - якщо true, то елемент `<option>` буде обраним.

Тут може бути невелика плутанина з `defaultSelected` і `selected`. Все просто: `defaultSelected` задає HTML-атрибут, його можна отримати як `option.getAttribute ('selected')`, а `selected` - вибрано значення чи ні, саме його важливо поставити правильно. Втім, зазвичай ставлять обидва цих значення в true або не ставлять зовсім (тобто false).

Приклад:

```
let option = new Option("Текст", "value");
// создаст <option value="value">Текст</option>
```

Той же елемент, але обраний:

```
let option = new Option("Текст", "value", true, true);
```

Елементи **<option>** мають властивості:

**option.selected** обрана опція.

**option.index** номер опції серед інших в списку **<select>**.

**option.text** вміст опції (то, що бачить відвідувач).

## Фокусування: focus / blur

### Події focus / blur

Подія focus викликається в момент фокусування, а blur - коли елемент втрачає фокус.

Використовуємо їх для валідації (перевірки) введених даних.

У прикладі нижче:

Обробник blur перевіряє, чи введений email, і якщо ні - показує помилку.

Обробник focus приховує це повідомлення про помилку (в момент втрати фокусу перевірка повториться):

```
<style>
  .invalid { border-color: red; }
  #error { color: red }
</style>
Ваш email: <input type="email" id="input">
<div id="error"></div>
<script>
input.onblur = function() {
  if (!input.value.includes('@')) { // не email
    input.classList.add('invalid');
    error.innerHTML = 'Пожалуйста, введите правильный email.'
  }
};
input.onfocus = function() {
  if (this.classList.contains('invalid')) {
    // удаляем индикатор ошибки, т.к. пользователь хочет ввести
    данные заново
    this.classList.remove('invalid');
    error.innerHTML = "";
  }
};
</script>
```

Ваш email:

Ваш email:   
Пожалуйста, введите правильный email.

Сучасний HTML дозволяє робити валідацію за допомогою атрибутів **required**, **pattern** і т.д. Іноді - це все, що нам потрібно. JavaScript можна використовувати, коли ми хочемо більше гнучкості. А ще ми могли б відправляти змінене значення на сервер, якщо воно правильне.

### Методи **focus** / **blur**

Методи **elem.focus ()** і **elem.blur ()** встановлюють / знімають фокус.

Наприклад, заборонимо відвідувачеві перемикатися з поля введення, якщо введене значення не минуло валідацію:

```
<style>
  .error {
    background: red;
  }
</style>
Ваш email: <input type="email" id="input">
<input type="text" style="width:280px" placeholder="введите
неверный email и кликните сюда">
<script>
  input.onblur = function() {
    if (!this.value.includes('@')) { // не email
      // показать ошибку
      this.classList.add("error");
      // ...и вернуть фокус обратно
      input.focus();
    } else {
      this.classList.remove("error");
    }
  };
</script>
```

Ваш email:

Це спрацює у всіх браузерах, крім Firefox (bug).

Якщо ми що-небудь введемо і натиснемо Tab або клікнемо в інше місце, тоді onblur поверне фокус назад.

Відзначимо, що ми не можемо «скасувати втрату фокусу», викликавши `event.preventDefault ()` в обробнику `onblur` тому, що `onblur` спрацьовує після втрати фокусу елементом.

### Вмикаємо фокусування на будь-якому елементі: **tabindex**

Багато елементів за замовчуванням не підтримують фокусування.

Які саме - залежить від браузера, але одне завжди вірно: підтримка **focus** / **blur** гарантована для елементів, з якими відвідувач може взаємодіяти: `<button>`, `<input>`, `<select>`, `<a>` і т.д.

З іншого боку, елементи форматування `<div>`, `<span>`, `<table>` - за замовчуванням не можуть отримати фокус. Метод `elem.focus ()` не працює для них, і події **focus** / **blur** ніколи не спрацьовують.

Це можна змінити HTML-атрибутом **tabindex**.

Будь-який елемент підтримує фокусування, якщо має **tabindex**. Значення цього атрибута - порядковий номер елемента, коли клавіша Tab (або щось аналогічне) використовується для перемикавання між елементами.

Тобто: якщо у нас два елементи, перший має `tabindex = "1"`, а другий `tabindex = "2"`, то перебуваючи в першому елементі і натиснувши Tab - ми перемістимося в другій.

Порядок перебору такий: спочатку йдуть елементи зі значеннями `tabindex` від 1 і вище, в порядку `tabindex`, а потім елементи без `tabindex` (наприклад, звичайний `<input>`).

При співпадаючих **tabindex** елементи перебираються в тому порядку, в якому йдуть в документі.

Є два спеціальних значення:

**tabindex = "0"** ставить елемент в один ряд з елементами без `tabindex`. Тобто, при перемиканні такі елементи будуть після елементів з **tabindex**  $\geq 1$ .

Зазвичай використовується, щоб включити фокусування на елементі, але не змінювати порядок перемикавання. Щоб елемент міг брати участь в формі нарівні зі звичайними `<input>`.

**tabindex = "- 1"** дозволяє фокусуватися на елементі тільки програмно. Клавіша Tab проігнорує такий елемент, але метод **elem.focus ()** буде діяти.

Наприклад, список нижче:

```
<ul>
  <li tabindex="1">Один</li>
  <li tabindex="0">Ноль</li>
  <li tabindex="2">Два</li>
  <li tabindex="-1">Минус один</li>
</ul>

<style>
  li { cursor: pointer; }
  :focus { outline: 1px dashed green; }
</style>
```

Порядок такий: 1 - 2 - 0. Зазвичай <li> не підтримує фокусування, але tabindex включає його, а також події і стилізацію псевдоклас: **focus**.

### Події focusin / focusout

Події **focus** і **blur** не спливають.

Наприклад, ми не можемо використовувати **onfocus** на <form>, щоб підсвітити її:

Приклад вище не працює, тому що коли користувач переміщує фокус на <input>, подія **focus** спрацьовує тільки на цьому елементі. Ця подія не спливає. Отже, **form.onfocus** ніколи не спрацьовує.

Ця проблема має два рішення.

Перше: особливість - focus / blur не спливають, але передаються вниз на фазі перехоплення.

Це спрацює:

```
<form id="form">
  <input type="text" name="name" value="Имя">
  <input type="text" name="surname" value="Фамилия">
</form>
<style> .focused { outline: 1px solid red; } </style>
<script>
  // установить обработчик на фазе перехвата (последний аргумент true)
  form.addEventListener("focus", () => form.classList.add('focused'),
true);
```

```
form.addEventListener("blur", () =>
form.classList.remove('focused'), true);
</script>
```

Друге рішення: події **focusin** і **focusout** - такі ж, як і **focus** / **blur**, але вони спливають.

Зауважте, що ці події повинні використовуватися з *elem.addEventListener*, але не з **on** *<event>*.

Другий робочий варіант:

```
<form id="form">
  <input type="text" name="name" value="Имя">
  <input type="text" name="surname" value="Фамилия">
</form>

<style> .focused { outline: 1px solid red; } </style>

<script>
  form.addEventListener("focusin", () =>
form.classList.add('focused'));
  form.addEventListener("focusout", () =>
form.classList.remove('focused'));
</script>
```

## Події: change, input, cut, copy, paste

Розглянемо різні події, супутні оновленню даних.

### Подія: change

Подія **change** спрацьовує після закінчення зміни елемента.

Для текстових **<input>** це означає, що подія відбувається при втраті фокуса.

Поки ми друкуємо в текстовому полі в прикладі нижче, подія не відбувається. Але коли ми переміщаємо фокус в інше місце, наприклад, натискаючи на кнопку, то відбудеться подія **change**:

```
<input type="text" onchange="alert(this.value)">
<input type="button" value="Button">
```

Для інших елементів: **select**, **input type = checkbox / radio** подія запускається відразу після зміни значення:

```
<select onchange="alert(this.value)">
  <option value="">Выберите что-нибудь</option>
  <option value="1">Вариант 1</option>
  <option value="2">Вариант 2</option>
  <option value="3">Вариант 3</option>
</select>
```

### Подія: **input**

Подія **input** спрацьовує кожного разу при зміні значення.

На відміну від подій клавіатури, вона працює при будь-яких змінах значень, навіть якщо вони не пов'язані з клавіатурними діями: вставка за допомогою миші або розпізнавання мови під диктовку тексту.

Наприклад:

```
<input type="text" id="input"> oninput: <span id="result"></span>
<script>
  input.oninput = function() {
    result.innerHTML = input.value;
  };
</script>
```

Якщо ми хочемо обробляти кожну зміну в **<input>**, то ця подія є найкращим вибором.

З іншого боку, подія **input** не відбувається при введенні з клавіатури або інших діях, якщо при цьому не змінюється значення в текстовому полі, тобто натискання клавіш **↵**, **⇨** і подібних при фокусі на текстовому полі не викличуть цю подію.

### Події: **cut, copy, paste**

Ці події відбуваються під час **вирізання / копіювання / вставки** даних.

Вони відносяться до класу **ClipboardEvent** і забезпечують доступ до даних, які копіюються / вставляються.

Ми також можемо використовувати **event.preventDefault ()** для запобігання дії за замовчанням, і в підсумку нічого не скопіюється / не вставиться.

Наприклад, код, наведений нижче, запобігає всім подібним подіям і показує, що ми намагаємося вирізати / копіювати / вставити:

```
<input type="text" id="input">
<script>
  input.oncut = input.oncopy = input.onpaste = function(event) {
    alert(event.type + ' - ' +
event.clipboardData.getData('text/plain'));
    return false;
  };
</script>
```



## ПЕЛІК ЛІТЕРАТУРИ

1. Хмелюк М.С. Конспект лекцій з дисципліни «Основи клієнтської розробки»
2. Хмелюк М.С. Методичні вказівки до виконання лабораторних робіт для студентів з дисципліни «Основи клієнтської розробки»
3. Lynn Ben Git Magic, publish: CreateSpace, 2012. – 68с
4. Lynn Ben Mercurial: The Definitive Guide, publish: O'Reilly Media, Inc., 2009.
5. Скот Чакон, Бен Штрауб Pro Git — профессиональный контроль версий, изд. – СПб.: Питер, 2019.- 496с.
6. Б. Лоусон, Р. Шарп - Изучаем HTML 5, изд: Питер, 2011
7. П. Лабберс — HTML 5 для профессионалов, изд: Вильямс, 2011
8. Бен Хеник — HTML и CSS Путь к совершенству, изд: O'Reilly (Питер), 2011. -336с
9. Дэвид Флэнаган JavaScript. Подробное руководство, изд.: Символ, 2017. -1080
10. Интернет-ресурс <https://learn.javascript.ru/>
11. Интернет-ресурс <https://git-scm.com/book/ru/v2>
12. Интернет-ресурс <http://habr.com>
13. Интернет-ресурс <https://puzzleweb.ru/>