До рішення

Типи даних

Значення в JavaScript завжди має певний тип даних. Наприклад, рядок або число.

У JavaScript є вісім основних типів даних. У цьому розділі ми розглянемо їх в цілому, а в наступних — детально поговоримо про кожен з них.

Ми можемо призначити змінній будь-який тип даних. Наприклад, в один момент змінна може бути рядком, а в інший – числом:

```
// тут не буде помилки
let message = "привіт";
message = 123456;
```

Мови програмування, які дають змогу таке робити, називаються “динамічно типізованими”. Мається на увазі, що типи даних визначені, але змінні не прив’язанні до жодного типу.

Число (number)

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

Тип *number* представляє і цілі числа, і числа з рухомою точкою.

Є багато операцій, що можна робити з числами, наприклад, множення `*`, ділення `/`, додавання `+`, віднімання `-` тощо.

Окрім звичайних чисел, є так звані “спеціальні числові значення”, що також мають відношення до цього типу даних: `Infinity`, `-Infinity` і `NaN`.

- `Infinity` являє собою математичну нескінченність ∞. Це спеціальне значення, що є більшим за будь-яке число.

  Ми можемо отримати його як результат ділення на нуль:

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

  Або безпосередньо посилатися на нього:

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

- `NaN` (Not a Number) являє собою помилку обчислення. Це є результат неправильної або невизначеної математичної операції, наприклад:

  ```
  alert("not a number" / 2); // NaN, таке ділення є помилковим
  ```

`NaN` є “причепливим” (“заразливим”). Будь-яка подальша математична операція з `NaN` повертає `NaN`:

```
alert( NaN + 1 ); // NaN
alert( 3 * NaN ); // NaN
alert( "not a number" / 2 - 1 ); // NaN
```

Отже, якщо десь у математичному виразі є `NaN`, він поширюється на весь результат (є лише один виняток: результатом операції `NaN ** 0` буде `1`).

Математичні операції є безпечними

Обчислення є “безпечним” в JavaScript. Ми можемо робити будь-що: ділити на нуль, звертатися до нечислового рядка як до числа тощо.

Виконання скрипта ніколи не зупиниться з фатальною помилкою (не “вмре”). У найгіршому випадку ми отримаємо в результаті `NaN`.

Спеціальні числові значення формально належать до типу “number”. Хоча, звісно, вони не є числами у загальноприйнятому розумінні.

Докладніше роботу з числами ми розглянемо у розділі Числа.

BigInt

У JavaScript, тип “number” не може містити числа більші за $(2^{53}-1)$ (це `9007199254740991`), або менші за $-(2^{53}-1)$ для від’ємних чисел. Це технічне обмеження, спричинене їхньою внутрішньою реалізацією.

Для більшості потреб цього достатньо, але бувають випадки, коли нам потрібні дійсно великі числа, наприклад, для криптографії або мікроксекундних часових міток (timestamps).

Нещодавно в мову був доданий тип `BigInt` для представлення цілих чисел довільної довжини.

Значення з типом `BigInt` створюється через додавання `n` у кінець цілого числа:

```
// буква "n" у кінці означає, що це число типу BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Через те, що тип `BigInt` рідко використовується, ми не розглядатимемо його в цьому розділі, проте ми винесли його в окремий розділ BigInt. Прочитайте його, якщо вам потрібні такі великі числа.

Проблеми із сумісністю

Цієї миті, підтримка типу `BigInt` є в останніх версіях Firefox/Chrome/Edge/Safari, але не в IE.

На сайті *MDN* є таблиця сумісності , де показано, які версії браузерів підтримують тип `BigInt`.

Рядок (string)

Рядок у JavaScript має бути оточений лапками.

```
let str = "Привіт";
let str2 = 'Одинарні лапки також дозволяються';
let phrase = `так можна вставляти ${str}`;
```

У JavaScript є три типи лапок.

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

Подвійні та одинарні лапки є “звичайними”. Тобто немає ніякої різниці, які саме використовувати.

Зворотні лапки є розширенням функціональності. Вони дають змогу вбудовувати змінні та вирази в рядок, обрамляючи їх в `${…}`, наприклад:

```
let name = "Іван";

// вбудована змінна
alert(`Привіт, ${name}е!`); // Привіт, Іване!

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

Вираз всередині `${…}` обчислюється, а результат обчислення стає частиною рядка. Ми можемо вбудувати будь-що: змінну `name`, або арифметичний вираз `1 + 2`, або щось набагато складніше.

Будь ласка, зауважте, що вбудовування можна робити тільки зі зворотніми лапками. Інші типи лапок не мають функціональності вбудовування!

```
alert("результат: ${1 + 2}"); // результат: ${1 + 2} (подвійні лапки не мають ніякого впливу)
```

Більш детально ми будемо висвітлювати рядки в розділі Рядки.

> **Немає типу *символ* (*character*).**
>
> У деяких мовах є спеціальний тип “character” для позначення єдиного символу. Наприклад, у мовах C та Java це `char`.
>
> У JavaScript немає такого типу. Є єдиний тип: `string`. Рядок може містити нуль символів (бути пустим), один символ або більше.

Булевий або логічний тип (boolean)

Логічний тип може приймати лише два значення: `true` (істина) та `false` (хиба).

Цей тип зазвичай використовується для зберігання значень так/ні: `true` означає “так, вірно”, а `false` означає “ні, не вірно”.

Наприклад:

```
let nameFieldChecked = true; // так, ім’я було перевірене
let ageFieldChecked = false; // ні, вік не був перевіреним
```

Логічне значення також можна отримати як результат порівняння:

```
let isGreater = 4 > 1;

alert(isGreater); // true (результат порівняння – "так")
```

Більш глибоко ми охопимо булеві значення у розділі Логічні оператори.

Значення “null”

Спеціальне значення `null` не належить до жодного з описаних вище типів.

Воно формує окремий власний тип, який містить лише значення `null`:

```
let age = null;
```

В JavaScript `null` не є “посиланням на неіснуючий об’єкт” або “покажчиком на null”, як може бути в інших мовах програмування.

Це лише спеціальне значення, яке представляє “нічого”, “порожнє” або “невідоме значення”.

У наведеному вище коді зазначено, що значення змінної `age` невідоме.

Значення “undefined”

Спеціальне значення `undefined` також стоїть окремо. Воно представляє власний тип, подібний до “null”.

`undefined` означає, що “значення не присвоєно”.

Якщо змінна оголошена, але їй не присвоєне якесь значення, тоді значення такої змінної буде `undefined`:

```
let age;

alert(age); // покаже "undefined"
```

Технічно, є можливість явно призначити `undefined` змінній:

```
let age = 100;

// змінюємо значення на undefined
age = undefined;

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

…Але ми не рекомендуємо так робити. Як правило, ми використовуємо `null`, щоби присвоїти змінній значення “порожнє” або “невідоме”, тоді як `undefined` зарезервоване для позначення початкового значення для неприсвоєних речей.

Об’єкти (object) та символи (symbol)

Тип `object` є особливим типом.

Усі інші типи називаються “примітивами”, тому що їхні значення можуть містити тільки один елемент (це може бути рядок, число, або будь-що інше). В об’єктах же зберігаються колекції даних і більш складні структури.

Об’єкти є важливою частиною мови, тому ми окремо розглянемо їх у розділі Об’єкти після того, як дізнаємося більше про примітиви.

Тип `symbol` використовується для створення унікальних ідентифікаторів в об’єктах. Ми згадали цей тип для повноти, проте докладніше вивчимо його після об’єктів.

Оператор typeof

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

Виклик `typeof x` повертає рядок із назвою 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 можуть потребувати додаткового пояснення:

1. `Math` — це вбудований об’єкт, який забезпечує математичні операції. Ми вивчимо його в розділі Числа. Тут він використаний лише як приклад об’єкта.
2. Результатом `typeof null` є `"object"`. Це офіційно визнана помилка поведінки `typeof`, що є ще з ранніх днів JavaScript і зберігається для сумісності. Безперечно, `null` не є об’єктом. Це особливе значення з власним типом. У цьому разі поведінка `typeof` некоректна.
3. Результатом `typeof alert` є `"function"`, тому що `alert` — це функція. Ми будемо вивчати функції в наступних розділах, де ми також побачимо, що в JavaScript немає спеціального типу “function”. Функції належать до типу “об’єкт”. Але `typeof` трактує їх по-іншому, повертаючи `"function"`. Це також присутнє з ранніх днів JavaScript. Технічно, така поведінка не зовсім правильна, але може бути зручною на практиці.

Синтаксис `typeof(x)`

Можливо ви зустрічали інший синтаксис: `typeof(x)`. Це те саме, що `typeof x`.

Щоби було зрозуміло: `typeof` – це оператор, а не функція. Тут дужки не є частиною `typeof`. Це щось на кшталт математичних дужок для групування.

Зазвичай, такі дужки містять математичні вирази, як `(2 + 2)`, але тут вони містять лише один аргумент `(x)`. Ці дужки дають змогу опустити пробіл між оператором `typeof` та його аргументом, і декому це подобається.

Тому вони надають перевагу синтаксису `typeof(x)`, проте синтаксис `typeof x` набагато поширеніший.

Підсумки

У JavaScript є 8 основних типів.

- `number` для будь-яких чисел: цілих або з рухомою точкою; цілі числа обмежені до `±(2^53-1)`.
- `bigint` для цілих чисел довільної довжини.
- `string` для рядків. Рядок може мати нуль або більше символів, немає окремого типу для одного символу.
- `boolean` для `true`/`false`.
- `null` для невідомих значень — автономний тип, який має єдине значення `null`.
- `undefined` для неприсвоєних значень — автономний тип, який має єдине значення `undefined`.
- `object` для більш складних структур даних.
- `symbol` для унікальних ідентифікаторів.

Оператор `typeof` дає змогу нам бачити, який тип зберігається в змінній.

- Зазвичай використовують синтаксис `typeof x`, проте `typeof(x)` також можливий.
- Повертає рядок із назвою типу, як-от `"string"`.
- Для `null` повертає `"object"` —- це помилка в мові, `null` насправді не об’єкт.

У наступних розділах ми зосередимося на примітивних значеннях, а коли ознайомимося з ними, то перейдемо до об’єктів.

Завдання

Лапки у рядках

важливість: 5

Який буде результат виконання скрипта?

```
let name = "Ілля";

alert( `привіт ${1}` ); // ?

alert( `привіт ${"name"}` ); // ?

alert( `привіт ${name}` ); // ?
```

До рішення

Взаємодія: alert, prompt, confirm

Оскільки основним середовищем для демонстрації можливостей JavaScript буде браузер, давайте розглянемо декілька функцій для взаємодії з користувачем: `alert`, `prompt` та `confirm`.

alert

Ми вже бачили цю функцію. Вона показує повідомлення та чекає, доки користувач не натисне кнопку “ОК”.

Наприклад:

```
alert("Привіт");
```

Мінівікно з повідомленням називається *модальним вікном*. Слово “модальний” означає, що відвідувач не зможе взаємодіяти з іншою частиною сторінки, натискати інші кнопки тощо, доки не завершить операції з вікном. У цьому випадку – поки він не натисне “ОК”.

prompt

Функція `prompt` приймає два аргументи:

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

Вона показує модальне вікно з текстовим повідомленням, полем, куди відвідувач може ввести текст, та кнопками OK/Скасувати.

`title`

Текст, який буде відображатися для відвідувача.

`default`

Необов'язковий другий параметр, початкове значення для поля введення тексту.

> **Квадратні дужки в синтаксисі `[...]`**
>
> В синтаксисі вище, навколо `default` є квадратні дужки. Вони означають, що цей параметр є необов'язковим.

Відвідувач може щось ввести у поле введення і натиснути OK. Ми отримаємо введений текст в `result`. Однак, користувач може скасувати введення, натиснувши "Скасувати" або клавішу `Esc`. В цьому випадку `result` буде мати значення `null`.

Виклик `prompt` повертає текст із поля введення або `null`, якщо введення було скасовано.

Наприклад:

```
let age = prompt('Скільки вам років?', 100);

alert(`Вам ${age} років!`); // Вам 100 років!
```

> **В IE: завжди вказуйте початкове значення `default`**
>
> Другий параметр є необов'язковим, але якщо ми не надамо його, Internet Explorer вставить у поле текст `"undefined"`.
>
> Запустіть цей код в Internet Explorer, щоб переконатися:
>
> ```
> let test = prompt("Test");
> ```
>
> Отже, щоб модальні вікна `prompt` мали добрий вигляд у IE, ми рекомендуємо завжди надавати другий аргумент:
>
> ```
> let test = prompt("Test", ''); // <-- для IE
> ```

confirm

Синтаксис:

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

Функція `confirm` показує модальне вікно з питанням `question` та двома кнопками: OK та Скасувати.

Результат: `true`, якщо натиснути кнопку OK, інакше — `false`.

Наприклад:

```
let isBoss = confirm("Ви бос?");

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

Підсумки

Ми вивчили 3 специфічні для браузера функції для взаємодії з відвідувачами:

`alert`

показує повідомлення.

`prompt`

показує повідомлення з проханням ввести текст. Повертає цей текст або `null`, якщо натиснута кнопка “Скасувати” або клавіша `Esc`.

`confirm`

показує повідомлення і чекає, коли користувач натисне “OK” або “Скасувати”. Повертає `true` для OK та `false` для “Скасувати”/`Esc`.

Усі ці методи є модальними: вони призупиняють виконання скриптів та не дають відвідувачам змогу взаємодіяти з рештою сторінки, поки вікно не буде закрите.

Є два обмеження, пов’язані з усіма методами вище:

1. Точне розташування модального вікна визначається браузером. Зазвичай це в центрі.
2. Точний вигляд вікна також залежить від браузера. Ми не можемо його змінити.

Це ціна простоти. Є способи показувати приємніші вікна та збагатити взаємодію з відвідувачем, але якщо “навороти” не мають значення, то ці методи працюють дуже добре.

Завдання

Проста сторінка

важливість: 4

Створіть вебсторінку, яка запитує ім’я та виводить його.

Запустити демонстрацію

До рішення

Перетворення типу

Здебільшого оператори та функції автоматично перетворюють значення, які їм надаються, на потрібний тип.

Наприклад, `alert` автоматично перетворює будь-яке значення в рядок, щоби показати його. Математичні операції перетворюють значення на числа.

Є також випадки, коли нам необхідно явно перетворити значення на очікуваний тип.

> **Поки що не говоримо про об'єкти**
>
> У цьому розділі ми не будемо охоплювати об'єкти. Поки що ми поговоримо тільки про примітиви.
>
> Пізніше, після ознайомлення з об'єктами, ми розглянемо їхнє перетворення в розділі Перетворення об'єктів в примітиви.

Перетворення на рядок

Перетворення на рядок відбувається, коли нам потрібне значення у формі рядка.

Наприклад, `alert(value)` робить це, щоби показати значення.

Також ми можемо викликати функцію `String(value)` для перетворення значення в рядок:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // тепер value - це рядок "true"
alert(typeof value); // string
```

Перетворення рядків здебільшого очевидне. `false` стає `"false"`, `null` стає `"null"` тощо.

Перетворення на число

Перетворення на числа відбувається в математичних функціях і виразах автоматично.

Наприклад, коли ділення `/` застосовується до не-чисел:

```
alert( "6" / "2" ); // 3, рядки перетворюються на числа
```

Ми можемо використовувати функцію `Number(value)` для явного перетворення `value` на число:

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // стає числом 123
```

```
alert(typeof num); // number
```

Явне перетворення зазвичай потрібно, коли ми читаємо значення з джерела на основі рядка, подібно текстовій формі, але очікуємо, що буде введено число.

Якщо рядок не є дійсним числом, результатом такого перетворення є `NaN`. Наприклад:

```
let age = Number("довільний рядок замість числа");

alert(age); // NaN, помилка перетворення
```

Правила перетворення на числа:

Значення	Результат
`undefined`	`NaN`
`null`	`0`
`true та false`	`1` та `0`
`string`	Пробіли на початку та з кінця видаляються. Якщо рядок, що залишився в результаті, порожній, то результатом є `0`. В іншому випадку число “читається” з рядка. Помилка дає `NaN`.

Приклади:

```
alert( Number("   123   ") ); // 123
alert( Number("123z") );      // NaN (помилка читання числа на місці символу "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

Зверніть увагу, що `null` та `undefined` тут поводяться по-різному: `null` стає нулем, а `undefined` стає `NaN`.

Більшість математичних операторів також виконують такі перетворення. Ми розглянемо їх в наступному розділі.

Перетворення на булевий тип

Булеве перетворення є найпростішим.

Воно відбувається в логічних операціях (пізніше ми познайомимося з умовними перевірками та іншими подібними конструкціями), але також може бути виконане явно за допомогою виклику `Boolean(value)`.

Правила перетворення:

- Значення, які інтуїтивно “порожні”, такі як `0`, порожній рядок, `null`, `undefined` та `NaN`, стають `false`.
- Інші значення стають `true`.

Наприклад:

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false

alert( Boolean("вітаю") ); // true
alert( Boolean("") ); // false
```

Зверніть увагу: рядок із нулем `"0"` є `true`

Деякі мови (а саме PHP) розглядають `"0"` як `false`. Але у JavaScript непустий рядок завжди `true`.

```
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // пробіли, також true (будь-які непусті рядки є true)
```

Підсумки

Три найпоширеніші перетворення типів — це перетворення на рядок, на число та на булевий тип.

`Перетворення на рядок` – Відбувається, коли ми щось виводимо. Може бути виконане за допомогою `String(value)`. Перетворення на рядок звичайно очевидне для примітивних значень.

`Перетворення на число` – Відбувається в математичних операціях. Може бути виконане з `Number(value)`.

Перетворення дотримується правил:

Значення	Результат
`undefined`	`NaN`
`null`	`0`
`true / false`	`1 / 0`
`string`	Рядок читається “як є”, пробіли з обох сторін ігноруються. Пустий рядок стає `0`. Помилка дає `NaN`.

`Перетворення на булевий тип` – Відбувається в логічних операціях. Може виконуватися за допомогою `Boolean(value)`.

Дотримується правил:

Значення	Результат
`0`, `null`, `undefined`, `NaN`, `""`	`false`
будь-які інші значення	`true`

Більшість із цих правил легко зрозуміти й запам’ятати. Примітними винятками, де люди зазвичай роблять помилки, є:

- `undefined` є `NaN` як число, а не `0`.

- `"0"` і рядки, що мають тільки пробіли, такі як `" "`, є `true` як булеві значення.

Об'єкти тут не охоплені. Ми повернемося до них пізніше в розділі Перетворення об'єктів в примітиви, який присвячений виключно об'єктам, після того, як ми дізнаємося про більш базові речі в JavaScript.

Базові оператори, математика

Зі шкільної програми ми знаємо багато арифметичних операцій, таких як додавання `+`, множення `*`, віднімання `-` тощо.

У цьому розділі ми почнемо з простих операторів, потім зосередимося на специфічних для JavaScript аспектах, які не охоплені шкільною арифметикою.

Терміни: "унарний", "бінарний", "операнд"

Перш ніж ми почнемо, давайте розберемо певну загальну термінологію.

- *Операнд* – це те, до чого застосовуються оператори. Наприклад, у множенні `5 * 2` є два операнди: лівий операнд `5` і правий операнд `2`. Іноді їх називають "аргументами", а не "операндами".
- Оператор є *унарним*, якщо він має один операнд. Наприклад, унарне заперечення `-` змінює знак числа:

```
let x = 1;

x = -x;
alert( x ); // -1, було застосоване унарне заперечення
```

- Оператор є *бінарним*, якщо він має два операнди. Наприклад, оператор мінус можна використовувати і у бінарній формі:

```
let x = 1, y = 3;
alert( y - x ); // 2, бінарний мінус віднімає значення
```

  Формально, у прикладах вище ми маємо два різні оператори, які позначаються однаковим символом: оператор заперечення – унарний оператор, який змінює знак числа, та оператор віднімання – бінарний оператор, який віднімає одне число від іншого.

Математика

JavaScript підтримує такі математичні операції:

- Додавання `+`,
- Віднімання `-`,
- Множення `*`,
- Ділення `/`,
- Остача від ділення `%`,

- Піднесення до степеня `**`.

Перші чотири операції зрозумілі, а от про `%` та `**` потрібно сказати декілька слів.

Остача від ділення %

Оператор остачі `%`, попри свій зовнішній вигляд, не пов'язаний із відсотками.

Результатом `a % b` є остача цілочислового ділення `a` на `b`.

Наприклад:

```
alert( 5 % 2 ); // 1 – остача від ділення 5 на 2
alert( 8 % 3 ); // 2 – остача від ділення 8 на 3
```

Піднесення до степеня **

Оператор піднесення до степеня `a ** b` множить `a` саме на себе `b` разів.

У школі ми записуємо це як a^b.

Наприклад:

```
alert( 2 ** 2 ); // 2² = 4
alert( 2 ** 3 ); // 2³ = 8
alert( 2 ** 4 ); // 2⁴ = 16
```

Так само як у математиці, оператор піднесення також можна використовувати для дробових чисел.

Наприклад, квадратний корінь це піднесення до степеня ½:

```
alert( 4 ** (1/2) ); // 2 (степінь 1/2 – це теж саме, що квадратний корінь)
alert( 8 ** (1/3) ); // 2 (степінь 1/3 – це теж саме, що кубічний корінь)
```

Об'єднання рядків через бінарний +

Розглянемо особливості операторів JavaScript, які виходять за межі шкільної арифметики.

Зазвичай оператор плюс `+` додає числа.

Але якщо бінарний `+` застосовується до рядків, він об'єднує їх:

```
let s = 'мій_' + 'рядок';
alert(s); // мій_рядок
```

Зверніть увагу, якщо будь-який з операндів є рядком, тоді інший також перетворюється на рядок.

Наприклад:

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

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

Бачите, не має значення, чи перший операнд – рядок, чи другий.

Ось складніший приклад:

```
alert(2 + 2 + '1' ); // "41", а не "221"
```

Тут оператори виконуються один за одним. Перший `+` додає два числа, тому він поверне `4`; а наступний оператор `+` вже додасть (об'єднає) попередній результат із рядком `1`. У підсумку ми отримаємо рядок `'41'` (`4 + '1'`).

```
alert('1' + 2 + 2); // "122", а не "14"
```

У цьому прикладі перший операнд – рядок, тому компілятор також опрацьовує інші два операнди як рядки. Операнд `2` приєднується (конкатенується) до `'1'`, тому в результаті буде `'1' + 2 = "12"`, а потім — `"12" + 2 = "122"`.

Лише бінарний `+` працює з рядками так. Інші арифметичні оператори працюють тільки з числами й завжди перетворюють свої операнди на числа.

Ось приклад, як працює віднімання й ділення:

```
alert( 6 - '2' ); // 4, '2' перетворюється на число
alert( '6' / '2' ); // 3, обидва операнди перетворюються на числа
```

Числове перетворення, унарний +

У оператора плюс `+` є дві форми: бінарна, яку ми використовували вище, та унарна.

Унарний плюс або, іншими словами, оператор плюс `+`, застосований до одного операнда, нічого не зробить, якщо операнд є числом. Але якщо операнд не є числом, унарний плюс перетворить його на число.

Наприклад:

```
// Нема ніякого впливу на числа
let x = 1;
alert( +x ); // 1

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

// Перетворює нечислові значення
alert( +true ); // 1
alert( +"" );   // 0
```

Він насправді працює як і `Number(...)`, але має коротший вигляд.

Необхідність перетворення рядків на числа виникає дуже часто. Наприклад, якщо ми отримуємо значення з полів HTML форми, вони зазвичай є рядками. Що робити, якщо ми хочемо їх підсумувати?

Бінарний плюс додав би їх як рядки:

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

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

Якщо ми хочемо використовувати їх як числа, нам потрібно конвертувати, а потім підсумувати їх:

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

// обидва значення перетворюються на числа перед застосуванням бінарного плюса
alert( +apples + +oranges ); // 5

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

З погляду математика надмірні плюси можуть здатися дивними. Але з погляду програміста тут немає нічого особливого: спочатку застосовуються унарні плюси, вони перетворюють рядки на числа, а потім бінарний плюс підсумовує їх.

Чому унарні плюси застосовуються до значень перед бінарними плюсами? Як ми побачимо далі, це пов'язано з їхнім *вищим пріоритетом*.

Пріоритет оператора

Якщо вираз має більше одного оператора, порядок виконання визначається їхнім *пріоритетом*, або, іншими словами, типовим порядком першості операторів.

Зі школи ми всі знаємо, що множення у виразі `1 + 2 * 2` має бути обчислене перед додаванням. Саме це і є пріоритетом. Кажуть, що множення має *вищий пріоритет*, ніж додавання.

Дужки перевизначають будь-який пріоритет, тому, якщо ми не задоволені типовим пріоритетом, ми можемо використовувати дужки, щоби змінити його. Наприклад: `(1 + 2) * 2`.

У JavaScript є багато операторів. Кожен оператор має відповідний номер пріоритету. Першим виконується той оператор, який має найбільший номер пріоритету. Якщо пріоритет є однаковим, порядок виконання — зліва направо.

Ось витяг із таблиці пріоритетів (вам не потрібно її запам'ятовувати, але зверніть увагу, що унарні оператори мають вищий пріоритет за відповідні бінарні):

Пріоритет	Ім'я	Знак
...	...	...

Пріоритет	Ім'я	Знак
15	унарний плюс	`+`
15	унарний мінус	`-`
14	піднесення до степеня	`**`
13	множення	`*`
13	ділення	`/`
12	додавання	`+`
12	віднімання	`-`
...	...	...
2	присвоєння	`=`
...	...	...

Як ми бачимо, "унарний плюс" має пріоритет `15`, що вище за `12` – пріоритет "додавання" (бінарний плюс). Саме тому, у виразі `"+apples + +oranges"`, унарні плюси виконуються перед додаванням (бінарним плюсом).

Присвоєння

Зазначимо, що присвоєння `=` також є оператором. Воно є у таблиці з пріоритетами й має дуже низький пріоритет `2`.

Тому, коли ми присвоюємо значення змінній, наприклад, `x = 2 * 2 + 1`, спочатку виконуються обчислення, а потім виконується присвоєння `=` зі збереженням результату в `x`.

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

alert( x ); // 5
```

Присвоєння = повертає результат

Той факт, що `=` є оператором, а не "магічною" конструкцією мови, має цікаве значення.

Усі оператори в JavaScript повертають значення. Це очевидно для `+` та `-`, але це також правдиво для `=`.

Виклик `x = значення` записує `значення` у `x`, *а потім повертає його*.

Ось демонстрація, яка використовує присвоєння як частину складнішого виразу:

```
let a = 1;
let b = 2;

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

alert( a ); // 3
alert( c ); // 0
```

У наведеному вище прикладі результат виразу `(a = b + 1)` є значенням, яке присвоювалося змінній `a` (тобто `3`). Потім воно використовується для подальших обчислень.

Чудернацький код, чи не так? Ми маємо розуміти, як це працює, бо іноді ми бачимо подібне в бібліотеках JavaScript.

Однак, будь ласка, не пишіть свій код так. Ці трюки, безумовно, не роблять код більш зрозумілим або читабельним.

Ланцюгові присвоєння

Іншою цікавою особливістю є здатність ланцюгового присвоєння:

```
let a, b, c;

a = b = c = 2 + 2;

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

Ланцюгове присвоєння виконується справа наліво. Спочатку обчислюється найправіший вираз `2 + 2`, а потім результат присвоюється змінним ліворуч: `c`, `b` та `a`. Зрештою всі змінні мають спільне значення.

Знову таки, щоби покращити читабельність коду, краще розділяти подібні конструкції на декілька рядків:

```
c = 2 + 2;
b = c;
a = c;
```

Так легше прочитати, особливо коли швидко переглядати код.

Оператор “модифікувати та присвоїти”

Часто нам потрібно застосувати оператор до змінної й зберегти новий результат у ту ж саму змінну.

Наприклад:

```
let n = 2;
n = n + 5;
n = n * 2;
```

Цей запис можна скоротити за допомогою операторів `+=` та `*=`:

```
let n = 2;
n += 5; // тепер n = 7 (те ж саме, що n = n + 5)
n *= 2; // тепер n = 14 (те ж саме, що n = n * 2)
```

```
alert( n ); // 14
```

Короткі оператори “модифікувати та присвоїти” є для всіх арифметичних та побітових операторів: `/=`, `-=` тощо.

Ці оператори мають такий же пріоритет, як і звичайне присвоєння, тому вони виконуються після більшості інших обчислень:

```
let n = 2;

n *= 3 + 5;

alert( n ); // 16  (права частина обчислюється першою, так само, як n *= 8)
```

Інкремент/декремент

Збільшення або зменшення на одиницю є однією з найпоширеніших числових операцій.

Тому для цього є спеціальні оператори:

- **Інкремент** `++` збільшує змінну на 1:

  ```
  let counter = 2;
  counter++;        // працює так само, як counter = counter + 1, але запис коротше
  alert( counter ); // 3
  ```

- **Декремент** `--` зменшує змінну на 1:

  ```
  let counter = 2;
  counter--;        // працює так само, як counter = counter - 1, але запис коротше
  alert( counter ); // 1
  ```

> **Важливо:**
>
> Інкремент/декремент можуть застосовуватися лише до змінних. Спроба використати їх із значенням, як от `5++`, призведе до помилки.

Оператори `++` та `--` можуть розташовуватися до або після змінної.

- Коли оператор йде за змінною, він у “постфіксній формі”: `counter++`.
- “Префіксна форма” – це коли оператор йде попереду змінної: `++counter`.

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

Чи є різниця? Так, але ми можемо побачити її тільки використавши значення, яке повертають `++/--`.

Розберімось. Як нам відомо, всі оператори повертають значення. Інкремент/декремент не є винятком. Префіксна форма повертає нове значення, тоді як постфіксна форма повертає

старе значення (до збільшення/зменшення).

Щоби побачити різницю, наведемо приклад:

```
let counter = 1;
let a = ++counter; // (*)

alert(a); // 2
```

У рядку `(*)`, *префіксна* форма `++counter` збільшує `counter` та повертає нове значення, `2`. Отже, `alert` показує `2`.

Тепер скористаємося постфіксною формою:

```
let counter = 1;
let a = counter++; // (*) змінили ++counter на counter++

alert(a); // 1
```

У рядку `(*)`, *постфіксна* форма `counter++` також збільшує `counter`, але повертає *старе* значення (до інкременту). Отже, `alert` показує `1`.

Підсумки:

- Якщо результат збільшення/зменшення не використовується, немає ніякої різниці, яку форму використовувати:

  ```
  let counter = 0;
  counter++;
  ++counter;
  alert( counter ); // 2, у рядках вище робиться одне і те ж саме
  ```

- Якщо ми хочемо збільшити значення *та* негайно використати результат оператора, нам потрібна префіксна форма:

  ```
  let counter = 0;
  alert( ++counter ); // 1
  ```

- Якщо ми хочемо збільшити значення, але використати його попереднє значення, нам потрібна постфіксна форма:

  ```
  let counter = 0;
  alert( counter++ ); // 0
  ```

Інкремент/декремент серед інших операторів

Оператори `++/--` також можуть використовуватися всередині виразів. Їхній пріоритет вищий за більшість інших арифметичних операцій.

Наприклад:

```
let counter = 1;
alert( 2 * ++counter ); // 4
```

Порівняйте з:

```
let counter = 1;
alert( 2 * counter++ ); // 2, тому що counter++ повертає "старе" значення
```

Хоча з технічного погляду це допустимо, такий запис робить код менш читабельним. Коли один рядок робить кілька речей – це не добре.

Під час читання коду швидке “вертикальне” сканування оком може легко пропустити щось подібне до `counter++`, і не буде очевидним, що змінна була збільшена.

Ми рекомендуємо стиль “одна лінія – одна дія”:

```
let counter = 1;
alert( 2 * counter );
counter++;
```

Побітові оператори

Побітові оператори розглядають аргументи як 32-бітні цілі числа та працюють на рівні їхнього двійкового представлення.

Ці оператори не є специфічними для JavaScript. Вони підтримуються у більшості мов програмування.

Список операторів:

- AND(і) (`&`)
- OR(або) (`|`)
- XOR(побітове виключне або) (`^`)
- NOT(ні) (`~`)
- LEFT SHIFT(зсув ліворуч) (`<<`)
- RIGHT SHIFT(зсув праворуч) (`>>`)
- ZERO-FILL RIGHT SHIFT(зсув праворуч із заповненням нулями) (`>>>`)

Ці оператори використовуються тоді, коли нам потрібно “возитися” з числами на дуже низькому (побітовому) рівні (тобто – вкрай рідко). Найближчим часом такі оператори нам не знадобляться, оскільки у веброзробці вони майже не використовуються. Проте в таких