```
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

галузях, як криптографія, вони можуть бути дуже корисними. Ви можете прочитати розділ Bitwise Operators на MDN, якщо виникне потреба.

Кома

Оператор “кома” (`,`) незвичайний і застосовується дуже рідко. Іноді цей оператор використовують для написання коротшого коду, тому нам потрібно знати його, щоби розуміти, що відбувається.

Оператор кома дає змогу обчислити кілька виразів, розділивши їх комою `,`. Кожен із них обчислюється, але повертається тільки результат останнього.

Наприклад:

```
let a = (1 + 2, 3 + 4);

alert( a ); // 7 (результат обчислення 3 + 4)
```

Тут обчислюється перший вираз `1 + 2` і його результат викидається. Потім обчислюється `3 + 4` і повертається як результат.

> **Кома має дуже низький пріоритет**
>
> Зверніть увагу, що оператор “кома” має дуже низький пріоритет, нижчий за `=`, тому дужки є важливими в наведеному вище прикладі.
>
> Без дужок, у виразі `a = 1 + 2, 3 + 4` спочатку обчислюються оператори `+`, підсумовуючи числа у `a = 3, 7`; потім оператор присвоєння `=` присвоює `a = 3`, а решта (число `7` після коми) ігнорується. Це як записати вираз `(a = 1 + 2), 3 + 4`.

Чому нам потрібен оператор, що викидає все, окрім останнього виразу?

Іноді його використовують у складніших конструкціях, щоби помістити кілька дій в один рядок.

Наприклад:

```
// три операції в одному рядку
for (a = 1, b = 3, c = a * b; a < 10; a++) {
  ...
}
```

Такі трюки використовуються в багатьох фреймворках JavaScript. Саме тому ми їх згадуємо. Але зазвичай вони не покращують читабельність коду, тому ми маємо добре подумати перед їх використанням.

Завдання

Challenge 1 **Постфіксна та префіксна форми**

Які кінцеві значення всіх змінних `a`, `b`, `c` та `d` після виконання коду нижче?

```
let a = 1, b = 1;

let c = ++a; // ?
let d = b++; // ?
```

Challenge 2 **Результат присвоєння**

Які значення мають `a` та `x` після виконання коду нижче?

```
let a = 2;

let x = 1 + (a *= 2);
```

Challenge 3 **Перетворення типу**

Які результати цих виразів?

```
"" + 1 + 0
"" - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5
"4" - 2
"4px" - 2
"  -9  " + 5
"  -9  " - 5
null + 1
undefined + 1
" \t \n" - 2
```

Добре подумайте, запишіть, а потім порівняйте з відповіддю.

Challenge 4 **Виправте додавання**

Нижче наведено код, що просить користувача ввести два числа і відображає їхню суму.

Він працює неправильно. Код у прикладі виводить `12` (для початкових значень у полях вводу).

У чому помилка? Виправте її. Результат має бути `3`.

```
let a = prompt("Перше число?", 1);
let b = prompt("Друге число?", 2);

alert(a + b); // 12
```