Конструкція "switch"

Конструкція `switch` може замінити кілька `if`.

Вона дає можливість більш наочного способу порівняння значення відразу з кількома варіантами.

Синтаксис

Конструкція `switch` має один або більше `case` блоків та необов'язковий блок `default`.

Вона виглядає так:

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]

  case 'value2':  // if (x === 'value2')
    ...
    [break]

  default:
    ...
    [break]
}
```

- Значення змінної `x` перевіряється на строгу рівність (===) значенню із першого блоку `case` (яке дорівнює `value1`), потім значенню із другого блоку (`value2`) і так далі.
- Якщо строго рівне значення знайдено, то `switch` починає виконання коду із відповідного `case` до найближчого `break` або до кінця всієї конструкції `switch`.
- Якщо жодне `case`-значення не збігається – виконується код із блоку `default` (якщо він присутній).

Приклад роботи

Приклад використання `switch` (код який буде виконаний виділено):

```
let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Замало' );
    break;
  case 4:
    alert( 'Точнісінько!' );
    break;
  case 5:
    alert( 'Забагато' );
    break;
  default:
    alert( 'Я не знаю таких значень' );
}
```

Тут `switch` починає порівнювати `a` з першим варіантом із `case`, який дорівнює `3`. Це не відповідає `a`.

Потім з другим, який дорівнює `4`. Цей варіант відповідає `a`, таким чином буде виконано код з `case 4` до найближчого `break`.

Якщо `break` відсутній, то буде продовжено виконання коду по наступним блокам `case` без перевірок.

Приклад без `break`:

```
let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Замало' );
  case 4:
    alert( 'Точнісінько!' );
  case 5:
    alert( 'Забагато' );
  default:
    alert( 'Я не знаю таких значень' );
}
```

В прикладі вище ми бачимо послідовне виконання трьох `alert`:

```
alert( 'Точнісінько!' );
alert( 'Забагато' );
alert( 'Я не знаю таких значень' );
```

Any expression can be a `switch/case` argument

Обидва `switch` та `case` допускають будь-який вираз в якості аргументу.

Наприклад:

```
let a = "1";
let b = 0;

switch (+a) {
  case b + 1:
    alert("виконано це, бо +a це 1, що строго дорівнює b + 1");
    break;

  default:
    alert("це не буде виконано");
}
```

Тут значення виразу `+a` буде `1`, що збігається з значенням виразу `b + 1` із блоку `case`, таким чином код із цього блоку буде виконано.

Групування “case”

Кілька варіантів блоку `case`, які використовують однаковий код, можуть бути згруповані.

Наприклад, якщо ми бажаємо виконати один і той самий код для `case 3` та `case 5`:

```
let a = 3;

switch (a) {
  case 4:
    alert('Вірно!');
    break;

  case 3: // (*) групуємо два блоки `case`
  case 5:
    alert('Невірно!');
    alert("Можливо вам варто відвідати урок математики?");
    break;

  default:
    alert('Результат виглядає дивно. Дійсно.');
}
```

Тепер обидва варіанти `3` та `5` виводять однакове повідомлення.

Можливість групування блоків `case` – це побічний ефект того, як `switch/case` працює без `break`. Тут виконання коду `case 3` починається з рядка `(*)` та проходить через `case 5`, бо немає `break`.

Тип має значення

Необхідно наголосити, що перевірка відповідності є завжди строгою. Значення повинні бути однакового типу аби вони збігалися.

Наприклад, розглянемо наступний код:

```
let arg = prompt("Введіть значення?");
switch (arg) {
  case '0':
  case '1':
    alert( 'Один або нуль' );
    break;

  case '2':
    alert( 'Два' );
    break;

  case 3:
    alert( 'Ніколи не буде виконано!' );
    break;
  default:
    alert( 'Невідоме значення' );
}
```

1. Для `0` та `1` буде виконано перший `alert`.
2. Для `2` – другий `alert`.
3. Але для `3`: результат виконання `prompt` є строкове значення `"3"`, яке строго не дорівнює `===` числу `3`. Таким чином ми маємо “мертвий код” в блоці `case 3`! Буде виконано код блоку `default`.

Завдання

Перепишіть конструкцію "switch" в аналогічну з використанням "if"

Напишіть код з використанням `if..else`, що відповідає наступній конструкції `switch`:

```
switch (browser) {
  case 'Edge':
    alert( "You've got the Edge!" );
    break;

  case 'Chrome':
  case 'Firefox':
  case 'Safari':
  case 'Opera':
    alert( 'Ми підтримуємо і ці браузери' );
    break;

  default:
    alert( 'Маємо надію, що ця сторінка виглядає добре!' );
}
```

Перепишіть умови "if" в конструкцію "switch"

Перепишіть код нижче використовуючи одну конструкцію `switch`:

```
let a = +prompt('a?', '');

if (a == 0) {
  alert( 0 );
}
if (a == 1) {
  alert( 1 );
}

if (a == 2 || a == 3) {
  alert( '2,3' );
}
```

Функції

Досить часто нам потрібно виконати однакову дію в декількох місцях програми.

Наприклад, нам треба показати якесь повідомлення, коли користувач входить або виходить з системи і може ще десь.

Функції — це головні “будівельні блоки” програми. Вони дозволяють робити однакові дії багато разів без повторення коду.

Ми вже стикались з такими вбудованими функціями, як-от `alert(message)`, `prompt(message, default)` та `confirm(question)`. Але ми теж можемо створювати свої функції.

Оголошення (декларація) функцій

Щоб створити функцію нам треба її *оголосити*.

Це виглядає ось так:

```
function showMessage() {
  alert('Всім привіт!');
}
```

Спочатку ми пишемо `function` — це ключове слово (keyword), яке дає зрозуміти комп’ютеру, що далі буде оголошення функції. Потім — *назву функції* і список її *параметрів* в дужках (розділені комою). Якщо параметрів немає, ми залишаємо *пусті дужки*. І нарешті, код функції, який також називають *тілом функції* між фігурними дужками.

```

```

```
function name(parameter1, parameter2, ... parameterN) {
  ...тіло функції...
}
```

Нашу нову функцію можна викликати, написавши її ім'я і дужки: `showMessage()`.

Наприклад:

```
function showMessage() {
  alert( 'Шановні друзі!' );
}

showMessage();
showMessage();
```

Виклик `showMessage()` виконує код із тіла функції. В цьому випадку, ми побачимо повідомлення двічі.

Цей приклад яскраво демонструє одну з найголовніших цілей функції – уникнення повторення коду.

Якщо нам потрібно змінити повідомлення, достатньо змінити тіло функції, яке виводить це повідомлення.

Локальні змінні

Змінна, яка оголошена в функції доступна лише в тілі цієї функції.

Наприклад:

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

  alert( message );
}

showMessage(); // Привіт, я JavaScript!

alert( message ); // <-- Помилка! Змінна недоступна поза функцією
```

Зовнішні змінні

Функція може використовувати зовнішні змінні, наприклад:

```
let userName = 'Іван';

function showMessage() {
  let message = 'Привіт, ' + userName;
  alert(message);
}

showMessage(); // Привіт, Іван
```

Функція має повний доступ до зовнішньої змінної. Вона теж може її змінювати.

Наприклад:

```
let userName = 'Іван';

function showMessage() {
  userName = "Богдан"; // (1) змінено зовнішню змінну

  let message = 'Здоровенькі були, ' + userName;
  alert(message);
}

alert( userName ); // Іван перед викликом функції showMessage

showMessage();

alert( userName ); // Богдан, значення було змінено після виклику функції showMessage
```

Зовнішня змінна використовується тоді, коли немає локальної.

Якщо всередині функції є змінна з таким самим ім'ям, то вона *перекриває* зовнішню. Наприклад, наступний код використовує локальну змінну `userName`. Зовнішня ігнорується.

```
let userName = 'Іван'; // оголошення зовнішньої змінної

function showMessage() {
  let userName = "Богдан"; // оголошення локальної змінної

  let message = 'Привіт, ' + userName; // Богдан
  alert(message);
}

// функція завжди віддасть перевагу локальним змінним
showMessage();

alert( userName ); // Іван, без змін, функція не змінила глобальну змінну
```

Глобальні змінні

Змінні, оголошені поза будь-якими функціями (такі як зовнішня змінна `userName` з коду вище), називаються *глобальні* змінні.

Глобальні змінні доступні в будь-якій функції (окрім випадків, коли глобальна змінна перекрита локальною).

Хорошою практикою вважається мінімізація використання глобальних змінних. У сучасному коді зазвичай є декілька або зовсім немає глобальних змінних. Більшість змінних знаходяться в межах функцій. Іноді буває корисно зберігати "загальні" дані (на рівні проєкту) в таких глобальних змінних.

Параметри

Ми можемо передати в функцію довільні дані використовуючи параметри.

В наступному прикладі, функція має два параметри: `from` і `text`.

```
function showMessage(from, text) { // параметри: from, text
  alert(from + ': ' + text);
}

showMessage('Анна', 'Привіт!'); // Анна: Привіт! (*)
showMessage('Анна', "Як справи?"); // Анна: Як справи? (**)
```

Під час виклику функції з цими параметрами, в рядках `(*)` та `(**)` відбувається копіювання значень параметрів в локальні змінні `from` та `text`. Ці змінні використовує функція.

Ось ще один приклад: маємо змінну `from`, яку передаємо в функцію. Зауважте: функція змінює значення `from`, проте ці зміни не видно назовні, тому що функція завжди отримує копію значення:

```
function showMessage(from, text) {

  from = '*' + from + '*'; // прикрашаємо "from"

  alert( from + ': ' + text );
}

let from = "Анна";

showMessage(from, "Привіт"); // *Анна*: Привіт

// значення "from" те саме, функція змінила локальну копію
alert( from ); // Анна
```

Коли значення передається як параметр функції, то його ще називають *аргумент*.

Кажучи “на хлопський розум”:

- Параметр – це змінна між дужками функції (використовується під час оголошення функції)
- Аргумент – це значення, передане в функцію під час її виклику (використовується під час виконання функції).

Ми оголошуємо функції, вказуючи їхні параметри, потім викликаємо їх, передаючи аргументи.

Дехто може сказати, що в прикладі вище "функцію `showMessage` оголошено з двома параметрами, потім викликано з двома аргументами: `from` і `"Привіт"`".

Типові значення

Якщо викликати функцію без аргументів, тоді відповідні значення стануть `undefined`.

Наприклад, функцію `showMessage(from, text)`, яку ми згадували вище, можна викликати з одним аргументом:

```
showMessage('Анна');
```

Помилки не виникне. Такий виклик виведе `"*Анна*: undefined"`. Оскільки значення для змінної `text` не задане, воно стане `undefined`.

Ми можемо задати так зване “типове” значення параметра, яке використовуватиметься, якщо не задати аргумент. Для цього потрібно написати значення через `=`:

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

showMessage("Анна"); // Анна: текст не задано
```

Тепер, якщо параметр `text` не задано, його значення стане `"текст не задано"`.

Тут `"текст не задано"` це рядок, проте це може бути складніший вираз, який обчислюється і присвоюється лише якщо параметр відсутній. Отож, такий варіант теж можливий:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() виконується лише якщо `text` не задано
  // результат виконання цієї функції присвоїться змінній `text`
}
```

Обчислення типових параметрів

В JavaScript, типовий параметр обчислюється кожного разу, коли викликається функція без відповідного параметру.

В прикладі вище, функція `anotherFunction()` не викличеться, якщо буде задано параметр `text`.

З іншого боку, вона буде викликатися кожного разу, коли `text` відсутній.

Типові параметри у старому JavaScript коді

Кілька років тому JavaScript не підтримував синтаксис типових значень для параметрів. Тому люди використовували інші способи їх визначення.

Нині ми можемо зустріти їх у старих скриптах.

Наприклад, явна перевірка на `undefined`:

```
function showMessage(from, text) {
  if (text === undefined) {
    text = 'текст повідомлення відсутній';
  }

  alert( from + ": " + text );
}
```

…Або за допомогою оператора `||`:

```
function showMessage(from, text) {
  // Якщо значення у змінній text дає false, призначається типове значення
  // це передбачає, що text == "" це те саме, що й його повна відсутність
  text = text || 'текст повідомлення відсутній';
  ...
}
```

Альтернативні типові параметри

Інколи виникає необхідність присвоїти типове значення для змінних під час виконання функції, а не під час її оголошення.

Під час виконання функції, ми можемо перевірити, чи параметр надано, порівнюючи його з `undefined`:

```
function showMessage(text) {
  // ...

  if (text === undefined) { // якщо параметр відсутній
    text = 'порожнє повідомлення';
  }

  alert(text);
}

showMessage(); // порожнє повідомлення
```

…Або ми можемо використати оператор `||`:

```
function showMessage(text) {
  // якщо text не задано (значення `undefined`) або `null`, тоді присвоїти рядок 'порожньо'
  text = text || 'порожньо';
  ...
}
```

Сучасні рушії JavaScript підтримують оператор null-злиття `??`. Його краще використовувати, коли “майже false” значення, типу `0`, мають вважатися за “нормальні”:

```
function showCount(count) {
  // якщо count має значення undefined чи null, показати "невідомо"
  alert(count ?? "невідомо");
}

showCount(0); // 0
showCount(null); // невідомо
showCount(); // невідомо
```

Повернення значення

В якості результату, функція може повертати назад значення в код, який викликав цю функцію.

Найпростіший приклад — функція, яка сумує два значення:

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

let result = sum(1, 2);
alert( result ); // 3
```

Директива `return` може бути в будь-якому місці функції. Коли виконання досягає цієї директиви, функція зупиняється, і в код, який викликав цю функцію, повертається значення (в прикладі вище, це значення присвоюється змінній `result`).

В одній функції може бути декілька директив `return`. Наприклад:

```
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('У вас є дозвіл ваших батьків?');
  }
}

let age = prompt('Скільки вам років?', 18);

if ( checkAge(age) ) {
  alert( 'Доступ надано' );
} else {
  alert( 'У доступі відмовлено' );
}
```

Можна використовувати `return` без значення. Це призведе до негайного виходу з функції.

Наприклад:

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Показуємо фільм" ); // (*)
  // ...
}
```

В коді вище, якщо `checkAge(age)` поверне `false`, тоді функція `showMovie` не дійде до виконання `alert`.

Функція з порожнім `return`, або без `return` повертає `undefined`

Якщо функція не повертає значення, тоді “повернене” значення буде `undefined`:

```
function doNothing() { /* порожньо */ }

alert( doNothing() === undefined ); // true
```

Порожній `return` це те саме, що `return undefined`:

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

Ніколи не додавайте новий рядок між `return` і значенням

Іноді кортить перенести довгий вираз після `return` на новий рядок, ось так:

```
return
 ('деякий' + 'довгий' + 'вираз' + 'або' + 'що' * f(a) + f(b))
```

Це не спрацює, тому що JavaScript вважатиме новий рядок після `return` за крапку з комою. Це працюватиме ось так:

```
return;
 ('деякий' + 'довгий' + 'вираз' + 'або' + 'що' * f(a) + f(b))
```

Тобто, повернеться порожній результат.

Якщо ми хочемо повернути довгий вираз, який займе декілька рядків, ми повинні писати його на одному рядку з `return`. Або обгорнути його в дужки. Ось так:

```
return (
  'деякий' + 'довгий' + 'вираз'
  + 'або' +
  'що' * f(a) + f(b)
  )
```

Такий варіант працюватиме так, як ми задумали.

Найменування функції

Функції виконують дії. Тому в їхніх іменах зазвичай використовують дієслова. Ім'я повинне бути лаконічним, повинне якнайточніше описувати, що робить функція, щоб кожен хто читає код зміг зрозуміти, що саме робить функція.

Поширена практика розпочинати ім'я функції зі словесного префіксу, який описує дію. В команді має бути домовленість щодо значення префіксів.

Наприклад, функції, які починаються з префіксу `"show"` зазвичай щось показують.

Функції, які починаються з …

- `"get…"` – повертають значення,
- `"calc…"` – щось обчислюють,
- `"create…"` – щось створюють,
- `"check…"` – щось перевіряють і повертають булеве значення.

Ось приклади таких імен:

```
showMessage(..)     // показує повідомлення
getAge(..)          // повертає вік (якось його отримує або обчислює)
calcSum(..)         // обчислює суму і повертає результат
```

```
createForm(..)      // створює форму (і зазвичай її повертає)
checkPermission(..) // перевіряє доступ, повертає true/false
```

Якщо є префікси, погляд на ім'я функції дає зрозуміти, яку роботу вона виконує і яке значення повертає.

> **Одна функція – одна дія**
>
> Функція повинна робити саме те, що написано в її імені, не більше.
>
> Дві незалежні дії зазвичай заслуговують двох функцій, навіть якщо вони зазвичай викликаються разом (у цьому випадку ми можемо створити 3-ю функцію, яка викликає ці дві).
>
> Ось декілька прикладів, які порушують це правило:
>
> - `getAge` – функція викликає `alert` з віком (а повинна лише отримувати вік).
> - `createForm` – функція змінює документ, додаючи форму до неї (а повинна лише створити форму і її вернути).
> - `checkPermission` – функція відображає повідомлення `доступ надано/відхилено` (а повинна лише повертати результат `true/false`).
>
> Ці приклади передбачають загальне значення префіксів. Ви та ваша команда можете вільно домовлятися про інші значення, але зазвичай вони не сильно відрізняються. У будь-якому випадку ви повинні чітко розуміти, що означає префікс, що може робити префіксна функція, а що ні. Усі функції з однаковими префіксами повинні підкорятися правилам. І команда повинна ділитися знаннями.

> **Дуже короткі імена функцій**
>
> Функції, які використовуються *дуже часто* деколи мають дуже короткі імена.
>
> Наприклад, фреймворк jQuery оголошує функцію знаком `$`. Бібліотека Lodash має вбудовану функцію, яка називається `_`.
>
> Це винятки. Загалом імена функцій повинні бути стислими та описовими.

Функції == Коментарі

Функції повинні бути короткими і робити щось одне. Якщо це щось велике, в цьому випадку доцільно розділити таку функцію на декілька менших. Іноді дотримуватися цього правила досить важко, але це, безумовно, хороша практика.

Невеликі функції не тільки полегшують перевірку та налагодження – саме їхнє існування виконує роль хороших коментарів, які покращують зрозумілість коду!

Ось для прикладу, порівняйте дві функції `showPrimes(n)`. Кожна з них виводить прості числа до `n`.

Перший варіант використовує мітку `nextPrime`:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {
```

```
    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert( i ); // просте число
  }
}
```

Другий варіант використовує додаткову функцію `isPrime(n)`, щоб перевіряти, чи число просте:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

    alert(i);  // просте число
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

Другий варіант легше зрозуміти, чи не так? Замість частини коду ми бачимо назву дії (`isPrime`). Іноді розробники називають такий код *самодокументованим*.

Отже, функції можна створювати, навіть якщо ми не маємо наміру повторно їх використовувати. Вони структурують код і роблять його читабельним та зрозумілим.

Підсумки

Оголошення функції виглядає ось так:

```
function ім'я(параметри, розділені, комою) {
  /* тіло, код функції */
}
```

- Значення, які передаються в функцію в якості параметрів, копіюються в локальні змінні.
- Функції мають доступ до зовнішніх змінних. Але це працює тільки зсередини назовні. Код поза функцією не має доступу до локальних змінних функції.
- Функція може повертати значення. Якщо цього не відбувається, результат буде `undefined`.

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

Завжди легше зрозуміти функцію, яка отримує параметри, працює з ними і повертає результат. На відмінну від функції, в якої немає параметрів, але яка змінює зовнішні змінні, що може призводити до побічних ефектів.

Найменування функцій:

- Ім'я функції повинне бути коротким і чітко відображати, що робить функція. Побачивши виклик функції в коді, ви повинні зразу зрозуміти, що функція робить, і що повертає.
- Функція – це дія, тому її ім'я зазвичай складається з дієслова.
- Є багато загальноприйнятих префіксів, такі як `create…`, `show…`, `get…`, `check…` тощо. Використовуйте їх щоб пояснити, що робить функція.

Функції – це основні будівельні блоки скриптів. Ми розглянули лише основи функцій в JavaScript, проте вже зараз цього достатньо, щоб почати їх створювати і використовувати. Це лише початок шляху. Ми будемо неодноразово повертатися до функцій і вивчатимо їх все глибше і глибше.

Завдання

Чи потрібен "else"?

Наступна функція повертає `true`, якщо параметр `age` більший за `18`.

Інакше вона запитує підтвердження через `confirm` і повертає його результат:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    // ...
    return confirm('Батьки дозволили?');
  }
}
```

Чи буде функція працювати по-іншому, якщо забрати `else`?

```
function checkAge(age) {
  if (age > 18) {
    return true;
  }
  // ...
  return confirm('Батьки дозволили?');
}
```

Чи є різниця в поведінці цих двох варіантів?

Перепишіть функцію, використовуючи '?' або '||'

Наступна функція повертає `true`, якщо параметр `age` більший за `18`.

Інакше вона запитує підтвердження через `confirm` і повертає його результат:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('Батьки дозволили?');
  }
}
```

Перепишіть функцію, щоб вона робила теж саме, але без `if` і в один рядок.

Зробіть два варіанти функції `checkAge`:

1. Використовуючи оператор `?`
2. Використовуючи оператор АБО `||`

Функція min(a, b)

Напишіть функцію `min(a, b)`, яка повертає менше з двох чисел `a` та `b`.

Наприклад:

```
min(2, 5) == 2
min(3, -1) == -1
min(1, 1) == 1
```

Функція pow(x, n)

Напишіть функцію `pow(x, n)`, яка повертає число `x`, піднесене до степеня `n`. Інакше кажучи, множить число `x` саме на себе `n` разів і повертає результат.

```
pow(3, 2) = 3 * 3 = 9
pow(3, 3) = 3 * 3 * 3 = 27
pow(1, 100) = 1 * 1 * ...* 1 = 1
```

Створіть сторінку, яка запитує `x` та `n`, а потім показує результат `pow(x, n)`.

P.S. В цій задачі функція повинна підтримувати лише натуральні значення `n`, тобто цілічисла, починаючи з `1`.