**Лекція 5. Рекурсія, стек. Іменовані функціональні вирази.** **Введення в методи і властивості.**

1. Рекурсія, стек.

2. Іменовані функціональні вирази.

3. Введення в методи і властивості.

1. У тілі функції можуть бути викликані інші функції для виконання підзадач.

Окремий випадок підвиклику - коли функція викликає сама себе. Це називається *рекурсією*.

Рекурсія використовується для ситуацій, коли виконання однієї складної задачі можна представити як якусь дію в сукупності з рішенням тієї ж задачі в більш простому варіанті.

Рекурсія - загальна тема програмування, яка не належить безпосередньо до JavaScript. Якщо ви програмували на інших мовах або вивчали програмування раніше у ВНЗ, то напевно вже знаєте, що це таке.

Ця глава призначена для читачів, які поки з цією темою незнайомі і хочуть краще розібратися в тому, як працюють функції.

**Ступінь pow (x, n) через рекурсію**

Як перший приклад використання рекурсивних викликів - розглянемо задачу зведення числа *x* в натуральну ступінь *n*.

Її можна представити як сукупність більш простого дії і більш простої задачі того ж типу ось так:

pow(x, n) = x \* pow(x, n - 1)

Тобто, xn = x \* xn-1.

Наприклад, обчислимо pow(2, 4), послідовно переходячи до більш простого завдання:

1. pow(2, 4) = 2 \* pow(2, 3)
2. pow(2, 3) = 2 \* pow(2, 2)
3. pow(2, 2) = 2 \* pow(2, 1)
4. pow(2, 1) = 2

На кроці 1 нам потрібно обчислити pow(2,3), тому ми робимо крок 2, далі нам потрібно pow(2,2), ми робимо крок 3, потім крок 4, і на ньому вже можна зупинитися, адже очевидно, що результат піднесення числа до степеня 1 - дорівнює самому числу.

Далі, маючи результат на кроці 4, він підставляється назад в крок 3, потім маємо pow(2,2) - підставляємо в крок 2 і на кроці 1 вже отримуємо результат.

Цей алгоритм на JavaScript:

function pow(x, n) {

if (n != 1) { // поки n != 1, зводити обчислення pow(x,n) до pow(x,n-1)

return x \* pow(x, n - 1);

} else {

return x;

}

}

alert( pow(2, 3) ); // 8

Кажуть, що «функція pow рекурсивно викликає сама себе » до n == 1.

Значення, на якому рекурсія закінчується, називають *базисом рекурсії*. В наведеному вище прикладі базисом є 1.

Загальна кількість вкладених викликів називають глибиною рекурсії. У випадку зі ступенем, всього буде n викликів.

Максимальна глибина рекурсії в браузерах обмежена, точно можна розраховувати на 10000 вкладених викликів, але деякі інтерпретатори допускають і більше.

Отже, рекурсію використовують, коли обчислення функції можна звести до її більш простого виклику, а його - ще до більш простого, і так далі, поки значення не стане очевидним.

**Контекст виконання, стек.**

Тепер ми подивимося, як працюють рекурсивні виклики. Для цього ми розглянемо, як взагалі працюють функції, що виконуються при виклику.

**У кожного виклику функції є свій «контекст виконання» (execution context).**

Контекст виконання - це службова інформація, яка відповідає поточному запуску функції. Вона включає в себе локальні змінні функції і конкретне місце в коді, на якому знаходиться інтерпретатор.

Наприклад, для виклику pow(2, 3) з прикладу вище буде створений контекст виконання, який буде зберігати змінні x = 2, n = 3. Ми схематично позначимо його так:

* Контекст: {x: 2, n: 3, рядок 1}

Далі функція pow починає виконуватися. Обчислюється вираз n != 1 - він дорівнює true, адже в поточному контексті n=3. Тому задіюється перша гілка if:

function pow(x, n) {

if (n != 1) { // поки n != 1 зводити обчислення pow(x,n) до pow(x,n-1)

return x \* pow(x, n - 1);

} else {

return x;

}

}

Щоб обчислити вираз x \* pow(x, n-1), потрібно провести запуск pow з новими аргументами.

**При будь-якому вкладеному виклику JavaScript запам'ятовує поточний контекст виконання в спеціальній внутрішній структурі даних - «стеку контекстів».**

Потім інтерпретатор приступає до виконання вкладеного виклику.

В даному випадку викликається та ж pow, проте це абсолютно неважливо. Для будь-яких функцій процес однаковий.

Для нового виклику створюється свій контекст виконання, і управління переходить в нього, а коли він завершений - старий контекст дістається з стека і виконання зовнішньої функції поновлюється.

Розберемо, що відбувається з контекстами більш детально, починаючи з виклику (\*):

function pow(x, n) {

if (n != 1) { // пока n!=1 зводити обчислення pow(..n) до pow(..n-1)

return x \* pow(x, n - 1);

} else {

return x;

}

}

alert( pow(2, 3) ); // (\*)

pow(2, 3)

Запускається функція pow, з аргументами x = 2, n = 3. Ці змінні зберігаються в контексті виконання, схематично зображеному нижче:

* Контекст: {x: 2, n: 3, рядок 1}

Виконання в цьому контексті триває, поки не зустріне вкладений виклик в рядку 3.

**pow(2, 2)**

У рядку 3 відбувається вкладений виклик pow з аргументами x=2, n=2. Поточний контекст зберігається в стеку, а для вкладеного виклику створюється новий контекст (виділено жирним нижче):

* Контекст: {x: 2, n: 3, рядок 3}
* **Контекст: {x: 2, n: 2, рядок 1}**

Звернемо увагу, що контекст включає в себе не тільки змінні, але і місце в коді, так що коли вкладений виклик завершиться - можна буде легко повернутися назад.

Слово «рядок» тут умовне, насправді, звичайно, запам’ятовано більш точне місце в ланцюжку команд.

**pow(2, 1)**

Знову вкладений виклик в рядку 3, на цей раз - з аргументами x=2, n=1. Створюється новий поточний контекст, попередній додається в стек:

* Контекст: {x: 2, n: 3, рядок 3}
* Контекст: {x: 2, n: 2, рядок 3}
* Контекст: {x: 2, n: 1, рядок 1}

На поточний момент в стеці вже два старих контексти.

Вихід з **pow(2, 1)**.

При виконанні pow(2, 1), на відміну від попередніх запусків, вираз n != 1 дорівнюватиме false, тому спрацює друга гілка if..else:

function pow(x, n) {

if (n != 1) {

return x \* pow(x, n - 1);

} else {

return x; // перша степінь числа рівна самому числу

}

}

Тут вкладених викликів немає, так що функція закінчує свою роботу, повертаючи 2. Поточний контекст більше не потрібен і видаляється з пам'яті, з стека відновлюється попередній:

* Контекст: {x: 2, n: 3, рядок 3}
* Контекст: {x: 2, n: 2, рядок 3}

Відновлюється обробка зовнішнього виклику pow (2, 2).

Вихід з **pow(2, 2)**.

... І тепер вже **pow(2, 2)** може закінчити свою роботу, повернувши 4. Відновлюється контекст попереднього виклику:

* Контекст: {x: 2, n: 3, рядок 3}

Відновлюється обробка зовнішнього виклику pow (2, 3).

Вихід з **pow(2, 3)**.

Самий зовнішній виклик закінчує свою роботу, його результат: pow(2, 3) = 8.

Глибина рекурсії в даному випадку склала: 3.

Як видно з ілюстрацій вище, глибина рекурсії дорівнює максимальному числу контекстів буде одночасно збережено в стеці.

Звернемо увагу на вимоги до пам'яті. Рекурсія призводить до зберігання всіх даних для незакінчених зовнішніх викликів в стеці, в даному випадку це призводить до того, що зведення в степінь n зберігає в пам'яті n різних контекстів.

Реалізація зведення в степінь через цикл набагато більш економна:

function pow(x, n) {

var result = x;

for (var i = 1; i < n; i++) {

result \*= x;

}

return result;

}

У такий функції pow буде один контекст, в якому будуть послідовно змінюватися значення i і result.

**Будь-яка рекурсія може бути перероблена в цикл. Як правило, варіант з циклом буде ефективніший.**

Але переробка рекурсії в цикл може бути нетривіальною, особливо коли в функції, в залежності від умов, використовуються різні рекурсивні підвиклики, коли розгалуження більш складне.

**Висновки**

Рекурсія - це коли функція викликає сама себе, як правило, з іншими аргументами.

Існує багато областей застосування рекурсивних викликів. Тут ми подивилися на один з них - рішення задачі шляхом зведення її до більш простої (з меншими аргументами), але також рекурсія використовується для роботи з «природно рекурсивними» структурами даних, такими як HTML-документи, для «глибокого» копіювання складних об'єктів.

Є й інші застосування, з якими ми зустрінемося в міру вивчення JavaScript.

**2. Іменовані функціональні вирази**

Спеціально для роботи з рекурсією в JavaScript існує особливе розширення функціональних виразів, яке називається «Named Function Expression» (скорочено NFE) або, українською, «іменований функціональний вираз».

**Named Function Expression**

Звичайний функціональний вираз:

var f = function(...) { /\* тіло функції \*/ };

Іменований з ім'ям sayHi:

var f = function sayHi(...) { /\* тіло функції \*/ };

Що ж це за ім'я, яке йде на додаток до f, і навіщо воно?

Ім'я функціонального вираження (sayHi) має особливий сенс. Воно є тільки зсередини самої функції (f).

Це обмеження видимості входить в стандарт JavaScript і підтримується всіма браузерами, крім IE8-.

Наприклад:

var f = function sayHi(name) {

alert( sayHi ); // зсередини функції - видно (виведе код функції)};

alert( sayHi ); // зовні - не видно (помилка: undefined variable 'sayHi')

Крім того, ім'я NFE не можна перезаписати:

var test = function sayHi(name) {

sayHi = "тест"; // спроба перезапису

alert( sayHi ); // function... (перезапис не вдався)

};

test();

У режимі use strict код вище видав би помилку.

Як правило, ім'я NFE використовується для єдиної мети - дозволити зсередини функції викликати саму себе.

**Приклад використання.**

NFE використовується в першу чергу в тих ситуаціях, коли функцію потрібно передавати в інше місце коду або переміщати з однієї змінної в іншу.

Внутрішнє ім'я дозволяє функції надійно звертатися до самої себе, де б вона не знаходилася.

Згадаймо, наприклад, функцію-факторіал з завдання Обчислити факторіал:

function f(n) {

return n ? n \* f(n - 1) : 1;

};

alert( f(5) ); // 120

Спробуємо перенести її в іншу змінну g:

function f(n) {

return n ? n \* f(n - 1) : 1;

};

var g = f;

f = null;

alert( g(5) ); // запуск функції с новим ім’ям - помилка при виконанні!

Помилка виникла тому що функція зі свого коду звертається до свого старого імені f. А цієї функції вже немає, f = null.

Для того, щоб функція завжди надійно працювала, оголосимо її як Named Function Expression:

var f = function factorial(n) {

return n ? n\*factorial(n-1) : 1;

};

var g = f; // скопіювали посилання на функцію-факторіал в g

f = null;

alert( g(5) ); // 120, працює!

**Застаріле спеціальне значення arguments.callee**

Якщо ви давно працюєте з JavaScript, то, можливо, знаєте, що раніше для цієї мети також служило спеціальне значення arguments.callee. Якщо ця назва вам ні про що не говорить - все в порядку, читайте далі, ми обов'язково обговоримо його в окремому розділі. Якщо ж ви в курсі, то варто мати на увазі, що воно офіційно виключено з сучасного стандарту. А NFE - це наше сьогодення.

**Висновки**

Якщо функція задана як Function Expression, їй можна дати ім'я. Воно буде доступне тільки всередині функції (крім IE8-). Це ім'я призначене для надійного рекурсивного виклику функції, навіть якщо вона записана в іншу зміну.

Звернемо увагу, що з Function Declaration так вчинити не можна. Таке «спеціальне» внутрішнє ім'я функції задається тільки в синтаксисі Function Expression.

**3. Введення в методи і властивості.**

Всі значення в JavaScript, за винятком null і undefined, містять набір допоміжних функцій і значень, доступних «через точку».

Такі функції називають «методами», а значення - «властивостями». Тут ми розглянемо основи використання властивостей і методів.

**Властивість str.length**

У рядки є властивість length, що містить довжину:

alert( "Привіт, світ!".length ); // 12

Можна і записати рядок в змінну, а потім запросити її властивість:

var str = "Привіт, світ!";

alert( str.length ); // 12

**Метод str.toUpperCase()**

Також у рядків є метод toUpperCase(), який повертає рядок у верхньому регістрі:

var hello = "Привіт, світ!";

alert( hello.toUpperCase() ); // "ПРИВІТ, СВІТ!"

**Виклик методу - через круглі дужки!**

Зверніть увагу, для виклику методу обов'язково потрібні круглі дужки.

Подивіться, наприклад, результат звернення до toUpperCase без дужок:

var hello = "Привіт";

alert( hello.toUpperCase ); // function...

Метод - це вбудована команда («функція», ми поговоримо про них пізніше), яку потрібно викликати для отримання значення. При зверненні без дужок ми отримаємо саму цю функцію. Як правило браузер виведе її якось так: "function toUpperCase() { ... }".

А щоб отримати результат - потрібно провести її виклик, додавши дужки:

var hello = "Привіт";

alert( hello.toUpperCase() ); // ПРИВІТ

**Метод num.toFixed (n)**

Є методи і у чисел, наприклад num.toFixed(n). Він округлює число num до n знаків після коми, при необхідності добиває нулями до даної довжини і повертає у вигляді рядка (зручно для форматованого виведення):

var n = 12.345;

alert( n.toFixed(2) ); // "12.35"

alert( n.toFixed(0) ); // "12"

alert( n.toFixed(5) ); // "12.34500"

**Звернення до методів чисел**

До методу числа можна звернутися і безпосередньо:

alert( 12.34.toFixed(1) ); // 12.3

... Але якщо число ціле, то буде проблема:

alert(12.toFixed(1)); // помилка!

Помилка відбудеться тому, що JavaScript очікує десятковий дріб після точки.

Це - особливість синтаксису JavaScript. Ось так - буде працювати:

alert( 12..toFixed(1) ); // 12.0

**Висновки**

У цьому розділі ми познайомилися з методами і властивостями.

Майже всі значення в JavaScript, крім хіба що null і undefined мають їх і надають через них різну функціональність.

Далі ми докладно розберемо основні властивості і методи структур даних в JavaScript.