**Лекція 11. Методи об’єктів, перетворення типів об’єктів.**

1. Методи об'єктів, ключове слово this
2. Перетворення об'єктів: *toString* і *valueOf*
3. Створення об'єктів через ключове слово "new"
4. **Методи об'єктів, ключове слово this**

До цього ми говорили про об'єкт лише як про сховище значень. Тепер підемо далі і поговоримо про об'єкти як про сутності зі своїми функціями («методами»).

*Методи у об'єктів*

При оголошенні об'єкта можна вказати властивість-функцію, наприклад:

var user = {

name: 'Василій',

// метод

sayHi: function() {

alert( 'Привіт!' );

}

};

// Виклик

user.sayHi();

Властивості-функції називають «методами» об'єктів. Їх можна додавати і видаляти в будь-який момент, в тому числі і явним привласненням:

var user = {

name: 'Василій'

};

user.sayHi = function() { // привласнили метод після створення об’єкту

alert('Привіт!');

};

// Виклик методу:

user.sayHi();

*Доступ до об'єкту через this*

Для повноцінної роботи метод повинен мати доступ до даних об'єкта. Зокрема, виклик user.sayHi () може захотіти вивести ім'я користувача.

Для доступу до поточного об'єкту з методу використовується ключове слово this.

Значенням this є об'єкт перед «точкою», в контексті якого викликаний метод, наприклад:

var user = {

name: 'Василій',

sayHi: function() {

alert( this.name );

}

};

user.sayHi(); // sayHi в контексті user

Тут при виконанні функції user.sayHi () в this буде зберігатися посилання на поточний об'єкт user.

Замість this всередині sayHi можна було б звернутися до об'єкта, використовуючи змінну user:

...

sayHi: function() {

alert( user.name );

}

...

Однак, таке рішення нестабільно. Якщо ми вирішимо скопіювати об'єкт в іншу змінну, наприклад admin = user, а в змінну user записати щось інше - звернення буде зовсім не за адресою:

var user = {

name: 'Василій',

sayHi: function() {

alert( user.name ); // призведе до помилки

}

};

var admin = user;

user = null;

admin.sayHi(); // // упс! всередині sayHi звернення за старим ім’ям, помилка!

Використання this гарантує, що функція працює саме з тим об'єктом, в контексті якого викликана.

Через this метод може не тільки звернутися до будь-якого властивості об'єкта, а й передати кудись посилання на сам об'єкт цілком:

var user = {

name: 'Василій',

sayHi: function() {

showName(this); // передати поточний об'єкт в showName

}

};

function showName(namedObj) {

alert( namedObj.name );

}

user.sayHi(); // Василій

Будь-яка функція може мати в собі this. Абсолютно неважливо, оголошена чи вона в об'єкті або окремо від нього.

Значення this називається контекстом виклику і буде визначено в момент виклику функції.

Наприклад, така функція, оголошена без об'єкта, цілком допустима:

function sayHi() {

alert( this.firstName );

}

Ця функція ще не знає, яким буде this. Це з'ясується при виконанні програми.

Якщо одну і ту ж функцію запускати в контексті різних об'єктів, вона буде отримувати різний this:

var user = { firstName: "Вася" };

var admin = { firstName: "Адмін" };

function func() {

alert( this.firstName );

}

user.f = func;

admin.g = func;

// дорівнює об'єкту перед точкою:

user.f(); // Вася

admin.g(); // Адмін

admin['g'](); // Адмін (не важливо, доступ до об'єкту через точку або квадратні дужки)

Отже, значення this не залежить від того, як функція була створена, вона визначається виключно в момент виклику.

*Значення this при виклику без контексту*

Якщо функція використовує this - це має на увазі роботу з об'єктом. Але і прямий виклик func () технічно можливий. Як правило, така ситуація виникає при помилку в розробці. При цьому this отримує значення window, глобального об'єкта:

function func() {

alert( this ); // виведе [object Window] або [object global]

}

func();

Таке поводження в старому стандарті. А в режимі use strict замість глобального об'єкта this буде undefined:

function func() {

"use strict";

alert( this ); // в виведе undefined (крім IE9-)

}

func();

Зазвичай якщо в функції використовується this, то вона, все ж, служить для виклику в контексті об'єкта, так що така ситуація - швидше виняток.

*Нормативний тип*

Контекст this ніяк не прив'язаний до функції, навіть якщо вона створена в оголошенні об'єкта. Щоб this передався, потрібно викликати функцію саме через точку (або квадратні дужки).

Будь-яка більш хитрий виклик призведе до втрати контексту, наприклад:

var user = {

name: "Вася",

hi: function() { alert(this.name); },

bye: function() { alert("Пока"); }

};

user.hi(); // Вася (простий виклик працює)

// а тепер викличемо user.hi або user.bye в залежності від імені

(user.name == "Вася" ? user.hi : user.bye)(); // undefined

В останньому рядку прикладу метод отриманий в результаті виконання тернарного оператора і тут же викликаний. Але this при цьому втрачається.

Якщо хочеться зрозуміти, чому, то причина криється в деталях роботи виклику obj.method ().

Адже він, насправді, складається з двох незалежних операцій: точка. - отримання властивості і дужки () - його виклик (мається на увазі, що це функція).

Функція, як ми говорили раніше, сама по собі не запам'ятовує контекст. Щоб «донести його» до дужок, JavaScript застосовує «фінт вухами» - точка повертає не функція, а значення спеціального «засланого» типу Reference Type.

Цей тип являє собою в'язку «base-name-strict», де:

base - якраз об'єкт,

name - ім'я властивості,

strict - допоміжний прапор для передачі use strict.

Тобто, контрольний тип (Reference Type) - це своєрідне «три-в-одному». Він існує виключно для цілей специфікації, ми його не бачимо, оскільки будь-який оператор відразу від нього позбавляється:

Дужки () отримують з base значення властивості name і викликають в контексті base.

Інші оператори отримують з base значення властивості name і використовують, а інші компоненти ігнорують.

Тому будь-яка операція над результатом операції отримання властивості, крім виклику, призводить до втрати контексту.

Аналогічно працює і отримання властивості через квадратні дужки obj [method].

1. **Перетворення об'єктів: *toString* і *valueOf***

Раніше, в розділі *Перетворення типів для примітивів* ми розглядали перетворення типів для примітивів. Тепер додамо в нашу картину світу об'єкти.

Бувають операції, при яких об'єкт повинен бути перетворений в примітив, Наприклад:

*Строкове перетворення* - якщо об'єкт виводиться через alert (obj).

*Чисельне перетворення* - при арифметичних операціях, порівняно з примітивом.

*Логічне перетворення* - при if (obj) і інших логічних операціях.

***Логічне перетворення***

Найпростіше - з логічним перетворенням. Будь-який об'єкт в логічному контексті - true, навіть якщо це порожній масив [] або об'єкт {}.

if ({} && []) {

alert( "Усі об’єкти - true!" ); // alert спрацює

}

***Строкове перетворення***

Строкове перетворення найпростіше побачити, якщо вивести об'єкт за допомогою alert:

var user = {

firstName: 'Василій'

};

alert( user ); // [object Object]

Як видно, вміст об'єкта не відобразився. Це тому, що стандартним строковим поданням призначеного для користувача об'єкта є рядок "[object Object]".

Такий висновок об'єкта не містить цікавої інформації. Тому має сенс його поміняти на щось більш корисне.

Якщо в об'єкті присутня метод toString, який повертає примітив, то він використовується для перетворення.

var user = {

firstName: 'Василій',

toString: function() {

return 'Користувач ' + this.firstName;

}

};

alert( user ); // Користувач Василій

Результатом toString може бути будь-який примітив. Метод toString не зобов'язаний повертати саме рядок. Його результат може бути будь-якого примітивного типу. Наприклад, це може бути число, як в прикладі нижче:

var obj = {

toString: function() {

return 123;

}

};

alert( obj ); // 123

Тому ми і називаємо його тут «строкове перетворення», а не «перетворення до рядка». Всі об'єкти, включаючи вбудовані, мають свої реалізації методу toString, наприклад:

alert( [1, 2] ); // toString для масивів виводить список елементів "1,2"

alert( new Date ); // toString для дат виводить дату у вигляді рядка

alert( function() {} ); // toString для функції виводить її код

***Чисельне перетворення***

Для чисельного перетворення об'єкта використовується метод valueOf, а якщо його немає - то toString:

var room = {

number: 777,

valueOf: function() { return this.number; },

toString: function() { return this.number; }

};

alert( +room ); // 777, викликається valueOf

delete room.valueOf; // valueOf видалено

alert( +room ); // 777, викликався toString

Метод valueOf зобов'язаний повертати примітивне значення, інакше його результат буде проігнорований. При цьому - не обов'язково числове.

У більшості вбудованих об'єктів такого valueOf немає, тому чисельне і строкове перетворення для них працюють однаково.

Винятком є ​​об'єкт Date, який підтримує обидва типи перетворень:

 alert( new Date() ); // toString: Дата у вигляді читається рядки

alert( +new Date() ); // valueOf: : к-ть мілісекунд, що пройшли з 01.01.1970

***Дві стадії перетворення***

Отже, об'єкт перетворений в примітив за допомогою toString або valueOf. Але на цьому перетворення не обов'язково закінчуються. Цілком можливо, що в процесі обчислень цей примітив буде перетворений в щось інше.

Наприклад, розглянемо застосування до об'єкту операції ==:

var obj = {

valueOf: function() {

return 1;

}

};

alert( obj == true ); // true

Об'єкт obj був спочатку перетворений в примітив, використовуючи чисельну перетворення, вийшло 1 == true.

Далі, так як значення все ще різних типів, застосовуються правила перетворення примітивів, результат: true.

Те ж саме - при додаванні з об'єктом за допомогою +:

var obj = {

valueOf: function() {

return 1;

}

};

alert( obj + "test" ); // 1test

Або ось, для різниці об'єктів:

var a = {

valueOf: function() {

return "1";

}

};

var b = {

valueOf: function() {

return "2";

}

};

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

alert( a - b ); // "1" - "2" = -1

Виняток: Date

Бінарний оператор плюс + зазвичай використовує чисельну перетворення і метод valueOf. Як ми вже знаємо, якщо відповідного valueOf немає (а його немає у більшості об'єктів), то використовується toString, так що в підсумку перетворення відбувається до рядка. Але якщо є valueOf, то використовується valueOf. Вище в прикладі якраз a + b це демонструють.

У об'єктів Date є і valueOf - повертає кількість мілісекунд, і toString - повертає рядок з датою. Але оператор + для Date використовує саме toString (хоча мав би valueOf).

Це і є виняток:

// бінарний плюс для дати toString, для інших об'єктів valueOf

alert( new Date + "" ); // "рядок дати"

Інших подібних винятків немає.

***Висновки***

У логічному контексті об'єкт - завжди true.

При строковому перетворенні об'єкта використовується його метод toString. Він повинен повертати примітивне значення, причому не обов'язково саме рядок.

Для чисельного перетворення використовується метод valueOf, який також може повернути будь-примітивне значення. У більшості об'єктів valueOf не працює (повертає сам об'єкт і тому ігнорується), при цьому для чисельного перетворення використовується toString.

Зауважимо, для повноти картини, що деякі тести знань в інтернет пропонують запитання на кшталт:

{}[0] // чому дорівнює?

{} + {} // а так?

Якщо ви запустите ці вирази в консолі, то результат може здатися дивним. Підступ тут в тому, що якщо фігурні дужки {...} йдуть не у виразі, а в основному потоці коду, то JavaScript вважає, що це не об'єкт, а «блок коду» (як if, for, але без оператора просто угруповання команд разом використовується рідко).

Ось блок коду з командою:

{

alert("Блок")

}

А якщо команду вилучити, то буде порожній блок {}, який нічого не робить. Два приклади вище якраз містять порожній блок на початку, який нічого не робить. Інакше кажучи:

{}[0] // то же что и: [0]

{} + {} // то же что и: + {}

Тобто, такі питання - нема на перетворення типів, а на розуміння, що якщо {...} знаходиться поза виразів, то це не об'єкт, а блок.

1. **Створення об'єктів через "new"**

Звичайний синтаксис {...} дозволяє створити один об'єкт. Але найчастіше потрібно створити багато однотипних об'єктів. Для цього використовують «функції-конструктори», запускаючи їх за допомогою спеціального оператора new.

конструктор

Конструктором стає будь-яка функція, викликана через new. Наприклад:

function Animal(name) {

this.name = name;

this.canWalk = true;

}

var animal = new Animal("їжачок");

Зауважимо, що, технічно, будь-яка функція може бути використана як конструктор. Тобто, будь-яку функцію можна викликати за допомогою new. Якось особливим чином вказувати, що вона - конструктор - не треба.

Але, щоб виділити функції, задумані як конструктори, їх називають з великої літери: Animal, а не animal.

Детальніше - функція, запущена через new, робить наступне:

* Створюється новий порожній об'єкт.
* Ключове слово this отримує посилання на цей об'єкт.
* Функція виконується. Як правило, вона модифікує this (тобто цей новий об'єкт), додає методи, властивості.
* Повертається this.

В результаті виклику new Animal("їжачок"); отримуємо такий об'єкт:

animal = {

name: " їжачок ",

canWalk: true

}

Іншими словами, при виклику new Animal відбувається щось в такому дусі (перша і остання рядок - це те, що робить інтерпретатор):

function Animal(name) {

// this = {};

// в this пишемо властивості, методи

this.name = name;

this.canWalk = true;

// return this;

}

Тепер багаторазовими викликами new Animal з різними параметрами ми можемо створити стільки об'єктів, скільки потрібно. Тому таку функцію і називають конструктором - вона призначена для «конструювання» об'єктів.

Іноді функцію-конструктор оголошують і тут же використовують, ось так:

var animal = new function() {

this.name = "Васька";

this.canWalk = true;

};

Так роблять, коли хочуть створити єдиний об'єкт даного типу. Приклад вище з тим же успіхом можна було б переписати як:

var animal = {

name: "Васька",

canWalk: true

}

Але звичайний синтаксис {...} не підходить, коли при створенні властивостей об'єкта потрібні більш складні обчислення. Їх можна виконати в функції-конструкторі і записати результат в this.

***Правила обробки return***

Як правило, конструктори нічого не повертають. Їх завдання - записати все, що потрібно, в this, який автоматично стане результатом. Але якщо явний виклик return все ж є, то застосовується просте правило:

* При виклику return з об'єктом, буде повернуто він, а не this.
* При виклику return з примітивним значенням, воно буде відкинуте.

Іншими словами, виклик return з об'єктом поверне об'єкт, а з чим завгодно, крім об'єкта - поверне, як зазвичай, this.

Наприклад, повернення об'єкта:

function BigAnimal() {

this.name = "Миша";

return { name: "Годзила" }; // <-- повернемо об'єкт

}

alert( new BigAnimal().name ); // Годзила, отримали об'єкт замість this

А ось приклад з поверненням рядки:

function BigAnimal() {

this.name = "Миша";

return "Годзила"; // <-- повернемо примітив

}

alert( new BigAnimal().name ); // Миша, отримали this (а Годзилла зник)

Ця особливість роботи new прописана в стандарті, але використовується вона досить рідко.

До речі, при виклику new без аргументів дужки можна не ставити:

var animal = new BigAnimal; // <-- без дужок

// теж саме, як і

var animal = new BigAnimal();

Не сказати, що викидання дужок - «хороший стиль», але такий синтаксис допустимий стандартом.

***Створення методів в конструкторі***

Використання функцій для створення об'єкта дає більшу гнучкість. Можна передавати конструктору параметри, що визначають яким чином вони створюються, і він буде «клепати» об'єкти заданим чином.

Додамо в створюваний об'єкт ще й метод. Наприклад, new User (name) створює об'єкт із заданим значенням властивості name і методом sayHi:

function User(name) {

this.name = name;

this.sayHi = function() {

alert( " Моє ім’я: " + this.name );

};

}

var ivan = new User("Іван");

ivan.sayHi(); // Моє ім’я: Іван

/\*

ivan = {

name: "Іван",

sayHi: функція

}

\*/

***Локальні змінні***

У функції-конструкторі буває зручно оголосити допоміжні локальні змінні і вкладені функції, які будуть видні тільки всередині:

function User(firstName, lastName) {

// допоміжна змінна

var phrase = "Привіт";

// допоміжна вкладена функція

function getFullName() {

return firstName + " " + lastName;

}

this.sayHi = function() {

alert( phrase + ", " + getFullName() ); // використання

};

}

var vasya = new User("Вася", "Шевченко");

vasya.sayHi(); // Привіт, Вася Шевченко

Ті функції і дані, які повинні бути доступні для зовнішнього коду, ми пишемо в this - і до них можна буде звертатися, як наприклад vasya.sayHi (), а допоміжні, які потрібні тільки всередині самого об'єкта, зберігаємо в локальній області видимості.

**Висновки**

Об'єкти можуть бути створені за допомогою функцій-конструкторів:

* Будь-яка функція може бути викликана з new, при цьому вона отримує новий порожній об'єкт в якості this, в який вона додає властивості. Якщо функція не вирішить повернути свій об'єкт, то її результатом буде this.
* Функції, які призначені для створення об'єктів, називаються конструкторами. Їх назви пишуть з великої літери, щоб відрізняти від звичайних.