**Лекція 12. Методи властивостей об’єктів.**

1. Дескриптори, гетери і сетери властивостей.
2. Статичні та фабричні методи.
3. **Дескриптори, гетери і сетери властивостей**

У цьому розділі ми розглянемо можливості, які дозволяють дуже гнучко і потужно управляти всіма властивостями об'єкта, включаючи їх аспекти - змінність, видимість в циклі for..in і навіть непомітно робити їх функціями. Вони підтримуються всіма сучасними браузерами, але не IE8-. Втім, навіть IE8 їх підтримує, але тільки для DOM-об'єктів (використовуються при роботі зі сторінкою, це зараз поза нашого розгляду).

Основний метод для управління властивостями - Object.defineProperty. Він дозволяє оголосити властивість об'єкта і, що найголовніше, тонко налаштувати його особливі аспекти, які ніяк інакше не змінити.

Object.defineProperty(obj, prop, descriptor)

*Аргументи:*

Obj - об'єкт, в якому оголошується властивість.

Prop - ім'я властивості, яке потрібно оголосити або модифікувати.

Descriptor- дескриптор - об'єкт, який описує поведінку властивості.

У ньому можуть бути наступні поля:

value - значення властивості, за замовчуванням undefined;

writable - значення властивості можна змінювати, якщо true. За замовчуванням false;

configurable - якщо true, то властивість можна видаляти, а також змінювати його в подальшому за допомогою нових викликів defineProperty. За замовчуванням false;

enumerable - якщо true, то властивість проглядається в циклі for..in і методі Object.keys(). За замовчуванням false;

get - функція, яка повертає значення властивості. За замовчуванням undefined;

set - функція, яка записує значення властивості. За замовчуванням undefined;

Щоб уникнути конфлікту, заборонено одночасно вказувати значення value і функції get/set. Або значення, або функції для його читання-запису, одне з двох. Також заборонено і не має сенсу вказувати writable при наявності get/set-функцій.

Далі ми докладно розберемо ці властивості на прикладах.

***Звичайна властивість***

Два таких виклику працюють однаково:

var user = {};

// 1. просте привласнення

user.name = "Вася";

// 2. вказування значення через дескриптор

Object.defineProperty(user, "name", { value: "Вася", configurable: true, writable: true, enumerable: true });

Обидва виклику вище додають в об'єкт user звичайне (видаляється, змінюване, перераховує) властивість.

***Властивість-константа***

Для того, щоб зробити властивість незмінною, змінимо його прапори writable і configurable:

Обидва виклика вище додають в об'єкт user звичайну (видаляється, змінюване, перераховує) властивість.

"use strict";

var user = {};

Object.defineProperty(user, "name", {

value: "Вася",

writable: false, // заборонити привласнення "user.name="

configurable: false // заборонити видалення "delete user.name"

});

// Тепер спробуємо змінити цю властивість.

// в strict mode присвоение "user.name=" викличе помилку

user.name = "Петя";

Зауважимо, що без use strict операція записи «мовчки» не спрацює. Лише якщо встановлено режим use strict, то додатково згенерує помилка.

***Властивість, прихована для for ... in***

Вбудований метод toString, як і більшість вбудованих методів, не бере участі в циклі for..in. Це зручно, так як зазвичай така властивість є «службовою».

На жаль, властивість toString, оголошене звичайним способом, буде видно в циклі for..in, наприклад:

var user = {

name: "Вася",

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

};

for(var key in user) alert(key); // name, toString

Ми б хотіли, щоб поведінка нашого методу toString було таким же, як і стандартного.

Object.defineProperty може виключити toString зі списку ітерації, поставивши йому прапор enumerable: false. За стандартом, у вбудованого toString цей прапор вже стоїть.

var user = {

name: "Вася",

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

};

// помечаем toString как не подлежащий перебору в for..in

Object.defineProperty(user, "toString", {enumerable: false});

for(var key in user) alert(key); // name

Звернемо увагу, виклик defineProperty НЕ перезаписав властивість, а просто модифікував настройки у існуючого toString.

***Властивість-функція***

Дескриптор дозволяє задати властивість, яка насправді працює як функція. Для цього в ньому потрібно вказати цю функцію в get.

Наприклад, у об'єкта user є звичайні властивості: ім'я firstName та прізвище surname.

Створимо властивість fullName, яке насправді є функцією:

var user = {

firstName: "Вася",

surname: "Петренко"

}

Object.defineProperty(user, "fullName", {

get: function() {

return this.firstName + ' ' + this.surname;

}

});

alert(user.fullName); // Вася Петренко

Звернемо увагу, зовні fullName - це звичайна властивість user.fullName. Але дескриптор вказує, що насправді його значення повертається функцією.

Також можна вказати функцію, яка використовується для запису значення, за допомогою дескриптора set.

Наприклад, додамо можливість присвоєння user.fullName наприклад вище:

var user = {

firstName: "Вася",

surname: " Петренко"

}

Object.defineProperty(user, "fullName", {

get: function() {

return this.firstName + ' ' + this.surname;

},

set: function(value) {

var split = value.split(' ');

this.firstName = split[0];

this.surname = split[1];

}

});

user.fullName = "Петро Васюченко";

alert( user.firstName ); // Петро

alert( user.surname ); // Васюченко

***Вказування get/set в літералах***

Якщо ми створюємо об'єкт за допомогою синтаксису {...}, то задати властивості-функції можна прямо в його визначенні. Для цього використовується спеціальний синтаксис: get властивість або set властивість.

Наприклад, нижче оголошений гетер-сетер fullName:

var user = {

firstName: "Вася",

surname: "Петренко",

get fullName() {

return this.firstName + ' ' + this.surname;

},

set fullName(value) {

var split = value.split(' ');

this.firstName = split[0];

this.surname = split[1];

}

};

alert( user.fullName ); // Вася Петренко (з геттера)

user.fullName = "Петро Васюченко";

alert( user.firstName ); // Петро (поставив сеттер)

alert( user.surname ); // Васюченко (поставив сеттер)

Здавалося б, навіщо нам призначати get/set для властивості через всякі хитрі виклики, коли можна зробити просто функції з самого початку? Наприклад, getFullName, setFullName ...

Звичайно, в ряді випадків властивості виглядають коротше, таке рішення просто може бути красивим. Але основний бонус - це гнучкість, можливість отримати контроль над властивістю в будь-який момент!

Наприклад, на початку розробки ми використовуємо звичайні властивості, наприклад у User буде ім'я name і вік age:

function User(name, age) {

this.name = name;

this.age = age;

}

var pete = new User("Петя", 25);

alert( pete.age ); // 25

З звичайними властивостями в коді менше букв, вони зручні, причини використовувати функції поки немає.

... Але рано чи пізно можуть статися зміни. Наприклад, в User може стати більш доцільно замість віку age зберігати дату народження birthday:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

}

var pete = new User("Петя", new Date(1987, 6, 1));

Що тепер робити зі старим кодом, який виводить властивість age?

Можна, звичайно, знайти всі місця і поправити їх, але це довго, а іноді і неможливо, скажімо, якщо ви взаємодієте зі сторонньою бібліотекою, код в якій - чужий і влазити в нього небажано.

Додавання get-функції age дозволяє обійти проблему легко і невимушено:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

// age буде враховувати вік по birthday

Object.defineProperty(this, "age", {

get: function() {

var today = new Date();

var yearDelta = today.getFullYear() - this.birthday.getFullYear();

if (today.getMonth() > this.birthday.getMonth() ||

(today.getMonth() === this.birthday.getMonth() && today.getDate() >= this.birthday.getDate())) {

return yearDelta;

}

return yearDelta - 1;

}

});

}

var pete = new User("Петро", new Date(1987, 6, 1));

alert( pete.birthday ); // і дата народження доступна

alert( pete.age ); // і вік

Зауважимо, що pete.age зовні як було властивістю, так і залишилося. Тобто, переписувати зовнішній код на виклик функції pete.age () не потрібно.

Таким чином, defineProperty дозволяє нам почати з звичайних властивостей, а в майбутньому, при необхідності, можна в будь-який момент замінити їх на функції, що реалізують складнішу логіку.

***Інші методи роботи з властивостями***

Object.defineProperties (obj, descriptors)

Дозволяє оголосити кілька властивостей відразу:

var user = {}

Object.defineProperties(user, {

firstName: {

value: "Петро"

},

surname: {

value: "Васюченко"

},

fullName: {

get: function() {

return this.firstName + ' ' + this.surname;

}

}

});

alert( user.fullName ); // Петро Васюченко

Object.keys (obj), Object.getOwnPropertyNames (obj)

Повертає масив - список властивостей об'єкта.

Object.keys повертає тільки enumerable-властивості.

Object.getOwnPropertyNames - повертає всі:

var obj = {

a: 1,

b: 2,

internal: 3

};

Object.defineProperty(obj, "internal", {

enumerable: false

});

alert( Object.keys(obj) ); // a,b

alert( Object.getOwnPropertyNames(obj) ); // a, b, internal

Object.getOwnPropertyDescriptor (obj, prop)

Повертає дескриптор для властивості obj [prop].

Отриманий дескриптор можна змінити і використовувати defineProperty для збереження змін, наприклад:

var obj = {

test: 5

};

var descriptor = Object.getOwnPropertyDescriptor(obj, 'test');

// замінимо value на гетер, для цього...

delete descriptor.value; // ..треба прибрати value/writable

delete descriptor.writable;

descriptor.get = function() { // і встановити get

alert( "Preved :)" );

};

// встановимо нову властивість замість старої

// якщо не видалити - defineProperty об’єднає дескриптор з новим

delete obj.test;

Object.defineProperty(obj, 'test', descriptor);

obj.test; // Preved :)

... І кілька методів, які використовуються дуже рідко:

Object.preventExtensions (obj)

Забороняє додавання властивостей в об'єкт.

Object.seal (obj)

Забороняє додавання і видалення властивостей, всі поточні властивості робить configurable: false.

Object.freeze (obj)

Забороняє додавання, видалення і зміна властивостей, всі поточні властивості робить configurable: false, writable: false.

Object.isExtensible (obj)

Повертає false, якщо додавання властивостей об'єкта було заборонено викликом методу Object.preventExtensions.

Object.isSealed (obj)

Повертає true, якщо додавання і видалення властивостей об'єкта заборонено, і всі поточні властивості є configurable: false.

Object.isFrozen (obj)

Повертає true, якщо додавання, видалення і зміна властивостей об'єкта заборонено, і всі поточні властивості є configurable: false, writable: false.

1. **Статичні та фабричні методи**

Методи і властивості, які не прив'язані до конкретного екземпляру об'єкта, називають «статичними». Їх записують прямо в саму функцію-конструктор.

***Статичні властивості***

У коді нижче використовуються статичні властивості Article.count і Article.DEFAULT\_FORMAT:

function Article() {

Article.count++;

}

Article.count = 0; // статична властивість-змінна

Article.DEFAULT\_FORMAT = "html"; // статична властивість-константа

Вони зберігають дані, специфічні не для одного об'єкта, а для всіх статей цілком.

Як правило, це частіше константи, такі як формат «за замовчуванням» Article.DEFAULT\_FORMAT.

***Статичні методи***

З прикладами статичних методів ми вже знайомі: це вбудовані методи String.fromCharCode, Date.parse.

Створимо для Article статичний метод Article.showCount ():

function Article() {

Article.count++;

//...

}

Article.count = 0;

Article.showCount = function() {

alert( this.count ); // (1)

}

// використання

new Article();

new Article();

Article.showCount(); // (2)

Тут Article.count - статична властивість, а Article.showCount - статичний метод.

Звернемо увагу на використання this в прикладі вище. Незважаючи на те, що змінна і метод - статичні, він все ще корисний. У рядку (1) він дорівнює Article.

***Приклад: порівняння об'єктів***

Ще один хороший спосіб застосування - порівняння об'єктів. Наприклад, у нас є об'єкт Journal для журналів. Журнали можна порівнювати - по товщині, по вазі, по іншим параметрам.

Оголосимо «стандартну» функцію порівняння, яка буде порівнювати за датою видання. Ця функція порівняння, природно, не прив'язана до конкретного журналу, але відноситься до журналів взагалі.

Тому поставимо її як статичний метод Journal.compare:

function Journal(date) {

this.date = date;

// ...

}

// вертає значення, більше за 0, якщо A більше B, інакше меньше за 0

Journal.compare = function(journalA, journalB) {

return journalA.date - journalB.date;

};

У прикладі нижче ця функція використовується для пошуку самого раннього журналу з масиву:

function Journal(date) {

this.date = date;

this.formatDate = function(date) {

return date.getDate() + '.' + (date.getMonth() + 1) + '.' + date.getFullYear();

};

this.getTitle = function() {

return "Випуск від " + this.formatDate(this.date);

};

}

Journal.compare = function(journalA, journalB) {

return journalA.date - journalB.date;

};

// використання:

var journals = [

new Journal(new Date(2012, 1, 1)),

new Journal(new Date(2012, 0, 1)),

new Journal(new Date(2011, 11, 1))

];

function findMin(journals) {

var min = 0;

for (var i = 0; i < journals.length; i++) {

// використовуємо статичний метод

if (Journal.compare(journals[min], journals[i]) > 0) min = i;

}

return journals[min];

}

alert( findMin(journals).getTitle() );

Статичний метод також можна використовувати для функцій, які взагалі не вимагають наявності об'єкта.

Наприклад, метод formatDate (date) можна зробити статичним. Він буде форматувати дату «як це прийнято в журналах», при цьому його можна використовувати в будь-якому місці коду, не обов'язково створювати журнал.

Наприклад:

function Journal() { /\*...\*/ }

Journal.formatDate = function(date) {

return date.getDate() + '.' + (date.getMonth()+1) + '.' + date.getFullYear();

}

// жодного об’єкту Journal немає, просто форматуємо дату

alert( Journal.formatDate(new Date) );

***Фабричні методи***

Розглянемо ситуацію, коли об'єкт потрібно створювати різними способами. Наприклад, це реалізовано у вбудованому об'єкті Date. Він по-різному обробляє аргументи різних типів:

* new Date () - створює об'єкт з поточною датою,
* new Date (milliseconds) - створює дату за кількістю мілісекунд milliseconds,
* new Date (year, month, day ...) - створює дату по компонентам рік, місяць, день ...
* new Date (datestring) - читає дату з рядка datestring

"Фабричний статичний метод" - зручна альтернатива такому конструктору. Так називається статичний метод, який служить для створення нових об'єктів (тому і називається «фабричним»).

Приклад вбудованого фабричного методу - String.fromCharCode (code). Цей метод створює рядок з коду символу:

 var str = String.fromCharCode(65);

alert( str ); // 'A'

Але рядки - занадто простий приклад, подивимося що-небудь складніше. Припустимо, нам потрібно створювати об'єкти User: анонімні new User () і з даними new User ({name: 'Вася', age: 25}).

Можна, звичайно, створити полиморфну функцію-конструктор User:

function User(userData) {

if (userData) { // если указаны данные -- одна ветка if

this.name = userData.name;

this.age = userData.age;

} else { // если не указаны -- другая

this.name = 'Аноним';

}

this.sayHi = function() {

alert(this.name)

};

// ...

}

// Использование

var guest = new User();

guest.sayHi(); // Аноним

var knownUser = new User({

name: 'Вася',

age: 25

});

knownUser.sayHi(); // Вася

Підхід з використанням фабричних методів був би іншим. Замість розбору параметрів в конструкторі - робимо два методи: User.createAnonymous і User.createFromData.

function User() {

this.sayHi = function() {

alert(this.name)

};

}

User.createAnonymous = function() {

var user = new User;

user.name = 'Анонім';

return user;

}

User.createFromData = function(userData) {

var user = new User;

user.name = userData.name;

user.age = userData.age;

return user;

}

// Використання

var guest = User.createAnonymous();

guest.sayHi(); // Анонім

var knownUser = User.createFromData({

name: 'Вася',

age: 25

});

knownUser.sayHi(); // Вася

***Переваги використання фабричних методів:***

Краща читаність коду. Як конструктора - замість однієї великої функції кілька маленьких, так і викликає коду - явно видно, що саме створюється.

Кращий контроль помилок, тому що якщо в createFromData нічого не передали, то буде помилка, а поліморфний конструктор створив би анонімного відвідувача.

Зручна розширюваність. Наприклад, потрібно додати створення адміністратора, без аргументів. Фабричний метод зробити легко: User.createAdmin = function () {...}. А для поліморфного конструктора виклик без аргументу створить аноніма, так що потрібно додати параметр - «тип відвідувача» і ускладнити цим код.

Тому поліморфні конструктори краще використовувати там, де потрібен саме поліморфізм, тобто коли незрозуміло, якого типу аргумент передадуть, і хочеться в одному конструкторі охопити всі варіанти.

А в інших випадках відмінна альтернатива - фабричні методи.