**Лекція 10. Модулі через замикання.** **Управління пам'яттю в JavaScript.**

1. Модулі через замикання.

2. Управління пам'яттю в JavaScript.

Прийом програмування «модуль» має величезну кількість варіацій. Він трохи схожий на лічильник, який ми розглядали раніше, використовує аналогічний прийом, але на рівні вище.

Його мета - приховати внутрішні деталі реалізації скрипта. У тому числі: тимчасові змінні, константи, допоміжні міні-функції і т.п.

Навіщо потрібен модуль?

Припустимо, ми хочемо розробити скрипт, який робить щось корисне на сторінці.

Вміючи працювати зі сторінкою, ми могли б зробити багато чого, але так як поки цього не було (скоро навчимося), то нехай скрипт просто виводить повідомлення:

Файл hello.js

// глобальна змінна нашого скрипта

var message = "Привіт";

// функція для виводу цьої змінної

function showMessage() {

alert( message );

}

// виводимо повідомлення

showMessage();

У цього скрипта є свої внутрішні змінні і функції.

В даному випадку це message і showMessage.

Припустимо, що ми хотіли б поширювати цей скрипт у вигляді бібліотеки. Кожен, хто хоче, щоб відвідувачам видавалося «Привіт» - може просто підключити цей скрипт. Досить завантажити і підключити, наприклад, як зовнішній файл hello.js - і готово.

**Якщо підключити подібний скрипт на сторінку «як є», то можливий конфлікт зі змінними, які вона використовує.**

Тобто, при підключенні до такої сторінці він її «зламає»:

<script>

var message = "Будь ласка, натисніть на кнопку";

</script>

<script src="hello.js"></script>

<button>Кнопка</button>

<script>

// очікується повідомлення зі змінної вище...

alert( message ); // але насправді буде виведено "Привіт"

</script>

Автор сторінки очікує, що бібліотека "hello.js" просто відпрацює, без побічних ефектів. А вона разом з цим перевизначила message в "Привіт".

Якщо ж прибрати скрипт hello.js, то сторінка буде виводити правильне повідомлення.

Знаючи внутрішній устрій hello.js нам, звичайно, зрозуміло, що проблема виникла через те, що змінна message з скрипта hello.js перезаписала оголошену на сторінці.

**Прийом проектування «Модуль»**

Щоб проблеми не було, всього лише потрібно, щоб у скрипта була своя власна область видимості, щоб його змінні не потрапили на сторінку.

Для цього ми загорнем весь його вміст в функцію, яку тут же запустимо.

Файл hello.js, оформлений як модуль:

(function() {

// глобальна змінна нашого скрипта

var message = "Привіт";

// функція для виводу цьої змінної

function showMessage() {

alert( message );

}

// виводимо повідомлення

showMessage();

})();

Цей скрипт при підключенні до тієї ж сторінки працюватиме коректно.

Виводитиметься «Привіт», а потім «Будь ласка, натисніть на кнопку".

**Навіщо дужки навколо функції?**

В наведеному вище прикладі оголошення модуля виглядає так:

(function() {

alert( "оголошуємо локальні змінні, функції, працюємо" );

// ...

}());

На початку і в кінці стоять дужки, бо інакше була б помилка.

Ось, для порівняння, невірний варіант:

function() {

// буде помилка

}();

Помилка при його запуску відбудеться тому, що браузер, бачачи ключове слово function в основному потоці коду, спробує прочитати Function Declaration, а тут імені немає.

Втім, навіть якщо ім'я поставити, то працювати теж не буде:

function work() {

// ...

}(); // syntax error

**Справа в тому, що «на місці» дозволено здійснювати виклики лише Function Expression.**

Загальне правило таке:

Якщо браузер бачить function в основному потоці коду - він вважає, що це Function Declaration.

Якщо ж function йде в складі більш складного виразу, то він вважає, що це Function Expression.

Для цього і потрібні дужки - показати, що у нас Function Expression, який за правилами JavaScript можна викликати «на місці».

Можна показати це іншим способом, наприклад поставивши перед функцією оператор:

+function() {

alert('Виклик на місці');

}();

!function() {

alert('Так теж буде працювати');

}();

**Експорт значення**

Прийом «модуль» використовується майже у всіх сучасних бібліотеках.

Адже що таке бібліотека? Це корисні функції, заради яких її підключають, плюс тимчасові змінні і допоміжні функції, які бібліотека використовує всередині себе.

Подивимося, наприклад, на бібліотеку lodash, хоча могли б і jQuery, там майже те ж саме.

Якщо її підключити, то з'явиться спеціальна змінна lodash (коротка назва\_), яку можна використовувати як функцію, і крім того в неї записані різні корисні властивості, наприклад:

* \_.defaults(src, dst1, dst2...) - копіює в об'єкт src ті властивості з об'єктів dst1, dst2 та інших, яких там немає.
* \_.cloneDeep(obj) - робить глибоке копіювання об'єкта obj, створюючи повністю незалежний клон.
* \_.size(obj) - повертає кількість властивостей в об'єкті, поліморфна функція: можна передати масив або навіть 1 значення.

Є й багато інших функцій, докладніше описаних в документації.

Приклад використання:

<p>Підключимо бібліотеку</p>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script

<p>Функція <code>\_.defaults()</code> додає відсутні властивості.</p>

<script>

var user = {

name: 'Вася'

};

\_.defaults(user, {

name: 'Не вказано',

employer: 'Не вказано'

});

alert( user.name ); // Вася

alert( user.employer ); // Не вказано

alert( \_.size(user) ); // 2

</script>

Тут нам не важливо, які функції або методи бібліотеки використовуються, нас цікавить саме як описана ця бібліотека, як в ній застосовується прийом «модуль».

Ось приблизний витяг з вихідного файлу:

;(function() {

// lodash - основна функція для бібліотеки

function lodash(value) {

// ...

}

// допоміжна змінна

var version = '2.4.1';

// ... інші допоміжні змінні та функції

// код функції size, поки що доступні тільки всередині

function size(collection) {

return Object.keys(collection).length;

}

// присвоїмо в lodash size інші функції, які потрібно винести з модулю

lodash.size = size

// lodash.defaults = ...

// lodash.cloneDeep = ...

// "експортувати" lodash назовні з модуля

window.\_ = lodash; // в оригінальному коді тут складніше, але смисл той же

}());

Усередині зовнішньої функції відбувається що завгодно, оголошуються свої локальні змінні, функції.

У window виноситься те, що потрібно зовні.

Технічно, ми могли б винести в window не тільки lodash, а й взагалі всі об'єкти і функції. На практиці, як раз навпаки, все ховають всередині модуля, глобальну область щоб уникнути конфліктів зберігають максимально чистою.

**Навіщо крапка з комою на початку?**

На початку коду вище знаходиться крапка з комою ; - це не помилка, а особливий «захист від дурнів».

Якщо вийде, що кілька JS-файлів об'єднані в один (і, швидше за все, стиснуті мініфікатором, але це не важливо), і програміст забув поставити крапку з комою, то буде помилка.

Наприклад, перший файл a.js:

var a = 5

Другий файл lib.js:

(function() {

// без крапки з комою на початку

})()

Після об'єднання в один файл:

var a = 5

// бібліотека

(function() {

// ...

})();

При запуску буде помилка, тому що інтерпретатор перед дужкою сам не вставить крапку з комою. Він просто зрозуміє код як var a = 5(function ...), тобто намагається викликати число 5 як функцію.

Такі правила мови, і тому рекомендується явно ставити крапку з комою. В даному випадку автор lodash ставить ; перед функцією, щоб попередити цю помилку.

**Експорт через return**

Можна оформити модуль і трохи по-іншому, наприклад передати значення через return:

var lodash = (function() {

var version;

function assignDefaults() { ... }

return {

defaults: function() { }

}

})();

Тут, до речі, дужки навколо зовнішньої function() { ... } не обов'язкові, адже функція і так оголошена всередині виразу присвоювання, а значить - є Function Expression.

Проте, краще їх ставити, для поліпшення читаності коду, щоб було відразу видно, що це не просте присвоєння функції.

**Висновок**

Модуль за допомогою замикань - це обертання пакета функціоналу в єдину зовнішню функцію, яка тут же виконується.

Всі функції модуля матимуть доступ до інших змінним і внутрішніх функцій цього ж модуля через замикання.

Наприклад, defaults  з прикладу вище має доступ до assignDefaults.

Але зовні програміст, що використовує модуль, може звертатися безпосередньо тільки до тих змінним і функцій, які експортовані. Завдяки цьому будуть приховані внутрішні аспекти реалізації, які потрібні тільки розробнику модуля.

Можна придумати і багато інших варіацій такого підходу. Зрештою, «модуль» - це всього лише функція-обгортка для приховування змінних.

**2. Управління пам'яттю в JavaScript**

Управління пам'яттю в JavaScript зазвичай відбувається непомітно. Ми створюємо примітиви, об'єкти, функції ... Все це займає пам'ять.

Що відбувається з об'єктом, коли він стає «не потрібен»? Чи можливо «переповнення» пам'яті? Для відповіді на ці питання - заліземо «під капот» інтерпретатора.

**Управління пам'яттю в JavaScript**

Головною концепцією управління пам'яттю в JavaScript є принцип досяжності (англ. reachability).

1. Певний набір значень вважається досяжним спочатку, зокрема:

* Значення, посилання на які містяться в стеку виклику, тобто - всі локальні змінні і параметри функцій, які зараз виконуються або знаходяться в очікуванні закінчення вкладеного виклику.
* Всі глобальні змінні.

Ці значення гарантовано зберігаються в пам'яті. Ми будемо називати їх *корінням*.

1. **Будь-яке інше значення зберігається в пам'яті лише до тих пір, поки є з кореня по посиланню або ланцюжку посилань.**

Для очищення пам'яті від недосяжних значень в браузерах використовується автоматичний Збирач сміття (англ. Garbage collection, GC), вбудований в інтерпретатор, який спостерігає за об'єктами і час від часу видаляє недосяжні.

Найпростіша ситуація тут з примітивами. При присвоєнні вони копіюються цілком, посилань на них не створюється, так що якщо у змінній був один рядок, а його замінили на інший, то попередній можна сміливо викинути.

Саме об'єкти вимагають спеціального «збирача сміття», який спостерігає за посиланнями, так як на один об'єкт може бути багато посилань з різних змінних і, при перезапису однієї з них, об'єкт може бути все ще доступний з іншої.

Далі ми подивимося ряд прикладів, які допоможуть в цьому розібратися.

Досяжність і наявність посилань

Є одне спрощення для роботи з пам'яттю: «значення залишається в пам'яті, поки на нього є хоча б одне посилання».

Але таке спрощення буде вірним лише в одну сторону.

* **Вірно - в тому плані, що якщо посилань на значення немає, то пам'ять з-під нього очищається.**

Наприклад, було створено посилання в змінній, і цю змінну тут же перезаписали:

var user = {

name: "Вася"

};

user = null;

Тепер об'єкт { name: "Вася" } більше недоступний. Пам'ять буде звільнена.

* **Невірно - в іншу сторону: наявність посилання не гарантує, що значення залишиться в пам'яті.**

Така ситуація виникає з об'єктами, які посилаються один на одного:

var vasya = {};

var petya = {};

vasya.friend = petya;

petya.friend = vasya;

vasya = petya = null;

Незважаючи на те, що об'єкти vasya і petya посилаються один на одного через посилання friend, тобто можна сказати, що на кожен з них є посилання, останній рядок робить ці об'єкти в сукупності недосяжними.

Тому вони будуть видалені з пам'яті.

Тут як раз і грає роль «досяжність» - обидва цих об'єкти стають недосяжні з коренів, в першу чергу, з глобальної області, стека.

Збирач сміття відстежує такі ситуації і очищає пам'ять.

**Алгоритм збирання сміття**

Збирач сміття йде від кореня по посиланнях і запам'ятовує всі знайдені об'єкти. По закінченню - він дивиться, які об'єкти в ньому відсутні і видаляє їх.

Наприклад, розглянемо приклад об'єкта «сім'я»:

function marry(man, woman) {

woman.husband = man;

man.wife = woman;

return {

father: man,

mother: woman

}

}

var family = marry({

name: "Василь"

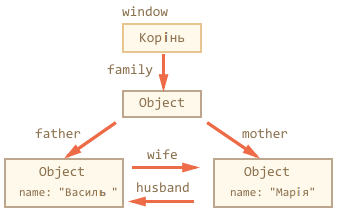
}, {

name: "Марія"

});

Функція marry приймає два об'єкти, дає їм посилання один на одного і повертає третій, що містить посилання на обидва.

Одержаний об'єкт family можна зобразити так:



Тут стрілочками показані посилання, а ось властивість name посиланням не є, там зберігається примітив, тому воно всередині самого об'єкта.

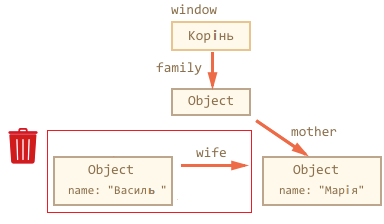
Щоб запустити збирач сміття, видалимо два посилання:

delete family.father;

delete family.mother.husband;

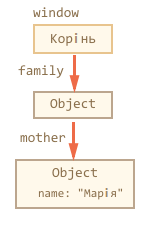
Звернемо увагу, видалення тільки одного з цих посилань ні до чого б не привело. Поки до об'єкта можна дістатися з кореня window, об'єкт залишається живий.

А якщо два, то виходить, що від колишнього family.father посилання виходять, але в нього - ні одна не йде:



**Абсолютно неважливо, що з об'єкта виходять якісь посилання, вони не впливають на досяжність цього об'єкта.**

Колишній family.father став недосяжним і буде видалений разом зі своїми даними, які також більш недоступні з програми.

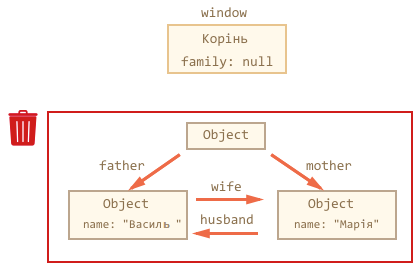


А тепер - розглянемо більш складний випадок. Що буде, якщо видалити головне посилання family?

Вихідний об'єкт - той же, що і на початку, а потім:

window.family = null;

Результат:



Як бачимо, об'єкти в конструкції все ще пов'язані між собою. Однак, пошук від кореня їх не знаходить, вони не досяжні, і значить збирач сміття видалить їх з пам'яті.

**Оптимізації**

Проблема описаного алгоритму - у великих затримках. Якщо об'єктів багато, то на пошук всіх досяжних піде досить багато часу. Але ж виконання скрипта при цьому має бути зупинено, вже проскановані об'єкти не повинні помінятися до закінчення процесу. Виходять невеликі, але неприємні паузи-зависання в роботі скрипта.

Тому сучасні інтерпретатори застосовують різні оптимізації.

Найчастіша - це поділ об'єктів на два види «старі» і «нові». Для кожного типу виділяється своя область пам'яті. Кожен об'єкт створюється в «новій» області та, якщо прожив досить довго, мігрує в стару. «Нова» область зазвичай невелика. Вона очищується часто. «Стара» - рідко.

На практиці виходить ефективно, зазвичай більшість об'єктів створюються і вмирають майже відразу, наприклад, служачи локальними змінними функції:

function showTime() {

alert( new Date() ); // цей об’єкт буде створений і помре відразу

}

**Замикання**

Об'єкти змінних, про які йшла мова раніше, в розділі про замикання, також схильні до збору сміття. Вони йдуть тим же правилам, що і звичайні об'єкти.

Об'єкт змінних зовнішньої функції існує в пам'яті до тих пір, поки існує хоч одна внутрішня функція, котра посилається на нього через властивість [[Scope]].

Наприклад:

* Зазвичай об'єкт змінних видаляється після закінчення роботи функції. Навіть якщо в ньому є оголошення внутрішньої функції:

function f() {

var value = 123;

function g() {} // g видно тільки всередині

}

f();

У коді вище value і g є властивостями об'єкта змінних. Під час виконання f() її об'єкт змінних знаходиться в поточному стеку виконання, тому живий. По закінченню, він стане недосяжним і буде прибраний з пам'яті разом з іншими локальними змінними.

* ... А ось в цьому випадку лексичне оточення, включаючи змінну value, буде збережено:

function f() {

var value = 123;

function g() {}

return g;

}

var g = f(); // функція g буде жити і збереже посилання на об’єкт змінних

У прихованому властивості g.[[Scope]] знаходиться посилання на об'єкт змінних, в якому була створена g. Тому цей об'єкт змінних залишиться в пам'яті, а в ньому - і value.

* Якщо f() буде викликатися багато разів, а отримані функції будуть зберігатися, наприклад, складатися в масив, то будуть зберігатися і об'єкти LexicalEnvironment з відповідними значеннями value:

function f() {

var value = Math.random();

return function() { return value; };

}

// 3 функції, кожна посилається на свій об’єкт змінних,

// кожний зі своїм значенням value

var arr = [f(), f(), f()];

Об'єкт LexicalEnvironment  живе рівно до тих пір, поки на нього існують посилання. У коді нижче після видалення посилання на g вмирає:

function f() {

var value = 123;

function g() {}

return g;

}

var g = f(); // функція g жива

// а значить в пам’яті лишається відповідний об’єкт змінних f()

g = null; // ..а ось тепер пам’ять буде очищена

**Оптимізація в V8 і її наслідки**

Сучасні JS-движки роблять оптимізації замикань по пам'яті. Вони аналізують використання змінних і в разі, коли змінна з замикання абсолютно точно не використовується, видаляють її.

У коді вище змінна value ніяк не використовується. Тому вона буде видалена з пам'яті.

**Важливий побічний ефект в V8 (Chrome, Opera) полягає в тому, що віддалена змінна стане недоступна і при налагодженні!**

Спробуйте запустити приклад нижче з відкритою консоллю Chrome. Коли він зупиниться, в консолі наберіть alert(value).

function f() {

var value = Math.random();

function g() {

debugger; // виконайте в консолі alert( value ); Нема такої змінної!

}

return g;

}

var g = f();

g();

Як ви могли побачити - немає такої змінної! Недоступна вона зсередини g. Інтерпретатор вирішив, що вона нам не знадобиться і видалив.

Це може привести до кумедних казусів при налагодженні, аж до того що замість цієї змінної буде інша, зовнішня:

var value = "Сюрприз";

function f() {

var value = "саме близьке значення";

function g() {

debugger; // виконайте в консолі alert( value ); Сюрприз!

}

return g;

}

var g = f();

g();

**Вплив управління пам'яттю на швидкість**

На створення нових об'єктів і їх видалення витрачається час. Це важливо мати на увазі в разі, коли важлива продуктивність.

Як приклад розглянемо рекурсію. При вкладених викликах кожен раз створюється новий об'єкт зі змінними і поміщається в стек. Потім пам'ять з-під нього потрібно очистити. Тому рекурсивний код буде завжди повільніше використовувати цикл, але наскільки?

Приклад нижче тестує складання чисел до даного через рекурсію в порівнянні зі звичайним циклом:

function sumTo(n) { // звичайний цикл 1+2+...+n

var result = 0;

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

result += i;

}

return result;

}

function sumToRec(n) { // рекурсія sumToRec(n) = n+SumToRec(n-1)

return n == 1 ? 1 : n + sumToRec(n - 1);

}

var timeLoop = performance.now();

for (var i = 1; i < 1000; i++) sumTo(1000); // цикл

timeLoop = performance.now() - timeLoop;

var timeRecursion = performance.now();

for (var i = 1; i < 1000; i++) sumToRec(1000); // рекурсія

timeRecursion = performance.now() - timeRecursion;

alert( "Різниця в " + (timeRecursion / timeLoop) + " раз" );

Різниця в швидкості на такому прикладі може становити, залежно від інтерпретатора, 2-10 разів.