Міністерство освіти і науки України
Дніпровський національний університет
ім. Олеся Гончара

М. М. Ясько

ОСНОВИ МОВИ JavaScript
Посібник

Ухвалено на вченій раді
протокол №8
від 22 грудня 2016 р.

Дніпро
РВВ ДНУ
2017

УДК 004.451
Я 86

Рецензенти: д-р техн. наук, проф. О.І. Міхальов
д-р техн. наук, проф. О.Г. Байбуз

Я 86 Ясько, М.М. Основи мови JavaScript [Текст]: посібник / М.М. Ясько. – Д.: РВВ ДНУ, 2017. – 48 с.

Розглянуто основні поняття мови JavaScript, типи даних; наведено приклади програм із детальними поясненнями.

Для студентів факультету прикладної математики ДНУ, які вивчають інформатику та комп'ютерну техніку, а також усіх, хто бажає самостійно вивчати сучасні веб-технології.

Темплан 2017, поз. 6

Навчальне видання

Микола Миколайович Ясько

Основи мови JavaScript
Посібник

Редактор Л.В. Дмитренко
Техредактор Т.І. Севост'янова
Коректор О.В. Бец

Підписано до друку 15.08.2017. Формат 60x84/16. Папір друкарський.
Друк плоский. Ум. друк. арк. 2,8. Ум. фарбовідб. 2,8. Обл.– вид. арк. 3,7.
Тираж 10 пр. Зам. № .

РВВ ДНУ, просп. Гагаріна, 72, м. Дніпро, 49010.
ПП «Ліра ЛТД», вул. Погребняка, 25, м. Дніпро, 49010.
Свідоцтво про внесення до Державного реєстру
серія ДК №188 від 19.09.2000р. Фактична адреса: вул. Наукова, 5

Вступ

JavaScript — це порівняно нова мова для написання сценаріїв, розроблена компанією «Netscape». Із її допомогою можна створювати інтерактивні web-сторінки найбільш зручним і ефективним способом.

JavaScript — назва реалізації стандарту мови програмування ECMAScript, заснованої на принципах прототипного програмування. Найпоширеніше і найвідоміше застосування мови — написання сценаріїв для веб-сторінок, утім її також застосовують для впровадження сценаріїв керування об'єктами, вбудованими в інші програми.

Багато людей вважають, що мова JavaScript — це та сама мова Java, адже вони мають подібні імена. Однак це не так. Не будемо зараз зупинятися на існуючих відмінностях, важливо лише пам'ятати, що JavaScript та Java — різні мови, хоча в них є багато спільного.

Найчастіше JavaScript застосовують як частину браузера, що дозволяє можливість коду на боці клієнта (виконуваному на пристрої кінцевого користувача) взаємодіяти із користувачем, керувати браузером, асинхронно обмінюватися даними із сервером, змінювати структуру та зовнішній вигляд веб-сторінки. Мову JavaScript також застосовують для програмування на боці сервера (подібно до таких мов програмування, як PHP, Java і C#), розробки ігор, стаціонарних і мобільних застосунків, сценаріїв у прикладному програмному забезпеченні (наприклад, у програмах що входять до складу Adobe Creative Suite), усередині PDF-документів тощо. Крім того, JavaScript має деякі властивості, притаманні функціональним мовам, — функції як об'єкти першого рівня, об'єкти як списки, каррінг (currying), анонімні функції, замикання (closures), що надає їй додаткової гнучкості.

JavaScript має C-подібний синтаксис, але порівняно із мовою Сі володіє важливими відмінностями:

- об'єкти із можливістю інтроспекції і динамічної зміни типу через механізм прототипів;
- функції як об'єкти першого класу;
- можливість обробки винятків;
- автоматичне перетворення типів;
- автоматичне прибирання «сміття»;
- анонімні функції.

У листопаді 1996 р. компанія «Netscape» заявила, що відправила JavaScript в організацію «Ecma International» для розгляду мови як промислового стандарту. У результаті подальшої роботи з'явилася стандартизована мова — ECMAScript. У червні 1997 р. «Ecma International» опублікувала першу редакцію специфікації ECMA-262. Рік по тому, у червні 1998 р., з метою адаптувати специфікацію до стандарту ISO/IEC-16262, було внесені деякі зміни і випущено другу редакцію. Третя редакція побачила світ у грудні 1999 р. Четверту версію

стандарту ECMAScript так і не було закінчено і четверта редакція не вийшла. Утім п’яту редакцію було випущено в грудні 2009 р.

Шосте видання, відоме як ES6 або ECMAScript 2015, має значний за обсягом новий синтаксис для написання великих програм, у тому числі класів і модулів, але і визначає їх семантично в тих же умовах, що і ECMAScript 5 суворого режиму (strict mode). Інші нові можливості включають ітератори, функцій зі стрілками, типізовані масиви, обіцянки (promise) тощо. Шосте видання також відоме як перша специфікація «ECMAScript Harmony» або «ES6 Harmony».

Сьоме видання ES7 або ECMAScript 2016 призначене для продовження теми мовної реформи, ізоляції коду, контролю ефектів і бібліотек/інструментів, що були введені у стандарт ES6. Нові функції включають паралелізм і Atomics, передачу двійкових даних, більшу кількість і математичних удосконалень, синтаксичну інтеграцію із обіцянками, які використовують для асинхронного програмування, типів SIMD, краще метапрограмування із класами, властивостями класу і, наприклад, перевантаження операторів, записи і кортежі і т.д.

Зауважимо, що в сучасних рушіях JavaScript більшість можливостей ES6 і ES7 підтримуються тільки в разі увімкненого суворого режиму (`"use strict"`).

Формат програми

Програма на JavaScript — просто набір операторів, які розділяють крапкою з комою. Після останнього оператора в рядку крапку з комою можна не ставити.

Оскільки JavaScript не має жорстких вимог до форматування тексту програми, можна довільно вставляти символи переведення рядка і відступу для кращої читабельності тексту.

Код програми на мові JavaScript розміщують або всередині HTML-сторінки за допомогою тегу `<script>`, або в окремому текстовому файлі.

Досить часто програму JavaScript називають скриптом або сценарієм. Скрипти виконуються у результаті того, що відбулася деяка подія, пов’язана із HTML-сторінкою. У багатьох випадках настання вказаних подій ініціює користувач.

Змінні та вирази JavaScript можна застосовувати як значення параметрів тегів HTML. У цьому випадку елементи JavaScript розміщують між амперсандом (&) та крапкою з комою (;), але вони мають бути обмежені фігурними дужками `{}` і їх слід застосовувати тільки як значення параметрів тегів. Наприклад, нехай визначена змінна `c` і їй присвоєно значення `green`. Наступний тег буде виводити текст зеленого кольору :

```
<font color="&{c};">текст зеленого кольору</font>
```

У програмі на мові JavaScript можна залишати коментарі, які ігнорує JavaScript-інтерпретатор і які слугують поясненням для розробників.

Однорядкові коментарі починаються із символів `//`. Текст, починаючи із цих символів і до кінця рядка, вважають коментарем. Складний коментар розміщується між символами `/*` і `*/` і може нараховувати кілька рядків.

Константи й літерали

Літерали — способи, за допомогою яких наводять значення в JavaScript.

Літерал (англ. literal — буквальний) — постійне значення певного типу даних, записане у вихідному коді комп'ютерної програми. Виділяють різні типи літералів: логічні (булеві), числові, символьні, літерали масивів та об'єктів.

Логічні літерали набувають двох значень — `true` («істина») або `false` («хибність»).

Числові літерали застосовують для запису чисел:

```
 1  -10 3.1415925 1E-4 3.2E+10  0xFF
```

Символьні літерали зазвичай являють собою рядок символів, записаний у лапках

```
"Це символьний рядок\n"
'А це такий самий рядок'
`А це фрагмент   тексту,
  що також являє собою символьний рядок.
  У ньому можна підставляти значення змінних,
 наприклад x=${x}
`
```

У мові JavaScript допускається застосування виразів, які слугують масивами-літералами та об'єктами-літералами. Наприклад:

```
[1,2,3,4,5,"Nick"] // Ініціалізатор масиву
{x:1, y:2} // Ініціалізатор об'єкта
функціональний літерал
function(x){return x*x;}  // Літерал функції
(x,y)=>x+y  // Це також літерал функції
```

У JavaScript регулярні вирази являють собою літерали особливого типу:

```
 /ac+df/g
 /^Section*\d+)/gm;
```

Операції

Виділяють 3 типи операцій:

- **унарні** — потребують одного операнда, наприклад `++i;`
- **бінарні** — потребують двох операндів, установлюють між ними, наприклад `a*b ;`
- **тернарні** — потребують трьох операндів: `a>b?a:b.`

Тернарний оператор є аналог умовного оператора `if...else`.

JavaScript містить великий набір убудованих операцій, які можна поділити на декілька груп.

Арифметичні операції

+ — додавання;

- — віднімання;

***** — множення;

/ — ділення;

% — залишок від ділення;

++ — інкремент;

— — декремент.

Операцію додавання + можна також застосовувати для конкатенації символьних рядків.

Додавання двох чисел повертає їх суму, але додавши число і символьний рядок, одержимо рядок:

```
x = 5 + 5;              // 10
y = "5" + 5;            // 55
y = Number("5") + 5;    // 10
z = "Hello" + 5;        // Hello5
```

Операції присвоєння

Операція	Приклад	Аналог
=	X = y	x = y
+=	X += y	x = x + y
-=	X -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

```
var list = [ 1, 2, 3 ]
var [ a, , b ] = list
```

```
[ b, a ] = [ a, b ]    // тут можлива помилка
```

Операції порівняння

Операція	Опис	Приклад	Результат
==	порівняння	"1"==1	true
===	порівняння значення і типу	"1"===1	false
!=	не дорівнює	"2"!=2	false
!==	різні значення або тип	"2"!==2	false
>	більше	2>1	true
<	менше	2<1	false
≥=	більше чи дорівнює	2≥2	true
≤=	менше чи дорівнює	3≤2	false
?	тернарний оператор	true?1:0	1

Логічні операції

Для операцій над логічними значеннями в JavaScript застосовують || (АБО), && (І) і ! (НЕ). Хоча їх і називають «логічними», але в JavaScript вони застосовні до значень будь-якого типу і можуть повертати значення будь-якого типу.

Операція	Приклад	Результат
||	a || b	Повертає a, якщо воно може бути перетворено до true; в іншому випадку повертає b. Таким чином, у разі виклику із булевими значеннями, поверне true тільки в тому випадку, коли хоча б один операнд правдивий, інакше поверне false.
&&	a && b	Повертає a, якщо воно може бути перетворене до false; в іншому випадку повертає b. Отже, у разі виклику із булевими значеннями, поверне true тільки якщо обидва операнди істинні, інакше false.
!	!a	Повертає false, якщо операнд може бути перетворений на true, в іншому випадку повертає true.

Оскільки логічні вирази обчислюють зліва направо, їх перевіряють на можливість «скорочених обчислень» за такими правилами:

```
false && що завгодно  // стає false
```

```
true || що завгодно   // стає true
```

Ці правила логіки гарантують, що результат завжди буде правильний, але зайвих обчислень при цьому не проводять. А отже, не обчислені вирази не дадуть побічних ефектів.

Побітові оператори JavaScript

Побітові оператори інтерпретують операнди як послідовність із 32 бітів (нулів і одиниць). Вони здійснюють операції, застосовуючи двійкове зображення числа і повертають нову послідовність із 32 біт (число) як результат.

Операція	Приклад	Опис
&	`a & b`	Ставить 1 на кожен біт результату, для якого відповідні біти операндів дорівнюють 1.
|	`a | b`	Ставить 1 на кожен біт результату, для якого хоча б один із відповідних бітів операндів дорівнює 1.
^	`a ^ b`	Ставить 1 на кожен біт результату, для якого тільки один (але не обидва) із відповідних бітів операндів дорівнює 1.
~	`~ a`	Замінює кожен біт операнда на протилежний.
<<	`a<<n`	Зрушує двійкове зображення `a` на `n` бітів вліво, додаючи справа нулі.
>>	`a>>n`	Зрушує двійкове зображення `a` на `n` бітів вправо, відкидаючи зрушувані біти.
>>>	`a>>>n`	Зрушує двійкове зображення `a` на `n` бітів вправо, відкидаючи зрушувані біти і додаючи нулі зліва.

Операції визначення типу

Операція	Опис
`typeof`	Повертає тип змінної (number, boolean, string, object).
`instanceof`	Повертає `true`, якщо об’єкт є екземпляр даного типу об’єкта.

Змінні й типи даних

Ім’я змінної — ідентифікатор, його застосовують:

- для імен змінних;
- імен констант;
- імен функцій;
- міток.

Мова JavaScript чутлива до регістру символів. Правило створення імен таке:

- першим символом в імені має бути буква `A-Za-z` або символи `$ _`;
- наступні символи можуть бути буквами, цифрами або символами `$ _`;
- зарезервовані слова не можуть бути іменами.

Список зарезервованих імен

Зарезервоване слово не може бути застосоване як ідентифікатор.

```
abstract    arguments  boolean    break      byte
case        cath       char       class*     const
continue    debugger   default    delete     do
double      else       enum*      eval       export*
extends*    false      final      finally    float
for         function   goto       if        implements
import*     in         instanceof int       interface
let         long       native     new         null
package     private    protected  public     return
short       static     super*     switch synchronized
this        throw      throws     transient  true
try         typeof     var        void       volatile
while       with       yield
```

Також слід уникати застосування імен вбудованих об’єктів, властивостей і методів JavaScript:

```
Array            Date       eval       function
hasOwnProperty   Infinity   isFinite   isNaN
isPrototypeOf    length     Math       NaN
name             Number     Object     prototype
String           toString   undefined  valueOf
```

Декларування змінних

Оператор `var` оголошує змінні з можливістю їх ініціалізації:

```
var count;  // оголошення однієї змінної
var count, amount, level;  // оголошення декількох змінних
```

```
var count = 0, amount = 100;  // оголошення з ініціалізацією
```

Оператор `let` оголошує змінні в рамках блока, обмеженого фігурними дужками з можливістю їх ініціалізації:

```
var x=1
{
 let x=5
}
```

Оператор `const` оголошує іменовані константи:

```
const c=2, gamma=1.4
gamma = 1.3  // помилка
```

Константи оголошують у межах блока, так само, як і змінні, визначені за допомогою оператора `let`. Значення іменованої константи не може змінитися через переприсвоєння, і вона не може бути повторно оголошена.

Мова JavaScript містить декілька типів даних (із уведенням нових стандартів кількість типів даних постійно зростає).

Прості типи:

- `String` — рядковий;
- `Number` —числовий;
- `Boolean` — логічний.

Композитні типи:

- `Array` — масив;
- `Int8Array, Float32Array`, ... — типізовані масиви;
- `ArrayBuffer` — спеціальний буфер;
- `Object` — об’єктний.

Спеціальні типи:

- `null` — нульовий;
- `undefined` — невизначений.

Однак відносно невелика кількість типів дозволяє створювати повноцінні сценарії для виконання багатьох функцій.

Тип змінної залежить від того, який тип інформації в ній зберігається. JavaScript не є жорстко типізована мова. Це означає, що не потрібно точно визначати тип змінної у момент її створення. Тип змінної привласнюється змінній автоматично протягом виконання скрипту. Так, можна визначити змінну в такий спосіб:

```
var answer=41
```

Пізніше можливо привласнити даній змінній значення іншого типу:

```
answer="Thanks for all the meat..."
```

Рушій JavaScript сам виконує автоматичне перетворення типів даних за необхідності. Для явного перетворення типів використовуються методи `Boolean`, `Number`, `Object` та `String`.

Вирази

Вирази записують за звичайними правилами, поширеними в більшості мов програмування.

```
var anExpression = 3 * (4. / 5) + 6;
var aSecondExpression = Math.PI * radius * radius;
var aThirdExpression = aSecondExpression + "%" +
anExpression;
var aFourthExpression = "(" + aSecondExpression + ")
% (" + anExpression + ")";
```

У разі виходу із діапазону, втрати значущих розрядів або ділення на нуль JavaScript не видає помилку. Якщо результат буде занадто великий і вийде з діапазону, то буде повернуто спеціальне значення «нескінченність» — `Infinity`. Втрата значущих розрядів: результат арифметичної операції виявляється дуже близьким до нуля. Якщо все ж втрата була, то буде повернуто 0 (нуль).

Глобальне значення `NaN` означає «не число». Є одна особливість у цього значення, операція перевірки на рівність (==) завжди повертає негативний результат, навіть якщо його порівнювати з самим собою. Щоб визначити, чи є значення змінної `x` значенням `NaN`, можна застосувати конструкцію

```
if (x != x) { ... }.
```

Оператори мови JavaScript

Оператор — це інструкція певної мови програмування, за допомогою якої задають деякий крок процесу обробки інформації на ЕОМ. Оператори застосовують для керування потоком команд у JavaScript. Один оператор може бути розбитий на кілька рядків, або, навпаки, в одному рядку може міститись декілька операторів.

Необхідно пам’ятати:

- що блоки операторів, такі як визначення функцій, мають бути узяті у фігурні дужки;
- крапку з комою застосовують для розділення окремих операторів.

JavaScript має набір вбудованих операторів, що керують логікою виконання програм.

Умовний оператор

```
if (умова) оператор1 [ else оператор2]
```

Оператор вибору

Оператор вибору `switch` виконує ту чи іншу послідовність операторів залежно від значення певного виразу. Він має вигляд

```
switch (вираз) {
  case значення:
    оператори
    break;
  case значення:
    оператори
    break;
  ...
  default:
    оператори
}
```

Тут вираз — це будь-який вираз, значення — можливе значення виразу, а оператори — будь-які групи операторів JavaScript. Оператор вибору спочатку обчислює значення виразу, а потім перевіряє, чи немає цього значення в одній із міток `case`. Якщо таке значення є, то виконуються оператори за цією міткою; якщо немає, то виконуються оператори за міткою `default`, якщо вона відсутня, то керування передається оператору, наступному за міткою `default`.

Необов'язковий оператор `break` вказує, що після виконання операторів керуванння передається оператору, наступному за `switch`. Якщо `break` відсутній, то після виконання операторів починають виконуватися оператори, які стоять після наступної мітки case (керування ніби «провалюється» у наступну мітку).

```
var gender = "female";
switch (gender) {
  case "female":
    console.log("Hello, miss!");
    break;
  case "male":
    console.log("Hello, sir!");
    break;
  default:
    console.log("Hello!");
}
```

Оператори циклу

Цикл — це послідовність операторів, виконання якої повторюється до тих пір, поки певна умова не стане хибною. JavaScript містить три оператори циклу: `for`, `while` і `do...while`, а також оператори `break` і `continue`, які застосовують усередині циклів. Близький до операторів циклу і оператор ітерації `for ... in`, який застосовують під час роботи з об'єктами.

Мітки операторів

Будь-який оператор або блок операторів у сценарії на мові JavaScript може мати довільну кількість міток. Їх можуть застосовувати оператори `break` і `continue` для позначення оператора до якого вони належать. Позначений оператор має вигляд

```
мітка: оператор
```

Як мітку можна застосовувати будь-який ідентифікатор, який не є зарезервованим словом.

Оператор `break` перериває виконання поточного циклу, оператора `switch` або позначеного оператора і передає керування оператору, наступному за перерваним. Цей оператор можна застосовувати тільки всередині циклів `while`, `do ... while`, `for` або `for...in`, а також всередині оператора `switch`. Він має дві форми:

```
break
break мітка
```

Перша форма оператора перериває виконання найбільш внутрішнього із циклів або операторів `switch`. Друга форма оператора перериває виконання оператора із заданою міткою.

Оператор `continue` завершує поточну ітерацію циклу або циклу, позначеного відповідною міткою, і починає нову ітерацію. Даний оператор застосовують тільки всередині циклів `while`, `do…while`, `for` або `for…in`. Він також має дві форми:

```
continue
continue мітка
```

Оператор `for…in`

Оператор `for…in` виконує задані дії для кожної властивості об’єкта або для кожного індексу масиву. Він має вигляд

```
for (змінна in вираз) оператор
```

У даному випадку змінна — це декларація змінної, вираз — будь-який вираз, значенням якого є об’єкт або масив, оператор — будь-яка група операторів JavaScript; якщо ця група містить більше одного оператора, то її слід узяти у фігурні дужки `{}`.

Оператор `for…of`

Оператор `for…of` виконує задані дії для кожного значення властивості об’єкта або для кожного елемента масиву. Він має вигляд

```
for (змінна in вираз) оператор ,
```

де змінна — це декларація змінної, вираз — будь-який вираз, значенням якого є об’єкт або масив, оператор — будь-яка група операторів JavaScript; якщо дана група містить більше одного оператора, то її потрібно узяти у фігурні дужки `{}`. Наприклад

```
let arr = [3, 5, 7];
arr.foo = "hello";
for (let i in arr) {
   console.log(i); //  "0", "1", "2", "foo"
}
for (let i of arr) {
   console.log(i); //  "3", "5", "7"
}
```

Оператор `with`

Оператор with задає ім’я об’єкта за замовчуванням. Він має вигляд

```
with (вираз) оператор ,
```

де вираз — будь-який вираз, значенням якого є об’єкт, оператор — будь-яка група операторів JavaScript; якщо ця група містить більше одного оператора, то її слід узяти у фігурні дужки {}.

Даний оператор діє таким чином. Для кожного ідентифікатора в операторі виконуюча система перевіряє, чи він є ім’я властивості об’єкта, заданого за замовчуванням. Якщо це так, то даний ідентифікатор вважають ім’ям властивості, якщо ж ні, то ім’ям змінної. Оператор `with` застосовують для скорочення розміру програмного коду і прискорення доступу до властивостей об’єктів. Наприклад, для того, щоб не вказувати кожного разу ім’я об’єкта `Math` для доступу до математичних функцій:

```
with(Math){
  x = cos(PI / 4) + sin(LN10);
  y = tan(2 * E);
}
```

Наразі не рекомендується широке застосування даного оператора.

Об’єкт Math

`Math` — це вбудований об’єкт, який має властивості й методи для математичних констант і функцій. На відміну від інших глобальних об’єктів, Math не конструктор. Всі властивості і методи Math статичні. Користувач застосовує константу π як `Math.PI` і викликає функцію синуса як `Math.sin(x)`, де `x` — аргумент методу. Константи визначають із точністю, що відповідає типу `double` у мові С.

Властивості об’єкта Math

`Math.E`	Стала Ейлера — приблизно 2.718.
`Math.LN2`	Натуральний логарифм 2 — приблизно 0.693.
`Math.LN10`	Натуральний логарифм 10 — приблизно 2.303.
`Math.LOG2E`	log_2E — приблизно 1.443.
`Math.LOG10E`	$log_{10}E$ — приблизно 0.434.
`Math.PI`	Відношення довжини кола до його діаметра – приблизно 3.1415926.
`Math.SQRT1_2`	Квадратний корінь із 1/2 – приблизно 0.707.
`Math.SQRT2`	Квадратний корінь із 2 – приблизно 1.414.

Методи об’єкта Math

`Math.abs(x)`	Повертає абсолютну величину числа.
`Math.acos(x)`	Повертає арккосинус числа.
`Math.acosh(x)`	Повертає гіперболічний арккосинус числа.
`Math.asin(x)`	Повертає арксинус числа.
`Math.asinh(x)`	Повертає гіперболічний арксинус числа.
`Math.atan(x)`	Повертає арктангенс числа.
`Math.atanh(x)`	Повертає гіперболічний арктангенс числа.
`Math.atan2(y,x)`	Повертає арктангенс своїх аргументів.
`Math.cbrt(x)`	Повертає кубічний корінь числа.
`Math.ceil(x)`	Повертає найменше ціле число, що більше або дорівнює аргументу.
`Math.clz32(x)`	Повертає кількість провідних нулів 32-розрядного цілого числа.
`Math.cos(x)`	Повертає косинус числа.
`Math.cosh(x)`	Повертає гіперболічний косинус числа.
`Math.exp(x)`	Повертає E^x.
`Math.expm1(x)`	Повертає $exp(x)-1$.
`Math.floor(x)`	Повертає найбільше ціле число, що менше або дорівнює `x`.
`Math.fround(x)`	Повертає найбільш точне зображення даного числа.
`Math.hypot([x[,y[, …]]])`	Повертає корінь квадратний із суми квадратів аргументів.
`Math.imul(x,y)`	Повертає результат 32-розрядного цілого множення.
`Math.log(x)`	Повертає натуральний логарифм числа.
`Math.log1p(x)`	Повертає $log(x+1)$.
`Math.log10(x)`	Повертає логарифм числа за основою 10.
`Math.log2(x)`	Повертає логарифм числа за основою 2.
`Math.max([x[, y[, …]]])`	Повертає найбільше із чисел.
`Math.min([x[, y[, …]]])`	Повертає найменше із чисел.
`Math.pow(x, y)`	Повертає x^y.

`Math.random()`	Повертає псевдовипадкове число від 0 до 1.
`Math.round(x)`	Повертає значення числа, округлене до найближчого цілого.
`Math.sign(x)`	Повертає знак числа.
`Math.sin(x)`	Повертає синус числа.
`Math.sinh(x)`	Повертає гіперболічний синус числа.
`Math.sqrt(x)`	Повертає квадратний корінь числа.
`Math.tan(x)`	Повертає тангенс числа.
`Math.tanh(x)`	Повертає гіперболічний тангенс числа.
`Math.toSource()`	Повертає рядок "`Math`".
`Math.trunc(x)`	Повертає цілу частину числа.

Можна вважати, що всі математичні функції розміщені в «пакеті» `Math`.

Об’єкт Date

Об’єкт `Date` описує конкретний момент часу. Час у даному випадку вимірюють кількістю мілісекунд, що минули з 1 січня 1970 р. за Грінвічем (початок ери Unix). Приклади створення об’єкта:

```
new Date();
new Date(value);
new Date(dateString);
new Date(year, month[, day[, hour[, minutes[,
seconds[, milliseconds]]]]]);
```

Цей об’єкт має багато методів, які дозволяють визначити будь які значення: рік, день, годину тощо. Дані методи мають певні префікси, які полегшують їх запам’ятовування:

set – методи для встановлення значень;

get – методи для одержання дати та часу;

to – методи для одержання рядків із об’єкта;

parse і UTC — методи для перетворення рядків на дату (англійський формат дати).

Нумерація днів тижня: 0 — неділя, 6 — субота. Нумерація місяців: 0 — січень, 11 — грудень.

Робота з масивами

Масив — різновид об’єкта для зберігання пронумерованих значень, що має додаткові методи для зручного маніпулювання цими даними. Масиви зазвичай застосовують для зберігання впорядкованих колекцій даних (списку

товарів на сторінці, студентів у групі і т.д.). Приклади створення масивів наведено нижче:

```
var myArray = new Array ()  // Створення пустого масиву
myArray [0] = 'Київ'
myArray [1] = 'Дніпро'
var myArray = new Array (100) // Масив і резервування місця для
// 100 змінних
var prices = new Array(12,15,20) // Створення нового масиву із
// трьох елементів
var prices = [12, 15, 20]       // Аналогічно попередньому рядку
```

Усі масиви, незалежно від способу створення, являють собою екземпляри класу (об'єкта) `Array`. Додавання елементів здійснюють за допомогою ініціалізації відповідного елемента.

Масиви в JavaScript не обов'язково мають бути «суцільними», тобто містити всі елементи. За необхідності можна створювати так звані «розріджені» масиви: якщо ми спробуємо одержати значення неініціалізованого елемента, то одержимо `undefined`.

Цікава особливість JavaScript — масив може одночасно містити елементи різних типів, у тому числі й інші масиви.

Клас `Array` містить властивість `length`, що дозволяє обчислити поточну довжину масиву. Значення `Array.length` на одиницю більше за номер останнього елемента масиву, тому що нумерація в масиві починається з нуля, а властивість `length` демонструє загальну кількість елементів. Оскільки масиви можуть мати невизначені елементи (із значенням `undefined`), то точніше формулювання звучить так: властивість `length` завжди на одиницю більша, ніж найбільший індекс (номер) елемента масиву. Властивість `length` автоматично оновлюється, залишаючись коректною у разі додавання нових елементів у масив.

Методи об'єкта `Array`

Метод `push` застосовують для додавання значень у масив. Він додає елементи, починаючи з поточної довжини `length` і повертає нову, збільшену довжину масиву.

`shift` видаляє елемент з індексом 0 і зрушує інші на один елемент униз, повертаючи видалений елемент.

`str=arrayObj.join([glue])` повертає з'єднані в рядок усі елементи масиву. `glue` — аргумент, за допомогою якого будуть з'єднані елементи масиву. Якщо аргумент не заданий, елементи будуть з'єднані комами.

`concat` додає `var newArray = array.concat (value1, value2, ..., valueN)` значення або змінні в масив.

`pop` видаляє і повертає останній елемент масиву.

`unshift` додає `arrayObj.unshift ([elem1 [, elem2 [, ... [, elemN]]]])` нові елементи на початок масиву.

`slice arrayObj.slice (start [, end])` повертає частину масиву.

`reverse` змінює порядок елементів на зворотний.

`sort arrayObj.sort ([sortFunction])` дозволяє відсортувати масив.

`splice arrayObj.splice (start, deleteCount, [elem1 [, elem2 [, ... [, elemN]]]])` видаляє частину масиву і додає нові елементи замість видалених.

`find` дозволяє вибрати вказані елементи.

```
[ 1, 3, 4, 2 ].find(x => x > 3) // 4
```

`filter` видаляє вказані елементи

```
[ 1, 3, 4, 2 ].filter(function (x) { return x > 3;
})[0]; // 4
```

`reduce()` застосовує функцію від акумулятора і переглядає кожне значення масиву (зліва направо).

```
[0, 1, 2, 3, 4].reduce(function(previousValue,
currentValue, currentIndex, array) {
  return previousValue + currentValue;
});
[0, 1, 2, 3, 4].reduce( (prev, curr) => prev + curr
);
var total = [ 0, 1, 2, 3 ].reduce( ( acc, cur ) =>
acc + cur, 0 );
```

Нагадаємо, що масиви JavaScript можуть містити як елементи інші масиви. Цю особливість можна використовувати для створення багатовимірних масивів. Так, для створення двовимірного масиву слід застосувати квадратні дужки двічі:

```
var matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

Типізовані масиви

Раніше JavaScript не мала механізмів для доступу до необроблених двійкових даних. Однак, оскільки веб-застосунки стають все потужнішими, додаючи такі функції, як маніпуляції з аудіо- і відеоданими, доступ до необроблених даних з застосуванням `WebSocket` тощо, то в останніх версіях стандарту було введено типізовані масиви.

Типізовані масиви дуже схожі на звичайні: вони мають довжину, усі стандартні методи масиву, елементи можуть бути доступні через операторні дужки `[]`. Вони відрізняються від звичайних масивів за такими параметрами:

- усі їх елементи мають однаковий тип, присвоєння елементів перетворює значення до цього типу;
- вони суміжні. Нормальні масиви можуть мати пропуски, типізовані ж масиви займають суцільний фрагмент пам'яті і не можуть мати пропусків;
- ініціалізуються нулями. Це обумовлено особливостями, описаними у попередньому пункті.

Утім типізовані масиви не слід плутати зі звичайними масивами, так виклик `Array.isArray()` на типізованому масиві повертає `false`. Крім того, не всі методи, доступні для звичайних масивів, підтримуються на типізованих масивах (наприклад, `push` і `pop`).

Для досягнення максимальної гнучкості та ефективності для роботи з типізованими масивами в JavaScript було введено спеціальний буфер. Буфер (реалізований об'єктом `ArrayBuffer`) — це об'єкт, що представляє фрагмент даних. Він не має типу і не пропонує жодного механізму для одержання доступу до його змісту. Для одержання доступу до пам'яті, що міститься в буфері, необхідно застосовувати спеціальні методи. Так, із `ArrayBuffer` можна створити типізований масив або об'єкт `DataView`, який являє собою буфер у певному форматі, і застосовувати його для читання і запису вмісту буфера.

Типізовані масиви можна застосовувати як спеціальні об'єкти, наведені нижче.

Тип	Розмір у байтах	Опис	С тип
`Int8Array`	1	8-бітне знакове ціле	`int8_t`
`Uint8Array`	1	8-бітне беззнакове ціле	`uint8_t`
`Uint8ClampedArray`	1	8-бітне беззнакове ціле	`uint8_t`
`Int16Array`	2	16-бітне знакове ціле	`int16_t`
`Uint16Array`	2	16-бітне беззнакове ціле	`uint16_t`
`Int32Array`	4	32-бітне знакове ціле	`int32_t`
`Uint32Array`	4	32-бітне беззнакове ціле	`uint32_t`
`Float32Array`	4	32-бітне типу float	`float`
`Float64Array`	8	64-бітне типу double	`double`

Доступ до елементів типізованих масивів можна одержати за допомогою ітераторів

```
let ui8 = Uint8Array.of(0,1,2);
   for (let byte of ui8) {
       console.log(byte);
   }
```

але зазвичай застосовують звичайний спосіб

```
let ui8 = new Uint8Array([0,1,2]);
   for(let i=0;i<ui8.length;i++) …
```

Перетворення на звичайні масиви

Після обробки масиву іноді потрібно перетворити його назад на звичайний масив. Це можна зробити за допомогою `Array.from` або застосувавши такий код:

```
var typedArray = new Uint8Array([1, 2, 3, 4]),
normalArray = Array.prototype.slice.call(typedArray);
```

API-інтерфейси із застосуванням типізованих масивів

`FileReader.readAsArrayBuffer()`

Метод читає зміст зазначеного файлу.

`XMLHttpRequest`

Метод `send()` підтримує передачу типізованих масивів і об'єктів `ArrayBuffer`.

`ImageData.data`

Являє собою `Uint8ClampedArray` одновимірний масив, що містить дані в порядку **RGBA**, із цілими значеннями від 0 до 255 включно.

`WebGL`

Під час програмування тривимірної графіки широко застосовують типізовані масиви.

Підпрограми в JavaScript

У процесі створення програми доцільно виділити в ній логічно незалежні частини (підпрограми). Кожну частину за необхідності можна розбити на окремі підпрограми і т.д. Розбиття програми на підпрограми полегшує процес відлагодження, оскільки дозволяє відлагоджувати кожну підпрограму окремо. Один раз створену і відлагоджену підпрограму можна застосовувати довільну кількість разів. Основним елементом для реалізації підпрограм в мові JavaScript є функція.

Функція — ключова концепція в JavaScript. Найважливішою особливістю мови є першокласна підтримка функцій (functions as first-class citizen). Будь-яка функція — це об'єкт, а отже, нею можна маніпулювати, зокрема:

- передавати як аргумент і повертати як результат у разі виклику інших функцій (функцій вищого порядку);
- створювати анонімно і привласнювати як значення змінних або властивостей об'єктів.

Це визначає високу виразну ефективність JavaScript і дозволяє відносити її до мов, що реалізують функціональну парадигму програмування.

Для того щоб послуговуватися функцією, ми повинні спочатку її визначити. Декларація функції має вигляд

```
function ім'я ([аргументи]){ оператори }
```

Тут ім'я — ідентифікатор, що задає ім'я функції, аргументи — необов'язковий список ідентифікаторів, розділених комами, який містить імена формальних аргументів функції, а оператори — будь-який набір операторів, який називають тілом функції і виконують у разі її виклику.

У мові програмування JavaScript функції поділяють на іменовані та анонімні. Основна їх відмінність у тому, що доступ до іменованої функції можна отримати скрізь, а до анонімної — тільки після її оголошення.

Змінні, визначені всередині функції, невидимі поза нею. Будь-яка функція має доступ до всіх змінних та інших функцій, визначених у тій же області видимості, що і сама функція. Іншими словами, функція, визначена в глобальній області видимості, має доступ до всіх змінних, визначених у глобальній області видимості. Функція, визначена усередині іншої функції, має доступ до всіх змінних, визначених у батьківській функції і будь-яких інших змінних, до яких має доступ батьківська функція.

Функції в JavaScript можна викликати з будь-якою кількістю аргументів. Якщо в функцію передано менше аргументів, ніж є у визначенні, то відсутні аргументи одержують значення `undefined`. Як параметр будь-якої функції можна передавати іншу функцію.

```
var ім'я = new Function(параметри, 'Тіло функції')
var sum = function(x,y){ return x+y;}
map( function (a) { return a*3 } , [1,2,3]) // =
[3,6,9]
```

Сучасний стандарт мови підтримує анонімні стрілкові функції (fat arrow function):

```
(param1, param2, …, paramN) => { оператори }
(param1, param2, …, paramN) => expression=
```

JavaScript надає кілька спеціальних функцій для керування рядковими й числовими значеннями:

- `eval()` виконує рядок, що являє собою фрагмент програми на мові JavaScript;
- `parseInt()` перетворює рядок на ціле число в заданій системі числення (якщо можливо) ;
- `parseFloat()` перетворює рядок на число із плаваючою точкою (якщо можливо) ;
- `escape()` обчислює новий рядок, у якому деякі символи будуть замінені шістнадцятковими послідовностями;
- `unescape()` обчислює новий рядок, в якому шістнадцяткові послідовності замінюються на символи, які вони зображають;

- `encodeURIComponent()` кодує компонент уніфікованого ідентифікатора ресурсу (URI) шляхом заміни кожного примірника певних символів на один, два, три або чотири байти, що відповідають кодуванню UTF-8;
- `decodeURIComponent()` виконує зворотне кодування;
- `isNaN([value])` повертає `true`, якщо вказаний аргумент не є число, в іншому випадку повертає `false`.

Виклик за допомогою .apply() і .call()

Виділяють і інші способи виклику функцій. Оскільки функції самі є об'єкти, вони містять власні методи. Зокрема, метод `apply()` можна застосовувати, наприклад, коли потрібно адресуватися до функції динамічно або передати різну кількість аргументів, або явно вказати контекст.

```
var fn = resolveAction();
fn.apply(context, [number1, number2]);
```

Типи аргументів функції можуть бути як примітивами (рядки, числа, логічні), так і об'єктами (включаючи `Array` та функції):

- значення-примітиви передаються у функції за значенням: значення копіюється, так що якщо функція змінить значення параметра, ця зміна не матиме зовнішнього ефекту;
- об'єкти передаються в функцію за посиланнями. Якщо функція змінює властивості об'єкта, то ці зміни будуть видимі поза функцією (побічний ефект).

Доступ до аргументів може бути одержаний безпосередньо. Ключове слово `arguments` адресує масив, де міститься список вхідних аргументів, у порядку передачі під час виклику. Загальна кількість аргументів міститься в `arguments.length`, а значення окремого аргументу можна одержати як `arguments[i]`, де `i` — порядковий номер аргументу, починаючи із нуля.

Сучасний стандарт мови дозволяє в елегантному функціональному стилі відмовитися від `arguments`, замінивши його оператором розгортки `(spread)(...)`:

```
const myConcat = (sep, ...arr) => arr.join(sep);
str = myConcat(":",1,2,3,4);   // 1:2:3:4
```

Сучасний стандарт мови дозволяє також явно вказувати значення за замовчуванням за допомогою виразу у випадку оголошення функції:

```
function multiply(a, b = 2){ return a*b;}
multiply(5); // 10
```

Символьні рядки

В JavaScript будь-які текстові дані є рядки. Внутрішнім форматом рядків, незалежно від кодування сторінки, є юнікод (Unicode). Рядки створюють за допомогою подвійних і одинарних лапок.

В JavaScript немає різниці між подвійними і одинарними лапками. Рядки можуть містити спеціальні символи:

Код	**Вивід**
`\0`	нульовий символ (символ `NULL`)
`\'`	одинарна лапка
`\"`	подвійна лапка
`\\`	зворотний слеш
`\n`	новий рядок
`\r`	поворот каретки
`\v`	вертикальна табуляція
`\t`	табуляція
`\b`	забій
`\f`	подача сторінки
`\uXXXX`	символ юнікоду
`\xXX`	символ із кодування Latin-1

Виділяють два способи вибрати конкретний символ у рядку. Перший передбачає застосування методу `charAt`. Іншим способом (уведеним в ECMAScript 5) є розгляд рядка як об'єкта, подібного до масиву, у якому символи мають відповідні числові індекси:

```
'кіт'.charAt(1);  // Поверне "і"
'кіт'[1];         // Поверне "і"
```

Глобальний об'єкт `String` є конструктор рядків або послідовностей символів.

```
str = new String(thing)
str = String(thing)
```

Слід наголосити, що JavaScript розрізняє об'єкти `String` і значення рядкового примітиву (те саме стосується і об'єктів `Boolean` і `Number`). Рядкові літерали (позначають лапками) і рядки, повернуті викликом `String` без використання ключового слова new, є рядкові примітиви. JavaScript автоматично перетворює примітиви на об'єкти `String`, так що на рядкових примітивах можна застосовувати методи об'єкта `String`. У контекстах, коли на примітиві рядка викликається метод або відбувається пошук властивості, JavaScript автоматично перетворює рядковий примітив на об'єкт і викликає відповідний метод або шукає властивість.

```
var s_prim = 'foo';
var s_obj = new String(s_prim);
console.log(typeof s_prim); // виведе 'string'
console.log(typeof s_obj);  // виведе 'object'
```

Рядкові примітиви і об'єкти `String` також дають різні результати у разі застосування глобальної функції `eval()`. Примітиви, що передаються в `eval()`, трактуються як вихідний код; об'єкти ж `String` трактуються так само, як і всі інші об'єкти, а саме: повертається сам об'єкт, наприклад:

```
var s1 = '2 + 2';          // створює рядковий примітив
```

```
var s2 = new String('2 + 2'); // створює об'єкт String
console.log(eval(s1));        // виведе число 4
console.log(eval(s2));        // виведе рядок '2 + 2'
```

Об'єкт `String` також завжди може бути перетворений на його примітивний аналог за допомогою методу `valueOf()`. Він має властивість `length`, яка повертає довжину рядка.

Властивість `String.prototype` дозволяє додавати нові властивості і методи до об'єкта `String`. Усі об'єкти `String` успадковуються від `String.prototype`. Зміни в прототипі об'єкта `String` поширюються на всі його екземпляри.

Метод `String.fromCharCode()` повертає рядок, створений із вказаної послідовності значень юнікоду.

Об'єкт `String` має також багато методів. У даному посібнику будуть наведені методи, які не належать до мови розмітки гіпертексту.

Методи, доступні на екземплярах `String`

`charAt()`	Повертає символ за вказаним індексом.
`charCodeAt()`	Повертає число, що являє собою значення символу в юнікоді, за вказаним індексом.
`codePointAt()`	Повертає невід'ємне ціле число, яке являє собою закодовану в UTF-16 кодову точку значення за вказаною позицією.
`concat()`	Об'єднує текст двох рядків і повертає новий рядок.
`includes()`	Визначає, чи перебуває рядок усередині іншого рядка.
`endsWith()`	Визначає, чи закінчується рядок символами іншого рядка.
`indexOf()`	Повертає індекс першого входження вказаного значення в об'єкті String або `-1`, якщо входжень немає.
`lastIndexOf()`	Повертає індекс останнього входження вказаного значення в об'єкті String або `-1`, якщо входжень немає.
`localeCompare()`	Визначає, чи є два рядки еквівалентними у поточному мовному стандарті.
`match()`	Визначає, чи відповідає рядок регулярному виразу.
`normalize()`	Повертає форму нормалізації юнікоду для даного рядка.
`quote()`	Повертає рядок у подвійних лапках (" ").
`repeat()`	Повертає рядок, що складається з елементів об'єкта, повторений вказану кількість разів.

`replace()`	Застосовують для пошуку в рядку регулярного виразу і для заміни знайденого підрядка на новий підрядок.
`search()`	Відслідковує збіг регулярного виразу з рядком.
`slice()`	Витягує частину рядка і повертає новий рядок.
`split()`	Розбиває об'єкт `String` на масив рядків, розділених зазначеним рядком або регулярним виразом.
`startsWith()`	Визначає, чи починається рядок символами іншого рядка.
`substr()`	Повертає вказану кількість символів у рядку, що починаються з вказаної позиції.
`substring()`	Повертає символи в рядку між двома індексами.
`toLocaleLowerCase()`	Спрямовує символи в рядку до нижнього регістру згідно з поточною локаллю.
`toLocaleUpperCase()`	Спрямовує символи в рядку до верхнього регістру згідно з поточною локаллю.
`toLowerCase()`	Повертає рядкове значення з символами в нижньому регістрі.
`toSource()`	Повертає літерал об'єкта, що представляє зазначений об'єкт; можна застосовувати це значення для створення нового об'єкта. Перевизначає метод Object.prototype.toSource ().
`toString()`	Повертає рядкове зображення вказаного об'єкта. Перевизначає метод Object.prototype.toString ().
`toUpperCase()`	Повертає рядкове значення із символами у верхньому регістрі.
`trim()`	«Обрізає» пробільні символи на початку і в кінці рядка. Частина стандарту ECMAScript 5.
`trimLeft()`	«Обрізає» пробільні символи з лівого боку рядка.
`trimRight()`	«Обрізає» пробільні символи з правого боку рядка.
`valueOf()`	Повертає примітивне значення зазначеного об'єкта. Перевизначає метод Object.prototype.valueOf ().
`[@@ iterator]()`	Повертає новий об'єкт ітератора `Iterator`.

Приклад застосування ітераторів за символами наведено нижче:

```
var str = 'ABCD\uD835\uDC68';
var strIter = str[Symbol.iterator]();
while(c=strIter.next().value) console.log(c);
for(c of str) console.log(c);  // Результат такий самий.
```

Регулярні вирази в JavaScript

Регулярні вирази — це формальна мова пошуку і здійснення маніпуляцій із підрядками тексту, заснована на застосуванні метасимволів (символів-джокерів, англ. Wildcard characters). По суті це рядок-зразок, що складається із символів і метасимволів і задає правило пошуку.

Регулярні вирази – шаблони, що застосовують для пошуку комбінацій символів у рядках. В JavaScript регулярні вирази також є об'єкти. Ці шаблони застосовують із методами `exec` і `test` об'єктів `RegExp` та з методами `match`, `replace`, `search` і `split` об'єктів `String`.

Регулярні вирази можна створити двома способами:

- застосовуючи літерал регулярного виразу, який складається з шаблону між символами слешів:

```
var re = /pattern/flags;
```

- викликаючи функцію конструктора об'єкта RegExp object:

```
var re = new RegExp(pattern [, flags]);
```

Шаблон регулярного виразу складається зі звичайних символів, наприклад `/abc/`, або комбінації звичайних і спеціальних символів – `/ab*c/` чи `/Chapter (\d+)\.\d*/`.

Дужки навколо будь-якої частини регулярного виразу означають, що дана частина рядка буде збережена. Після цього вона може бути повторно застосована. Наприклад

```
str.replace(/("|')([^'"])*\1","")
```

Методи, що застосовують регулярні вирази

Регулярні вирази застосовують методи `exec` і `test` об'єкта `RegExp` і методи `match`, `replace`, `search` і `split` об'єкта `String`. Якщо потрібно просто перевірити, чи містить даний рядок підрядок, який відповідає зразку, то застосовують методи `test` або `search`. Якщо ж необхідно «витягти» підрядок, що відповідає зразку, то доведеться застосувати методи `exec` або `match`. Метод `replace` забезпечує пошук заданого підрядка і заміну його на інший рядок, а метод `split` дозволяє розбити рядок на кілька підрядків, ґрунтуючись на регулярному виразі або звичайному текстовому рядку.

Метасимволи, застосовні у регулярних виразах, наведено нижче.

Символ	Значення
\	Робить звичайні символи спеціальними. Наприклад, вираз / s / шукає просто символ 's'. А якщо поставити \ перед s, то / \ s /уже позначає пробільний символ. І навпаки, якщо символ спеціальний, наприклад *, то \ зробить його просто звичайним символом "зірочка". Так, /a*/ шукає 0 або більше символів 'a', що йдуть поспіль. Щоб знайти a із зірочкою 'a*', поставимо \ перед спец. символом: /a*/
^	Позначає початок вхідних даних. Якщо встановлено прапорець

Символ	Значення
	багаторядкового пошуку ("m"), то спрацює на початку нового рядка. Наприклад, /^A/ не знайде ‘A’ в "an A", але знайде ‘A’ в "An A"
$	Позначає кінець вхідних даних. Якщо встановлено прапорець багаторядкового пошуку, то також спрацює в кінці кожного рядка. Наприклад, /t$/ не знайде ‘t’ в "eater", але знайде в "eat"
*	Позначає повторення 0 або більше разів. Наприклад, /to*/ знайде ‘boooo’ в "A ghost booooed" і ‘b’ в "A bird warbled", але нічого не знайде в "A goat grunted"
+	Позначає повторення 1 або більше разів. Еквівалентно {1,}. Наприклад, /a+/ знайде ‘a’ в "candy" і все ‘a’ в "caaaaaaandy"
?	Означає, що елемент може бути як наявним, так і відсутнім. Наприклад, / e? Le? / Знайде ‘el’ в "angel" і ‘le’ в "angle." Якщо застосовують відразу після одного з квантифікаторів * + ? або {}, то задає «нежадібний» пошук (повторення мінімально можливої кількості разів, до найближчого наступного елемента шаблону), на противагу «жадібному» режиму за замовчуванням, за якого кількість повторень максимальна, навіть якщо наступний елемент шаблону теж підходить. Крім того, ? застосовують у попередньому перегляді, описаному в таблиці під (?=) (?!) (?:
.	(Десяткова точка) позначає будь-який символ, крім переходу на інший рядок: \n \r \u2028 \u2029. (Можна застосовувати [\s\S] для пошуку будь-якого символу). Наприклад, /.n/ знайде ‘an’ і ‘on’ в "day, an apple is on the tree", але не ‘nay’
(x)	Знаходить x і запам’ятовує. Називають «запам’ятовувальними дужками». Наприклад, /(foo)/ знайде і запам’ятає ‘foo’ в "foo bar." Знайдений підрядок зберігається в масиві-результаті пошуку або в наперед визначених властивості об’єкта RegExp: $1, ..., $9. Крім того, дужки об’єднують те, що в них розміщене, в єдиний елемент шаблону
(?:x)	Знаходить x, але не запам’ятовує знайдене. Називають «не запам’ятовувальними дужками». Знайдений підрядок не зберігається в масиві результатів і властивості RegExp. Як і всі дужки, об’єднують те, що в них розташоване, в єдиний підшаблон
x(?=y)	Знаходить x, тільки якщо за ним йде y. Наприклад, /Jack(?=Sprat)/ знайде ‘Jack’, тільки якщо за ним іде ‘Sprat’. /Jack(?=Sprat|Frost)/ знайде ‘Jack’, тільки якщо за ним

Символ	Значення
	слідує ‘Sprat’ або ‘Frost’. Однак ні ‘Sprat’ nor ‘Frost’ не увійдуть у результат пошуку
x(?!y)	Знаходить x, тільки якщо за ним не йде y. Наприклад, /\d+(?!\.)/ знайде число, тільки якщо за ним не іде десяткова точка
x|y	Знаходить x або y. Наприклад, /green|red/ знайде ‘green’ у "green apple" або ‘red’ у "red apple"
{n}	n — позитивне ціле число. Знаходить рівно n повторень попереднього елемента. Наприклад, /a{2}/ не знайде ‘a’ в "candy," але знайде обидва a в "caandy" і перші дві a в "caaandy"
{n,}	n — позитивне ціле число. Знаходить n і більш за повторення елемента. Наприклад, /a{2,}/ не знайде ‘a’ в "candy", але знайде всі ‘a’ в "caandy" і в "caaaaaaandy"
{n,m}	n і m — позитивні цілі числа. Знаходить від n до m повторень елемента
[xyz]	Набір символів. Знаходить будь-який із перелічених символів. Можна вказати проміжок, застосовуючи тире. Наприклад, [abcd] — те саме, що [a-d]. Знайде ‘b’ в "brisket" і ‘c’ в "ache"
[^xyz]	Будь-який символ, крім зазначених у наборі. Можливо вказати проміжок. Наприклад, [^ abc] — те саме, що [^ a-c]. Знайде ‘r’ у "brisket" і ‘h’ у "chop"
[\b]	Знаходить символ backspace. (Не плутати з \ b)
\B	Позначає не межу слів. Наприклад, /\w\Bn/ знайде ‘on’ в "noonday", а /y\B\w/ знайде ‘ye’ в "possibly yesterday"
\cX	X – буква від A до Z. Позначає контрольний символ у рядку Наприклад, /\cM/ позначає символ Ctrl-M
\d	Знаходить цифру із будь-якого алфавіту. Слід застосовувати [0-9], щоб знайти тільки звичайні цифри. Наприклад, /\d/ або /[0-9]/ знайде ‘2 ‘в "B2"
\D	Знайде нецифрові символи (усі алфавіти). [^0-9] — еквівалент для /\D/"
\f\r\n	Відповідні спецсимволи form-feed, line-feed, перехід рядка
\s	Знайде будь-який пробільний символ, включаючи пробіл, табуляцію,

Символ	Значення
	переходи рядка і інші пробільні символи. Наприклад, `/\s\w*/` знайде ‘bar’ у "foo bar"
`\S`	Знайде будь-який символ, крім пробільних. Наприклад, `/\S\w*/` знайде ‘foo’ в "(foo bar)"
`\t`	Символ табуляції
`\v`	Символ вертикальної табуляції
`\w`	Знайде будь-який словесний (латинський алфавіт) символ, включаючи літери, цифри і знак підкреслення. Еквівалентний `[A-Za-z0-9_]`. Наприклад, / \ w / знайде ‘a’ в "apple," ‘5 ‘в "$ 5.28," і ‘3’ в "3D"
`\W`	Знайде будь-який символ слова. Еквівалентний `[^A-Za-z0-9_]`
`\n`	n — ціле число. Обернене посилання на n-й збережений підрядок у дужках. Наприклад, `/apple(,)\sorange \1/` знайде ‘apple, orange,’ в "apple, orange, cherry, peach"
`\0`	Знайде байт із кодом 0. Не слід додавати в кінці інші цифри
`\xhh`	Знайде символ із кодом hh (2 шістнадцяткові цифри)
`\uhhhh`	Знайде символ із кодом hhhh (4 шістнадцяткові цифри)

Прапорці регулярних виразів:

`g` — глобальний пошук;
`i` — нечутливий до регістру;
`m` — багаторядковий;
`y` — виконати «липкий» пошук, починаючи з поточної позиції у рядку.

Метод	Опис
`exec`	Метод об’єкта `RegExp`, який шукає збіг у рядку; повертає масив.
`test`	Метод об’єкта `RegExp`, який перевіряє, чи є збіг у рядку щодо шаблону і повертає `false`, якщо порівняння зі зразком виявилося невдалим, в іншому випадку `true`.
`match`	Метод об’єкта `String`, який виконує пошук збігу у рядку. Повертає масив інформації або `null`.
`search`	Метод об’єкта `String`, який перевіряє на збіг у рядку. Повертає індекс збігу або -1, якщо пошук не вдається.
`replace`	Метод об’єкта `String`, призначений для пошуку зразка і заміни

знайденого підрядка на новий.

`split` Метод об’єкта `String`, який застосовує регулярний вираз або рядок, щоб розбити рядок на масив підрядків.

Формат JSON

JSON (англ. JavaScript Object Notation; укр. об’єктний запис JavaScript, вимовляють «джейсон») — це текстовий формат обміну даними між комп’ютерами. JSON являє собою текст і може бути прочитаний людиною. Він дозволяє описувати об’єкти та інші структури даних. Головним чином застосовують для передачі структурованої інформації в Інтернеті.

JSON здебільшого застосовують під час розробки веб-програм, а саме у ході застосування технології AJAX. Його часто застосовують як заміну XML під час асинхронної передачі структурованої інформації між клієнтом і сервером. При цьому перевагою JSON перед XML є те, що він дозволяє складні структури в атрибутах, займає менше місця і безпосередньо перетворюється за допомогою JavaScript на об’єкти.

Якщо говорити про веб-застосування, то він ефективний у задачах обміну даними як між браузером і сервером (AJAX), так і між самими серверами (програмні HTTP-інтерфейси). Формат JSON також застосовують для зберігання складних динамічних структур в базах даних (MongoDB) або файловому кеші (localStorage).

JSON передбачає наявність двох структур:

- набір пар ім’я/значення. У мові JavaScript це реалізовано як об’єкт;
- упорядкований список значень. У мові JavaScript це реалізовано як масив.

Значення може бути зображено рядком у подвійних лапках або числом, логічними `true` чи `false`, або `null`, або об’єктом, або масивом. Крім того, специфікація JSON включає коментарі в стилі `/* */`. Наступний приклад демонструє об’єкт, що описує людину, у форматі JSON. У об’єкті є рядкові поля імені і прізвища, об’єкт, що описує адресу, і масив, який містить список телефонів

```
{
   "firstName": "John",
   "lastName": "Doe",
   "address": {
       "streetAddress": "Park ave, app. 25",
       "city": "Orlando",
       "postalCode": 12345
   },
   "phoneNumbers": [
       "050-4412312",
       "065-5012345"
   ]
```

```
}
```

Хоча JSON призначений для передачі даних у серіалізованому вигляді і його застосовують у багатьох мовах програмування, його синтаксис відповідає синтаксису JavaScript і це спричиняє низку проблем, що стосуються безпеки. Часто для обробки даних, одержаних від зовнішнього джерела у форматі JSON, застосовують функцію `eval()` без попередньої перевірки. Техніка застосування `eval()` робить систему вразливою, якщо джерело JSON-даних не належить до надійних.

Із метою підвищити безпеку під час застосування даного формату в JavaScript було введено спеціальний об'єкт `JSON`. Він має два методи: `parse` і `stringify`.

Метод `parse` перетворює рядок JSON на об'єкт.

```
JSON.parse(text [, reviver]),
```

де

`text` — рядок JSON;

`reviver` — функція, що викликається для кожного члена об'єкта.

Даний метод повертає об'єкт або масив.

Функція `JSON.stringify()` перетворює значення JavaScript на рядок `JSON`.

```
JSON.stringify( value [, replacer] [, space],
```

де

`value` — значення, зазвичай об'єкт або масив;

`replace` — функція або масив, застосовна для перетворення результатів;

`space` — додає відступи, пробіли і символи розриву рядка в текст JSON.

Даний метод повертає рядок, що містить текст JSON.

Значення, які не мають зображення в JSON, такі як `undefined`, не будуть перетворені. В об'єктах вони будуть видалені, у масивах — замінені значенням `null`. Рядкові значення розпочинаються і закінчуються лапками. Всі символи юнікоду можуть бути узяті в лапки, за винятком символів, яким повинні передувати escape-символи (зі зворотною скісною лінією).

Неможливо створити об'єкт `JSON` за допомогою оператора `new`. За спроби це зробити з'являється помилка. Проте можна перевизначити або змінити даний об'єкт.

Непродумане застосування `JSON` робить сайти вразливими до підробки міжсайтових запитів. Оскільки тег `<script>` допускає застосування джерела, що належить до іншого домену, це дозволяє виконувати код, переданий у форматі JSON, в контексті іншої сторінки, що уможливлює компрометацію паролів або іншої конфіденційної інформації користувачів, які пройшли авторизацію на іншому сайті.

Об’єктно-орієнтоване програмування у JavaScript

JavaScript має низку властивостей об’єктно-орієнтованої мови, але за рахунок концепції прототипів підтримка об’єктів у ній відрізняється від традиційних мов ООП.

Об’єкт — це складний тип даних, він об’єднує безліч значень в єдиний модуль і дозволяє зберігати і одержувати останні за їх іменами. Іншими словами, об’єкти — це невпорядковані колекції властивостей, кожна з яких має своє ім’я і значення. Властивостями можуть бути звичайні змінні, що містять значення, чи об’єкти.

У JavaScript виділяють два способи створення об’єктів: за допомогою оператора `new`, за яким йде функція-конструктор або за допомогою літерала об’єкта:

```
var empty = new Object();
var empty = {};  // об’єкт створений за допомогою літерала
```

Зазвичай для доступу до значень властивостей об’єкта застосовують оператор точку (`.`). Значення в лівій частині оператора має бути посиланням на об’єкт, до властивостей якого потрібно одержати доступ. Значення в правій частині оператора має бути ім’ям властивості. Властивості об’єкта функціонують як змінні: у них можна зберігати значення і зчитувати їх:

```
var user = {
      name: "Ганнібал Лектор",
      age: 34,
      married: false
};

user.age = 35;
user.male = "чоловік"; // додаємо нову властивість
```

У даному прикладі важливо звернути увагу на два моменти — нову властивість об’єкта, можна додати в будь-якому місці, просто присвоївши цій властивості значення. Також значення уже створених властивостей можна змінювати в будь-який момент, наприклад простим присвоєнням нового значення.

Зауважимо, що значення властивостей, розташованих усередині літерала об’єкта, указують після двокрапки. Якщо додається нова властивість або присвоюється нове значення вже існуючого об’єкта, то замість двокрапки застосовують операцію присвоювання. Як уже було наголошено, значенням властивості може бути об’єкт

```
var obj = {
  name: "Потап",
  colors: {
    first: "yellow",
    second: "blue"
  }
```

```
};
```

Значенням властивості colors є об'єкт `{first: "yellow",second: "blue"}`.

Для видалення властивостей об'єкта застосовують оператор `delete`:

```
 delete obj.colors;
```

Цикл за властивостями `for in` дозволяє послідовно перебрати всі властивості об'єкта. Його можна застосувати, наприклад, під час налагодження сценаріїв, у процесі роботи з об'єктами, які можуть мати довільні властивості із заздалегідь невідомими іменами, для виведення імен властивостей на екран або для роботи із їх значеннями. Синтаксис даного циклу має такий вигляд:

```
for (var ім'я_змінної in об'єкт) { ... }
```

Перед виконанням циклу ім'я однієї з властивостей присвоюється змінній у вигляді рядка. У тілі циклу цю змінну можна застосовувати як для одержання імені властивості, так і її значення за допомогою оператора `[]`.

```
var user = {
  name: "Ганнібал Лектор",
  age: 34,
  married: false
};
for (var name in user) s+=name+"="+user[name]+ "\n";
```

Цикл `for in` не перераховує властивості в якомусь заданому порядку, і хоча він перераховує всі властивості, визначені користувачем, деякі наперед визначені властивості й методи JavaScript він не перераховує.

Для перевірки наявності тієї чи іншої властивості застосовують оператор `in`. Ліворуч від оператора поміщають ім'я властивості у вигляді рядка, праворуч — об'єкт, який перевіряють на наявність зазначеної властивості.

```
var obj = {};
if ("a" in obj) {
  obj.a = 1;
}else {
   // Такої властивості не існує
}
```

У разі звернення до відсутньої властивості повертається значення `undefined`. Тому для перевірки так само досить часто застосовують інший спосіб — порівняння значення з `undefined`.

```
if (obj.x !== undefined) obj.x = 1;
```

Як нам відомо, доступ до властивостей об'єкта здійснюють за допомогою оператора **.** (точка). Доступ до властивостей об'єкта можливий також за допомогою оператора `[]`, який зазвичай застосовують під час роботи з масивами. Таким чином, нижченаведені два вирази мають однакове значення і на перший погляд нічим не відрізняються:

```
obj.property = 10;
obj['property'] = 10;
```

Важлива відмінність між цими двома синтаксисами, на яку слід звернути увагу, полягає в тому, що в першому варіанті ім'я властивості являє собою ідентифікатор, а в другому — рядок.

Примітка: ще одна різниця між доступом до властивості об'єкта через `.` (точку) і `[]` полягає у тому, що на ім'я властивості в разі доступу за допомогою оператора `.` накладені синтаксичні обмеження — це ті ж правила іменування, що і для звичайної змінної, у той час як у випадку звернення до властивості об'єкта за допомогою оператора `[]` ім'я властивості задають у вигляді рядка і воно може містити будь-які символи.

```
obj["Моє ім'я"] = "Ганнібал";
```

Який же спосіб краще застосовувати в ході написання коду? Зазвичай звернення до властивості через точку застосовують, якщо користувач на етапі написання програми вже знає назви властивостей. А якщо властивості визначатимуть під час виконання, наприклад, їх уводитиме відвідувач і записуватиме в змінну, то єдиний вибір — квадратні дужки.

У JavaScript прості типи (числа, рядки і т.д.) передають за значенням, об'єкти, навпаки, присвоюються і передаються за посиланням — це означає, що у змінній як значення зберігається не сам об'єкт, а посилання на місце в пам'яті, де він розташований.

У разі копіювання змінної копіюється лише посилання, а об'єкт залишається в єдиному екземплярі. Отже, дві змінні можуть посилатися на один і той же об'єкт.

Методи об'єкта в JavaScript

Метод — це функція, яка зберігається як значення властивості об'єкта і може викликатися в його контексті.

Метод повинен мати доступ до даних об'єкта для повноцінної роботи. Для доступу до властивостей об'єкта із методу застосовують ключове слово `this`. Воно посилається на об'єкт, у контексті якого викликається метод, і дозволяє звертатися до інших його методів і властивостей.

Функція-конструктор і оператор `new`

Крім літерального синтаксису створення об'єкта його можна створювати за допомогою функції-конструктора і оператора `new`.

Конструктор — це функція, яка здійснює ініціалізацію властивостей об'єкта і призначена для застосування спільно із оператором new.

```
var myCar = new Car();
```

Оператор `new` створює новий порожній об'єкт без будь-яких властивостей, а потім викликає функцію-конструктор (або просто конструктор), передаючи їй щойно створений об'єкт. Головне завдання конструктора полягає в ініціалізації новоствореного об'єкта — установці всіх його властивостей, які необхідно ініціалізувати до того, як об'єкт зможе застосувати програма. Після

того як об’єкт створений і ініціалізований, змінній myCar присвоюють посилання на нього. Створювані об’єкти таким чином зазвичай називають екземпляром об’єкта (або класу), у нашому випадку myCar є екземпляр об’єкта Car.

У разі виклику конструктора без аргументів дужки можна не ставити.

Важливою особливістю виклику конструктора є застосування властивості prototype конструктора як прототипу нового об’єкта. Це означає, що всі об’єкти, створені за допомогою одного конструктора, успадковують один і той же об’єкт-прототип і, відповідно, є члени одного і того ж класу.

```
function User(name,age){
 this.name=name||"John";
 this.age=age||0;
 this.toString=()=>"Name="+this.name+",
age="+this.age;
}
var u=new User("Tom",20);
console.log(u.toString()); // Name=Tom, age=20
```

Для створення об’єктів в ES6 було введено класи. Для того щоб оголосити клас, слід застосувати ключове слово class із ім’ям класу. Синтаксис класів не вводить нову об’єктно-орієнтовану модель успадкування в JavaScript. Просто JavaScript-класи забезпечують набагато простіший і зрозуміліший синтаксис для створення об’єктів.

Метод constructor — це спеціальний метод для створення і ініціалізації об’єкта, створеного класом. У класі може бути тільки один спеціальний метод із ім’ям constructor.

Ключове слово static визначає статичний метод класу. Статичні методи викликають без створення екземпляра класу і їх неможливо викликати під час створення екземпляра класу. Їх часто застовують для створення допоміжних функцій:

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
        this.toStr = ()=>"("+this.x+","+this.y+")" ;
    }
    static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;
        return Math.hypot(dx,dy);
    }
}
const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
console.log(p1.toStr());          // виклик методу
```

```
console.log(Point.distance(p1, p2));//
```
виклик статичного методу

Ключове слово `extends` застосовують в оголошенні класу, щоб створити клас, як дитя іншого класу.

Ключове слово `super` застосовують для виклику функції батьківського об’єкта:

```
‘use strict’;
class Polygon {
  constructor(height, width) {
    this.name = ‘Polygon’;
    this.height = height;
    this.width = width;
    this.toStr= ()=> this.name+" height="+this.height+"
width="+this.width;
  }
}
class Square extends Polygon {
  constructor(length) {
    super(length, length);
    this.name = ‘Square’;
  }
}
sq = new Square(2,2);
console.log(sq.toStr())
```

Функції дуже часто застосовують як методи об’єктів, що реалізують об’єктно-орієнтоване програмування (ООП).

Важлива відмінність між оголошенням функцій і оголошенням класів полягає в тому, що оголошення функцій можна поставити в будь-якому місці програми, а оголошення класів — перед його застосуванням. Спочатку потрібно оголосити свій клас, а потім одержати доступ до нього, в іншому випадку код, подібний до нижченаведеного, видаватиме помилку:

```
var p = new Polygon(1,2); // ReferenceError
class Polygon { ... }
```

Класи утворюють ієрархію, в якій кожен клас може бути нащадком раніше визначеного класу. Нащадок класу має всі його властивості, але може мати додаткові властивості або змінювати властивості свого предка. При цьому набір властивостей даного класу зафіксований у його оголошенні і не може бути змінений у ході виконання програми. Можна сказати, що тут поточний стан реалізується екземплярами класів, а наслідування — структурою і поведінкою.

JavaScript підтримує наслідування, засноване на прототипах. Кожен конструктор має відповідний прототип об’єкта і кожен об’єкт, створений ним, містить приховане посилання на даний прототип. Прототип, у свою чергу, може містити посилання на свій прототип і т. д. Так утворюється ланцюжок прототипів. Посилання на властивість об’єкта — це посилання на перший прототип у ланцюжку прототипів об’єкта, який містить властивість із такою назвою. Іншими словами, якщо об’єкт має властивість з такою назвою, то

застосовують посилання на цю властивість; якщо ні, то досліджують прототип даного об’єкта і т. д.

У JavaScript поточний стан і методи реалізуються об’єктами, а структура і поведінка — успадковуються. На відміну від мов, заснованих на класах, властивості можна динамічно додавати до об’єктів і динамічно видаляти. Зокрема, конструктори не зобов’язані присвоювати значення всіх чи деяких властивостям створюваного об’єкта.

```
function PoliceOfficer() {
  this.retire=true;
  this.getJob=()=>"Police Officer";
}
var Capitan = function (age) {
  this.rank = "Capitan";
  this.age = age;
};
Capitan.prototype = new PoliceOfficer();
Capitan.prototype.getRank = function () {
  return this.rank;
};
var John = new Capitan(37);

print(John.getJob()); // 'Police Officer'
print(John.getRank()); // 'Capitan'
print(John.retire); // true
```

Даний приклад свідчить, що властивість `prototype` дозволяє реалізувати наслідування.

Об’єкт Promise

Промісифікація (англ. Promise — обіцянка) — це процесс створення обгортки для асинхронного функціонала. Після промісифікації застосування функціонала найчастіше стає набагато зручнішим.

`Promise` — це спеціальний об’єкт, який містить свій стан. Спочатку `pending` («очікування»), потім або `fulfilled` («виконано успішно»), або `rejected` («виконано з помилкою»). На `Promise` можна навішувати колбеки двох типів:

- `onFulfilled` — спрацьовують, коли `Promise` в стані «виконаний успішно»;
- `onRejected` — спрацьовують, коли `Promise` в стані «виконаний із помилкою».

За своєю суттю обіцянки трохи схожі на події за винятком:

- обіцянка може завершитися тільки один раз: або успішно, або помилково;
- якщо обіцянка виконалася і користувач тільки після цього встановить їй колбек, він відпрацює, незважаючи на те, що подія вже давно відбулася.

Це надзвичайно зручно для асинхронних запитів, оскільки користувачу нецікаво знати причину, а набагато цікавіше обробити результати події.

Спосіб застосування `Promise` в загальних рисах такий:

1. Код, якому необхідно зробити щось асинхронно, створює об'єкт `promise` і повертає його.
2. Зовнішній код, отримавши `promise`, навішує на нього обробники.
3. По завершенні процесу асинхронний код переводить `promise` в стан `fulfilled` (із результатом) або `rejected` (із помилкою). При цьому автоматично викликаються відповідні обробники в зовнішньому коді.

Синтаксис створення `Promise`:

```
'use strict';
let promise = new Promise((resolve, reject) => {
  // через 1 секунду готов результат: result
  setTimeout(() => resolve("result"), 1000);
  // через 2 секунди — reject з помилкою, він буде
проігнорований
  setTimeout(() => reject(new Error("ignored")), 2000);
});
promise.then(
    result => alert("Fulfilled: " + result), // спрацює
    error => alert("Rejected: " + error) // не спрацює
  );
```

Зробимо таку обгортку для запитів за допомогою `XMLHttpRequest`. Функція `httpGet(url)` буде повертати об'єкт, який у разі успішного завантаження даних із url перейде у стан fulfilled з цими даними, а у випадку помилки — у rejected з інформацією про помилку:

```
function httpGet(url) {
  return new Promise(function(resolve, reject) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);
    xhr.onload = function() {
      if (this.status == 200) {
        resolve(this.response);
      } else {
        var error = new Error(this.statusText);
        error.code = this.status;
        reject(error);
      }
    };
```

```
    xhr.onerror = function() {
      reject(new Error("Network Error"));
    };
    xhr.send();
  });
}
```

Приклад застосування:

```
httpGet("/article/promise/user.json").then(
    response => alert(`Fulfilled: ${response}`),
    error => alert(`Rejected: ${error}`)
  );
```

Зауважимо, що деякі сучасні браузери вже підтримують `fetch` - новий вбудований метод для AJAX-запитів, покликаний замінити `XMLHttpRequest`.

Об’єкт Proxy

Проксі (proxy) — особливий об’єкт, завдання якого — перехоплювати звернення до іншого об’єкта і, за необхідності, модифікувати їх. Програмісти застосовують проксі для оголошення розширеної семантики JavaScript-об’єктів. Синтаксис:

```
let proxy = new Proxy(target, handler)
```

де:

- `target` — об’єкт, звернення до якого слід перехоплювати;
- `handler` — об’єкт із «пастками»: функціями-перехоплювачами для операцій до `target`.

Майже будь-яка операція може бути перехоплена й оброблена проксі до або навіть замість доступу до об’єкта target, наприклад: читання і запис властивостей, одержання списку властивостей, виклик функції (якщо `target` - функція) тощо. Різних типів «пасток» досить багато. Якщо для операції немає пастки, то вона виконується безпосередньо над `target`.

Спочатку ми докладно розглянемо найважливіші «пастки», а потім подивимося і на їх повний список.

Найчастішими є «пастки» для читання і запису в об’єкт:

```
get(target, property [,receiver]) /* Спрацьовує під час читання властивості */
set(target, property, value [,receiver])  /* Спрацьовує у ході запису властивості */
```

Аргументи:

- `target` — цільовий об’єкт, той же який був переданий першим аргументом у `new Proxy`;

- `property` — ім'я властивості;
- `receiver` — об'єкт, до якого було застосовано присвоювання. Зазвичай сам проксі або прототип, що успадковує від нього. Цей аргумент застосовують нечасто.

Метод `set` повинен повернути `true`, якщо присвоювання успішно оброблено і `false` у разі помилки (обумовлює генерацію `TypeError`). Нижче наведено приклад застосування операцій читання і запису:

```
'use strict';
let user = {};
let proxy = new Proxy(user, {
  get(target, prop) {
    if(prop=='name') return false;
    return target[prop];
  },
  set(target, prop, value) {
    if(prop=='firstName' && value=='Doe') return false;
    target[prop] = value;
    return true;
  }
});
proxy.firstName = "John";     // запис
console.log(user.firstName); // читання: John
proxy.firstName = "Doe";      // помилка
console.log(user.name);       // помилка
```

Під час кожної операції читання і запису властивостей у коді, наведеному вище, спрацьовують методи `get/set`. Із їх допомогою значення в кінцевому рахунку потрапляє в об'єкт (або зчитується з нього).

«Пастка» `has` спрацьовує в операторі `in` і деяких інших випадках, коли перевіряють наявність властивості. Синтаксис `has` аналогічний `get`.

`deleteProperty` за синтаксисом аналогічна `get/has`. Спрацьовує під час операції `delete`, повинна повернути `true`, якщо видалення було успішним.

«Пастка» `enumerate` перехоплює операції `for..in` і `for..of` за об'єктом. Якщо «пастки» немає, то ці оператори працюють із вихідним об'єктом. Якщо ж `enumerate` є, то вона буде викликана з єдиним аргументом `target` і зможе повернути ітератор для властивостей. У нижчеподаному прикладі ітерація відбувається за всіма властивостями, крім тих, що починаються з підкреслення `_`:

```
'use strict';
let user = {
  name: "John",
  surname: "Doe",
  _version: 1,
```

```
  _secret: 768
};
let proxy = new Proxy(user, {
  enumerate(target) {
    let props = Object.keys(target).filter(function(prop){
      return prop[0] != '_';
    });
    return props[Symbol.iterator]();
  }
});
// відфільтровані властивості, що починаються з _
for(let prop in proxy) {
  alert(prop); // виводить властивості name, surname
}
```

Проксі працює не тільки зі звичайними об'єктами, але і з функціями. Якщо аргумент `target` проксі — функція, то стає доступна «пастка» `apply` для її виклику. Метод

```
apply(target, thisArgument, argumentsList)
```

отримує аргументи:

- `target` — вихідний об'єкт;
- `thisArgument` — контекст `this` виклику;
- `argumentsList` — аргументи виклику у вигляді масиву.

Вона може обробити виклик сама і/або передати його функції.

```
'use strict';
function sum(a, b) {  return a + b; }
let proxy = new Proxy(sum, {
  // передасть виклик у target, попередньо повідомивши про нього
  apply: function(target, thisArg, argumentsList) {
    console.log(`Буду обчислювати суму:
${argumentsList}`);
    return target.apply(thisArg, argumentsList);
  }
});
// Виводить спочатку повідомлення з проксі, а потім уже суму
console.log( proxy(1, 2) );
```

Щось подібне можна зробити за допомогою замикання. Але проксі може набагато більше. У тому числі і перехоплювати виклики через `new`.

«Пастка» `construct (target, argumentsList)` перехоплює виклики за допомогою `new`. Вона отримує вихідний об'єкт `target` і список аргументів `argumentsList`.

Список можливих функцій-перехоплювачів, які може задавати `handler`:

- `getPrototypeOf` перехоплює звернення до методу `getPrototypeOf`;
- `setPrototypeOf` перехоплює звернення до методу `setPrototypeOf`;
- `isExtensible` перехоплює звернення до методу `isExtensible`;
- `preventExtensions` перехоплює звернення до `preventExtensions`;
- `getOwnPropertyDescriptor` перехоплює звернення до методу `getOwnPropertyDescriptor`;
- `defineProperty` перехоплює звернення до `defineProperty`;
- `has` перехоплює перевірку існування властивості в операторі `in` і деяких інших методах вбудованих об'єктів;
- `get` перехоплює читання властивості;
- `set` перехоплює запис властивості;
- `deleteProperty` перехоплює видалення властивості оператором `delete`;
- `enumerate` спрацьовує в разі виклику `for..in` або `for..of`, повертає ітератор для властивостей об'єкта;
- `ownKeys` перехоплює звернення до методу `getOwnPropertyNames`;
- `apply` перехоплює виклики `target()`;
- `construct` перехоплює виклики `new target()`.

Кожен перехоплювач запускають із `handler` як `this`. Це означає, що `handler` крім пасток може містити й інші властивості та методи. Кожен перехоплювач одержує в аргументах `target` і додаткові параметри залежно від типу. Якщо перехоплювач у `handler` не вказаний, то операцію за замовчуванням здійснюють безпосередньо над `target`.

Proxy дозволяє модифікувати поведінку об'єкта як завгодно, перехоплювати будь-які звернення до його властивостей і методів, включаючи виклики для функцій. Особливо важлива можливість перехоплювати звернення до відсутніх властивостей.

Пошук і обробка помилок

JavaScript можна змусити бути чіткішою, перевівши її в суворий режим (strict mode). Для цього зверху файлу або тіла функції пишуть `"use strict"`. Приклад:

```
"use strict";
//  цей код буде виконуватися в стандарті ES5
```

```
counter = 100; // тут буде помилка
```

Так, якщо не написати `var` перед змінною, як у прикладі перед `counter`, JavaScript за замовчуванням створить глобальну змінну і застосує її. У суворому режимі видається помилка. Це дуже зручно. Проте помилка не видається, коли глобальна змінна вже існує, а видається тільки тоді, коли створюється нова змінна з тим же іменем.

Ще одна змінна — прив'язка `this` містить `undefined` у тих функціях, які викликали не як методи. Коли ми викликаємо функцію не в суворому режимі, `this` посилається на об'єкт у глобальній області видимості. Тому якщо користувач випадково неправильно викличе метод у суворому режимі, JavaScript видасть помилку, якщо спробує прочитати щось із `this`, а не буде просто працювати з глобальним об'єктом.

Суворий режим має ще декілька функцій. Він забороняє викликати одну і ту саму функцію з різною кількістю параметрів і видаляє деякі потенційно проблемні властивості мови (наприклад, інструкцію `with`). Напис `"use strict"` перед текстом програми рідко спричиняє проблеми, проте допомагає користувачу бачити їх.

Обробка винятків (також опрацьовування (обробля́ння) виняткóвих ситуáцій, англ. exception handling) — механізм мов програмування, призначений для обробки помилок часу виконання і інших можливих проблем (винятків), які можуть виникнути у ході виконання програми.

Коли виняток доходить до низу стека і його ніхто не зловив — його обробляє оточення. Як саме — залежить від конкретного оточення. У браузерах опис помилки видається в консоль (вона зазвичай доступна в меню «Інструменти» або «Розробка»).

Генерація власних помилок

Оператор `throw` генерує помилку. Ключове слово `throw` застосовують для викидання винятку.

```
throw <об'єкт помилки>
```

Як об'єкт помилки можна передати що завгодно, це може бути навіть не об'єкт, а число або рядок, але все ж краще, щоб це був об'єкт, бажано — сумісний зі стандартним, тобто щоб у нього були як мінімум властивості `name` і `message`.

В JavaScript вбудовано низку конструкторів для стандартних помилок: `SyntaxError`, `ReferenceError`, `RangeError` та деякі інші. Наприклад:

```
if (!user.name) {
    throw new SyntaxError("Дані некоректні");
  }
```

Для обробки винятків у мові JavaScript застосовують конструкцію `try..catch`. Даний синтаксис має такий вигляд:

```
try {
   .. пробуємо виконати код ..
} catch(e) {
   .. перехоплюємо виняток ..
} finally {
   .. виконуємо завжди ..
}
```

Суттєвим недоліком мови JavaScript є те, що вона не надає безпосередньої підтримки вибіркового відлову винятків: або ловимо всі, або ні.

Секція `finally` не обов'язкова, але якщо вона є, то виконується завжди. Тобто блок `finally` виконується за будь-якого виходу із `try..catch`, у тому числі і `return`.

Викид винятку приводить до розмотуванню стека до момента зустрічі із блоком `try..catch` або доходження до дна стека. Значення винятку буде передано в блок `catch`, який зможе переконатися в тому, що це дійсно шуканий виняток і обробити його. Для роботи з непередбачуваними подіями в потоці програми можна застосувати блоки `finally`, щоб певні частини коду були виконані у будь-якому випадку.

Тип даних `Symbol`

Новий тип даних `Symbol` слугує для створення унікальних ідентифікаторів. Спочатку розглянемо оголошення і особливості символів, а потім – їх застосування. Синтаксис має вигляд

```
let sym = Symbol();
```

Звернемо увагу, що не `new Symbol`, а просто `Symbol`, тому що це — примітив. У символів є і відповідний `typeof`:

```
'use strict';
let sym = Symbol();
alert( typeof sym ); // symbol
```

Кожен символ — унікальний. У функції `Symbol` є необов'язковий аргумент — «ім'я символу».

```
'use strict';
let sym = Symbol("name");
alert( sym.toString() ); // Symbol(name)
```

Але при цьому, якщо у двох символів однакове ім'я, то це не означає, що вони однакові:

```
alert( Symbol("name") == Symbol("name") ); // false
```

Якщо потрібно в різних частинах програми застосувати однаковий символ, то можна передати між ними об'єкт символ або застосувати «глобальні символи» і «реєстр глобальних символів», які розглянемо пізніше.

Глобальні символи

Існує «глобальний реєстр» символів, який дозволяє, за необхідності, мати загальні «глобальні» символи, які можна отримати з реєстру за іменем.

Для читання (або створення за відсутності) «глобального» символу слугує виклик `Symbol.for(ім'я)`.

```
'use strict';
// створити символ у реєстрі
let name = Symbol.for("name");
// читання з реєстра
alert( Symbol.for("name") == name ); // true
```

Таким чином, можна з різних частин програми, звернувшись до реєстру, отримати єдиний глобальний символ з ім'ям "name".

У виклику `Symbol.for`, який повертає символ із заданим ім'ям, є зворотний виклик — `Symbol.keyFor(sym)`. Він дозволяє отримати за глобальним символом його ім'я:

```
'use strict';
// створити символ у реєстрі
let name = Symbol.for("name");
// отримати ім'я символа
alert( Symbol.keyFor(name) ); // name
```

Зауважимо, що Symbol.keyFor працює тільки для глобальних символів, для інших буде повернуто `undefined`:

```
'use strict';
alert( Symbol.keyFor(Symbol.for("name")) ); // namealert(
Symbol.keyFor(Symbol("name2")) ); // undefined
```

Застосування символів

Символи можна застосовувати як імена для властивостей об'єкта, які не будуть перераховані ітератором.

```
let user = {
  name: "Дмитро",
  age: 30,
  [Symbol.for("isAdmin")]: true
};
// в циклі for..in також не буде символу
alert( Object.keys(user) ); // name, age
// доступ до властивості через глобальний символ
console.log( user[Symbol.for("isAdmin")] ); //true
console.log( user.isAdmin );      // undefined
```

Найширше застосування символів передбачено всередині самого стандарту JavaScript. У сучасному стандарті є багато системних символів. Їх список є в

специфікації ES6. У ній для стислості символи прийнято позначати як @@ім'я, наприклад @@iterator, але доступні вони як властивості Symbol.

У випадку введення нових стандартів Ecmascript не можна передбачити і зарезервувати певні властивості наявних об'єктів для нового функціонала. Тому ввели цілий тип «символи». Їх можна застосовувати для задання таких властивостей, тому що вони:

- унікальні;
- не беруть участі в циклах;
- не зламають старий код.

Продемонструємо відсутність конфлікту для нової системної властивості Symbol.iterator:

```
'use strict';
let obj = {
  iterator: 1,
  [Symbol.iterator]() {}
}
alert(obj.iterator); // 1
alert(obj[Symbol.iterator]) // function, конфлікту немає
alert( Object.getOwnPropertySymbols(obj)[0].toString());
alert( Object.getOwnPropertyNames(obj) ); // iterator
```

Щоб одержати всі символи об'єкта, застосовують метод Object.getOwnPropertySymbols. Він повертає всі символи в об'єкті (і тільки їх). Зауважимо, що стара функція getOwnPropertyNames символи не повертає, що гарантує відсутність конфліктів зі старим кодом.

Інтернаціоналізація

Об'єкт Intl — це простір імен для ECMAScript Internationalization API, який забезпечує порівняння рядків, форматування чисел, форматування дати і часу відповідно до встановленої мовної локалі. Конструктори для об'єктів Collator, NumberFormat і DateTimeFormat є властивості об'єкта Intl. Опишемо їх, а також функціональні можливості, загальні для інтернаціоналізації функцій, чутливих до вибраної мовної локалі.

Intl.Collator Конструктор для коллатора — об'єкта, який дозволяє порівнювати рядки відповідно до прийнятої (не англійської) мови.

```
// in German,  "ä" sorts with "a"
// in Swedish, "ä" sorts after "z"
var list = [ "ä", "a", "z" ]
var l10nDE = new Intl.Collator("de")
var l10nSV = new Intl.Collator("sv")
```

```
l10nDE.compare("ä", "z") === -1
l10nSV.compare("ä", "z") === +1
console.log(list.sort(l10nDE.compare)) // ["a","ä","z"]
console.log(list.sort(l10nSV.compare)) // ["a","z","ä"]
```

`Intl.DateTimeFormat` Конструктор для об'єктів, які дозволяють форматувати дату і час відповідно до вибраної мовної локалі.

```
var l10nEN = new Intl.DateTimeFormat("en-US")
var l10nDE = new Intl.DateTimeFormat("de-DE")
l10nEN.format(new Date("2015-01-02")) === "1/2/2015"
l10nDE.format(new Date("2015-01-02")) === "2.1.2015"
```

`Intl.NumberFormat` Конструктор для об'єктів, які дозволяють форматувати числа відповідно до вибраної мовної локалі.

```
var l10nEN = new Intl.NumberFormat("en-US")
var l10nDE = new Intl.NumberFormat("de-DE")
l10nEN.format(1234567.89) === "1,234,567.89"
l10nDE.format(1234567.89) === "1.234.567,89"
```

Слід зазначити, що даний об'єкт ще досить «сирий» і не всі його методи реалізовані в сучасних браузерах.

Короткий тлумачний словничок до тексту

Browser — веб-оглядач.
API (Application Programming Interface) — набір функцій, що забезпечують стандартний доступ до будь-яких функціональних можливостей.
ECMA (European Computer Manufacturers Association) — некомерційна асоціація європейських виробників комп'ютерів, що займається розробкою стандартів для інформаційних і комунікаційних систем.
ECMAScript — стандартизована ECMA варіант мови програмування JavaScript.
GMT (Greenwich Mean Time) — див UTC.
HTML (Hypertext Markup Language) — тегова мова, яку застосовують для створення платформно-незалежних гіпертекстових документів.
Java — об'єктно-орієнтована мова програмування, незалежна від платформи. У веб застосовують для написання аплетів, що виконуються на стороні клієнта.
JavaScript — об'єктно-орієнтована інтерпретована мова програмування, незалежна від платформи. У веб застосовують для написання сценаріїв.
Mozilla — міжнародна група розробників стандартизованого платформно-незалежного веб-оглядача Firefox, яка створила ядро Gecko.
Netscape Navigator — веб-оглядач, створений корпорацією «Netscape».
Unicode — міжнародний стандарт кодування символів, у якому кожен символ кодується 16-розрядним словом. Див. також UTF-8.
UTC (Universal Coordinated Time) — стандартний світовий час. Раніше називали часом за грінвіцьким меридіаном (GMT, Greenwich Mean Time).
UTF-8 (Unicode Transformation Format) — перетворювач кодів Unicode, призначений для зберігання і транспортування текстової інформації в Інтернеті.
XML (Extensible Markup Language) — тегова мова, яку застосовують для створення платформно-незалежних форматів зберігання даних. XML є гранично спрощена підмножина мови SGML, розроблена як веб-стандарт, але коло її сучасних застосувань набагато ширше.
Властивість (property) — будь-який компонент об'єкта.
Інтернет (Internet) — глобальна комп'ютерна мережа, на якій базується веб.
Компонент (component) — програмний об'єкт, для якого визначено інтерфейс його взаємодії з операційним середовищем.
Локалізація (localization) — внесення змін у програму або документ, пов'язане з перекладанням їх на іншу мову. Локалізація може потребувати зміни таблиці кодування символів, форматів дати і часу, знака грошової одиниці тощо.
Спадкування (inheritance) — в об'єктно-орієнтованих мовах програмування — можливість створення похідних класів і інтерфейсів, які успадковують усі властивості і методи базового класу або інтерфейсу.
Сервер (server) — мережний комп'ютер, який надає клієнтам різні послуги. Див. також веб-сервер.
Сценарій (script) — невелика програма, написана на скриптовій мові, наприклад JavaScript.

Список рекомендованої літератури

Закас, Н. JavaScript. Оптимизация производительности [Текст] /Н.Закас. — Б.м.: Питер, 2012. — 256 с.

Закас, Н. JavaScript для профессиональных веб-разработчиков [Текст] /Н.Закас. — Б.м.: Питер, 2015. — 960 с.

Крокфорд, Д. JavaScript. Сильные стороны [Текст] /Д.Крокфорд. — Б.м.: Питер, 2013. — 176 с.

Маккоу, Ф. Веб-приложения на JavaScript [Текст] /А.Маккоу. — Б.м.: Питер, 2012. — 288 с.

Макфарланд, Д. JavaScript и jQuery. Исчерпывающее руководство [Текст] /Д.С.Макфарланд. — Б.м.: Питер, 2016. — 880 с.

Резиг, Д. Секреты JavaScript ниндзя [Текст] /Д.Резиг, Б.Бибо. — Б.м.: Питер, 2016. — 416 с.

Флэнаган, Д. JavaScript [Текст]: карманный справочник /Д.Флэнаган. — М.: Издат. дом «Вильямс», 2014. — 320 с.

Документація по JavaScript [Електронний ресурс]. — Режим доступу: https://developer.mozilla.org/uk/ — Mozilla Developer Network. — Заголовок з екрана.

Современные возможности ES-2015 [Электронный ресурс]. — Режим доступа: https://learn.javascript.ru/es-modern — Основы JavaScript. — Загл. с экрана.

Зміст