Symbol

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol

https://learn.javascript.ru/symbol

http://xn--80adth0aefm3i.xn--j1amh/Symbol

Symbol

Тип даних `Symbol` може бути використаний для:

- створення унікальних ідентифікаторів;
- опису властивостей об’єкта;
- створення «прихованих» властивостей, оскільки не відображається у циклі `for..in` та `Object.keys;`
- створення глобальних ідентифікаторів;
- налаштування властивостей об’єктів (задання правил отримання примітивних значень, створення ітераторів, ...).

Загальна форма	Приклад
`Symbol(description)` `description` - необов'язковий параметр - рядок з описом Symbol (не впливає на створення)	`let sym1 = Symbol()` `let sym2 = Symbol("foo")  // цей` `let sym3 = Symbol("foo")  // і цей символи є різними` `alert(sym2 == sym3); // false`

Опис властивостей об’єкта

Може бути використаний у випадку, коли хочемо уникнути конфліктів додавання властивостей у різних частинах коду

(можливо функціях зовнішніх бібліотек)

Загальна форма	Приклад
При додаванні до існуючого об’єкта об’єкт [символ] = значення	let user = { name: "Вася" } let id = Symbol("id") **user[id] = 1** alert(user[id])
При описі літерала необхідно вказувати **у квадратних дужках** { . . . . [символ] : значення . . . . }	let id = Symbol("id"); let user = { name: "Вася", **[id]**: 123 };

```
const privateProp = Symbol("private");
const obj = {
    name: "John",
    [privateProp]: "secret data"
};

console.log(Object.keys(obj)); // ["name"]
console.log(obj[privateProp]); // secret data
```

Нехай маємо масив об’єктів без ідентифікаторів (який вам **зараз локально** (тільки у вашому фрагменті) потрібний

Додаємо поле «id» з унікальними значеннями

```
function addSymbolIds(objects) {
    return objects.map(obj => ({
        ...obj,
        id: Symbol() // Без опису
    }));
}

const dataFromBackend = [
    { name: "John", age: 30 },
    { name: "Jane", age: 25 },
    { name: "Bob", age: 40 }
];

const dataWithIds = addSymbolIds(dataFromBackend);
```

Додаємо поле з унікальною назвою і значеннями

```
const ID_SYMBOL = Symbol.for("object.id");

function addSymbolIds(objects) {
    return objects.map(obj => ({
        ...obj,
        [ID_SYMBOL]: Symbol()
    }));
}

const dataFromBackend = [
    { name: "John", age: 30 },
    { name: "Jane", age: 25 },
    { name: "Bob", age: 40 }
];

const dataWithIds = addSymbolIds(dataFromBackend);
```

Особливості властивостей-символів

Не відображається у циклі `for..in`	`let id = Symbol("id");` `let user = {` `  name: "Вася",` `  age: 30,` `  [id]: 123` `};` `for (let key in user) alert(key); //тільки name, age` `//` `alert( "Прямий: " + user[id] );`
Не відображаться у `Object.keys`	
Копіюється з використанням `Object.assign`	`let id = Symbol("id");` `let user = {` `  [id]: 123` `};` `let clone = Object.assign({}, user);` `alert( clone[id] ); // 123`
Отримати список усіх клічів-символів `Object.getOwnPropertySymbols(obj)`	
Отримати список усіх ключів (символьних і несимвольних) `Reflect.ownKeys(obj)`	

Глобальні символи

Можуть бути використані для створення глобальних ідентифікаторів (у глобальному реєстрі)

Створення/зчитування

Загальна форма	Приклад
//створюємо з використанням **`Symbol.for!!`** `Symbol.for(key)` Якщо символа з ключем `key` немає, то він буде створений `Symbol(key)` і буде збережений у глобальному реєстрі з ключем `key`	// читаємо символ з глобального реєстра let id = `Symbol.for("id");` // якщо немає, то буде створено // в іншій частині коду можемо прочитати і зберегти у іншій змінній let idAgain = Symbol.for("id"); //порівнюємо (це один і той же символ) alert(id === idAgain); // true

Отримання ключа глобального символу з глобального реєстру

Загальна форма	Приклад
`Symbol.keyFor(sym)` ***(тільки для глобальних символів!)***	// створюємо символи let sym = Symbol.for("name"); let sym2 = Symbol.for("id"); // отримуємо імена символів alert(Symbol.keyFor(sym)); // name alert(Symbol.keyFor(sym2)); // id

```
//---------- ПРИКЛАД. Створити унікальний ідентифікатор для конфігурації-------
// Створюємо символ у глобальному реєстрі для конфігурації
const CONFIG_SYMBOL = Symbol.for("analytics.config");

// Клас для аналітики
class Analytics {
  constructor() {
    // Ініціалізуємо конфігурацію за замовчуванням
    this[CONFIG_SYMBOL] = {
      loggingEnabled: false,
      apiKey: null
    };
  }
  // Метод для налаштування конфігурації
  configure({ loggingEnabled, apiKey }) {
    this[CONFIG_SYMBOL] = {
      loggingEnabled: !!loggingEnabled,
      apiKey: apiKey || null
    };
  }
  // Метод для відстеження подій
  trackEvent(eventName) {
    const config = this[CONFIG_SYMBOL];
    if (config.loggingEnabled)   console.log(`Tracking event: ${eventName}, API Key: ${config.apiKey}`);
     else console.log(`Event tracking is disabled: ${eventName}`);
  }
  // Статичний метод для отримання конфігурації з іншого модуля
  static getConfig(instance) {   return instance[Symbol.for("analytics.config")];   }
}
// Експортуємо екземпляр бібліотеки
const analytics = new Analytics();
export default analytics;
```

```
import analytics from "./analytics.js";

// Налаштовуємо бібліотеку
analytics.configure({
    loggingEnabled: true,
    apiKey: "xyz123"
});

// Відстежуємо подію
analytics.trackEvent("UserLogin");
```

```
import analytics from "./analytics.js";

// Отримуємо конфігурацію за допомогою того ж символу
const config = Analytics.getConfig(analytics);
console.log("Current analytics config:", config);

// Відстежуємо ще одну подію
analytics.trackEvent("PageView");
```

Властивості об'єкта Symbol:

Властивість	Призначення
Symbol.toPrimitive	символ для перетворення об'єкта у примітивне значення
Symbol.iterator	вказує ітераційну поведінка об'єкта
Symbol.hasInstance	як конструктор об'єкта розпізнає об'єкт в якості його примірника
Symbol.match	визначає відповідність регулярного виразу до рядка
Symbol.description	опис символу
Symbol.replace	символ який вказує на метод який замінює значення
Symbol.search	символ який вказує на метод який повертає індекс в рядку
Symbol.species	яку функція конструктора використовувати для створення похідних об'єктів
Symbol.toStringTag	опис об'єкта
Symbol.unscopables	символ для виключення з with
Symbol.split	символ який вказує на метод який розбиває рядок на індекси

http://xn--80adth0aefm3i.xn--j1amh/Symbol

Перетворення об'єктів у примітиви

При перетворенні об'єктів у примітиви:

- спочатку перевіряється наявність метода Symbol.toPrimitive, і використовуємо лише його. Параметр hint дозволяє визначити, до якого типу треба привести об'єкт ("string", "number", "default");
- якщо метода немає Symbol.toPrimitive і перетворення до String то викликаємо `toString` (або `valueOf`, якщо метода `toString` немає);
- якщо метода Symbol.toPrimitive немає і перетворення до Number, то викликаємо `valueOf` (або `toString`).

Тип параметра ***hint***	Випадки застосування	Приклад для тестового об'єкта user `let user = {` `  name: "John",` `  money: 1000` `}`
string	Коли відбувається явне/неявне перетворення типу до рядка	1) явне перетворення до string String(user) // без методів приведення дорівнює '[object Object]' 2) використання у операторах, де очікується string alert(user) 3) використання у якості ключа об'єкта otherObj[user] = 17 // {[object Object]: 17}, Object.keys(otherObj) -> ['[object Object]']
number	Коли відбувається явне/неявне перетворення до типу Number	1) явне перетворення до Number Number(user) // без методів приведення дорівнює NaN 2) виконання математичних операцій let r = user * 7 // без методів приведення дорівнює NaN 3) виконання операцій порівняння user>9 //false user<9 //false
default	У випадку, коли однозначно не вдається визначити до якого типу примітива треба виконати приведення	1)бінарний плюс може виконуватись як з String так і з Number 2)порівняння може виконуватись з величинами різних типів **(у переважній більшості такі перетворення мають таку ж реалізацію як і для hint = number)**

Опис метода Symbol.toPrimitive

<table>
<tr><th>Загальна форма</th><th>Приклад</th></tr>
<tr><td>----- Додавання метода до існуючого об’єкта ----

об’єкт [Symbol.toPrimitive] = function(hint) {
  ....
};</td><td><pre>let user = {
  name: "John",
  money: 1000
}
//-------------------------------------
user[Symbol.toPrimitive]=function(hint) {
    let result
    switch(hint){
      case 'number': result= this.money
         break
      case 'string': result= this.name
         break
      default:
         result null
       }
     return result
}
let res = +user // 1000
let str = “Hello ” + user // Hello John</pre></td></tr>
<tr><td>------ Додавання до літерала/класу (назва у дужках [ ]) -----
let об’єкт ={
 ....
 [Symbol.toPrimitive] (hint) {
   ....
 },
 ....
}</td><td><pre>let user = {
  name: "John",
  money: 1000,

  [Symbol.toPrimitive](hint) {
    let result
    switch(hint){
      case 'number': result= this.money
         break
      case 'string': result= this.name
         break
      default:
         result = null
    }
     return result
  }
}
let res = +user // 1000
let str = “Hello ” + user // Hello John</pre></td></tr>
</table>

Задача. Дано об’єкт - Student (піб, масив оцінок). При приведення до рядка повертати «піб». При приведенні до числа повертати мінімальну оцінку

Ітератори

Ітеративні об'єкти :

- дозволяють здійснювати перегляд даних об'єкта по одному за раз
- дозволяють здійснювати перегляд даних з використанням циклу `for..of`
- вбудовані колекції даних таких як масиви, рядки, Set, Map та ін.
- є узагальненням масивів і дозволяють здійснювати поступовий перегляд даних, які формуються згідно з заданими правилами
- реалізують метод Symbol.iterator

Для реалізації ітератора у об'єкті повинна бути функція з назвою Symbol.iterator, яка повинна повертати об'єкт з функцією next(), яка у свою чергу повинна повертати при кожному виклику об'єкт

`{done: Boolean, value: any},`

де done=true означає, що ітерація завершена, інакше value – це наступне значення.

Загальна форма	Приклад
---- реалізація ітератора на існуючому об'єкті --- *//1)додаємо метод Symbol. iterator* об'єкт **[Symbol. iterator] = function() {** . . . . . . *//2) функція повинна повертати об'єкт з методом next* **return {** **next: ( ) => {** . . . . . . *//3)метод next повертає об'єкт {done:.., value :...}* **return { done: … , value: …}** . . . . . . **}** **}** . . . . . . **}**	//тестовий об'єкт, до якого будемо додавати ітератор let range = { from: 1, to: 5 }; //1)при використанні for..of спочатку викликається метод Symbol.iterator range[**Symbol.iterator**] = function() { //2) Далі, for..of працює тільки з цим ітератором, запитуючи у нього наступні значення викликаючи метод next return { **next: ( )** => { // 3. він повинен повертати значення як об'єкт {done:.., value :...} if (this.current <= this.last) { return { done: false, value: this.current++ }; } else { return { done: true }; } } } } `for (let num of range) {` `alert(num) // 1, 2, 3, 4, 5` `}`

----- Реалізація ітератора при описі об'єкта/класу (назва у дужках [ ])-----

```
let об'єкт ={
 ........
   //1) функція Symbol.iterator є методом об'єкта і повертає this
 [Symbol.iterator] ( ) {
     ........
    return this
 },
 ........
   2) цей об'єкт також має містити метод next()
 next ( ) {
     ........
     return { done: ... , value: ...}
     ........
 }
 ........
}
```

```
//тестовий об'єкт, до якого будемо додавати ітератор
let range = {
  from: 1,
  to: 5,

  //1)при використанні for..of спочатку викликається метод Symbol.iterator
  [Symbol.iterator]() {
    this.current = this.from;
    return this
  },
  //2) Далі for..of працює викликаючи метод next
 next() {
    if (this.current <= this.to) {
      return { done: false, value: this.current++ }
    } else {
      return { done: true }
    }
  }
}
```

```
for (let num of range) {
  alert(num) // 1, 2, 3, 4, 5
}
```

Задача. Дано об’єкт - Student (піб, масив оцінок). Створити ітератор, який дозволить перебирати оцінки

Приклад ітератора з класом

--- Реалізація ітератора при описі класу (назва у дужках [ ])-----

```
class назва_класу ={
 ........
   //1) функція Symbol.iterator є методом об'єкта і повертає this
 [Symbol.iterator] ( ) {
    ........
    return this
 },
 ........
   2) цей об'єкт також має містити метод next()
 next ( ) {
     ........
     return { done: … , value: …}
     ........
 }
 ........
}
```

```
class Range {
  constructor(from, to) {
    this.from = from
    this.to = to
  }

//1)при використанні for..of спочатку викликається метод Symbol.iterator
  [Symbol.iterator]() {
    this.current = this.from
    return this
  }
//2) Далі for..of працює викликаючи метод next
  next() {
    if (this.current <= this.to) {
      return { done: false, value: this.current++ }
    } else {
      return { done: true }
    }
  }
}
const range = new Range(1, 5)

// тепер це працює!

for (let num of range) {
  alert(num) // 1, 2, 3, 4, 5
}
```

Задача. Дано клас - Student (піб, масив оцінок). Створити ітератор, який дозволить перебирати оцінки

Задача. Дано клас Library, що може містити (масив об’єктів книг (назва, автор, ціна)).
При перегляді у циклі потрібно видавати рядкове значення «назва - ціна»

Задача. Випадкові числа з кроком не більше 10. Створити клас, об’єкт якого можна було б використати як ітеро
об’єкт і він видавав поступово випадкові числа починаючи від 1 і кожне наступне є більшим не більше ніж на 10

Генератори ***https://uk.javascript.info/generators***

- особливий тип функції, що призначений для генерування значень згідно заданих правил
- при описі функції додатково ставиться зірочка «function*»
- кожне наступне значення повертається не з використанням «return», а з використанням «yield»
- об'єкт генератор має функцію next, яка повертає
  {value: значення, done: false}, якщо значення ще не закінчились
  {value: undefined, done: true}, якщо значення закінчились

Загальна форма	Приклад
//--------- опис ------- function* назва_генератора (параметри){ . . . . . . yield значення1; yield значення2; yield значення3; . . . . . . }	function* seasonsNumber() { yield 1 yield 2 yield 3 yield 4 }
//----- використання у циклі for..of ----	for (const item of seasonsNumber()) { console.log(item) // 1 //2 //3 //4 }
//---- явний виклик метода next	const generatorNumberGenerator = seasonsNumber() //------------------------- console.log(generatorNumberGenerator.next()) //{value: 1, done: false} console.log(generatorNumberGenerator.next()) //{value: 2, done: false} console.log(generatorNumberGenerator.next()) //{value: 3, done: false} console.log(generatorNumberGenerator.next()) //{value: 4, done: false} console.log(generatorNumberGenerator.next()) //{value: undefined, done: true} console.log(generatorNumberGenerator.next()) //{value: undefined, done: true}
Використання для формування колекцій	let sequence = [0, ... seasonsNumber ()] // [0,1,2,3,4]

Композиція генераторів

Загальна форма (використати ***yield****)	Приклад
//------ базові генератори -------- function* generator1 () { . . . . . . } function* generator2 () { . . . . . . } . . . . . . //------- комплексний генератор function* complexGenerator() { ***yield**** generator1 (); ***yield**** generator2 (); . . . . }	//------ базовий генератор -------- function* generateSequence(start, end) { for (let i = start; i <= end; i++) yield i; } //------- комплексний генератор --- function* generatePasswordCodes() { ***yield**** generateSequence(48, 57); ***yield**** generateSequence(65, 90); ***yield**** generateSequence(97, 122); } //------------------------ let arr=[... generatePasswordCodes()]

Комбінація генераторів і ітераторів

Загальна форма (*[Symbol.iterator])	Приклад Поступово повертати значення від from до to
let об'єкт = { *[Symbol.iterator]() { . . . . . . yield значення . . . . . . } } }	let range = { from: 1, to: 5, *[Symbol.iterator]() { for(let value = this.from; value <= this.to; value++) { yield value } } }; //---------- alert([...range]) // 1,2,3,4,5

Задача. Створити генератор, у який передається список і генератор поступово видає випадкове значення з цьог
списку поки вони не закінчаться