

Типове данни в JavaScript

Презентация от екип 6



Съдържание

01

Примитивни и
референтни типове
данни

02

Масиви и обекти в
JavaScript



03

Итератори и
генератори

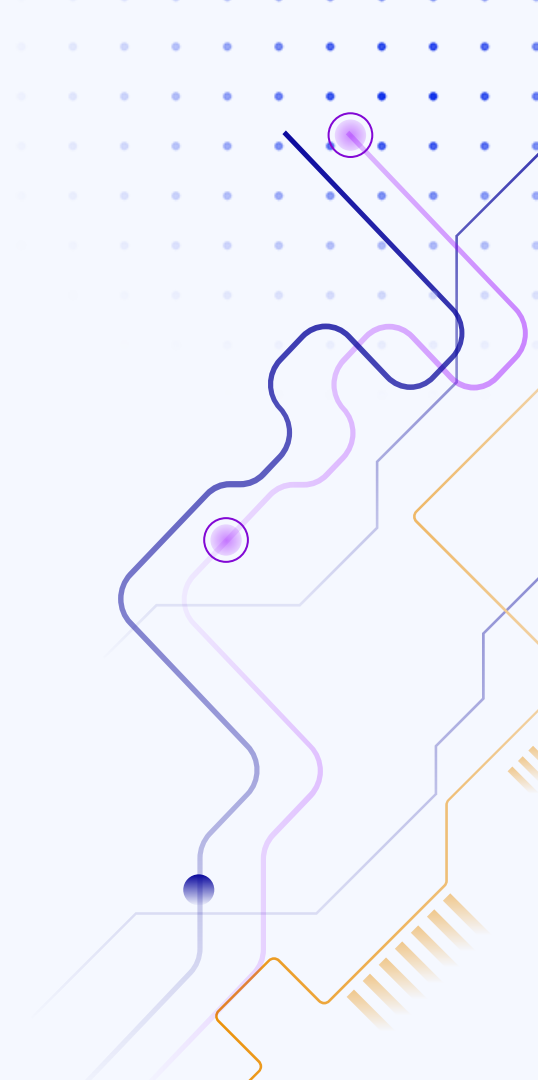
04

Map/Set,
WeakMap/WeakSet и
Garbage Collector



01

ОСНОВНИ ТИПОВЕ В JavaScript



Въведение

Javascript е слабо типизиран език с **динамично** типизиране.

Данните в Javascript се делят основно на два типа:

- **Примитивни типове**
- **Референтни типове**

Примитивни типове

- Не могат да се променят след като са били създадени
- Те са **необекти**
- Съхраняват се в **stack** паметта
- Копират се по **стойност**

! Важно!

```
let variable = "text";  
variable = "test";
```

Това **НЕ** е промяна на стойността. **"test"** е изцяло нова стойност.

Number

Това е основният тип за числа в JavaScript, като може да съдържа както цели числа, така и числа с плаваща запетая.

```
let age = 30;    // Цяло число  
let temperature = 36.6; // Число с  
плаваща запетая
```

! Важно! В JavaScript **няма** различие между цели и дробни числа, те се представят чрез типа **Number**.

String

Низовете са последователности от символи и се използват за представяне на текст.

```
let name = "John";  
let greeting = 'Hello, world!';  
let multiline = `This is a  
multi-line string`;
```

! Важно! Низовете са **непроменяеми** (immutable), т.е. не можем да променим отделни символи в низа, можем само да заменяме целия низ.

Boolean

Този тип има само две възможни стойности: `true` или `false`.

```
let isActive = true;  
let hasPermission = false;
```

Използва се в логически изрази и условия.

Undefined

Този тип се използва, когато променлива е декларирана, но не е инициализирана с конкретна стойност.

```
let user;  
console.log(user); // undefined
```

Null

Представява "липса на стойност". Това е уникален примитив, който показва, че променливата не съдържа валидна стойност.

```
let noValue = null;
```

⚠ Важно! `null` не е същото като `undefined`, но и двете често се използват, за да означават "празна" стойност.

Symbol

Тип за създаване на уникални идентификатори, които могат да се използват, например, като ключове в обекти.

```
let id1= Symbol('id');  
let id2= Symbol('id');  
let id3= Symbol.for(id2);  
let id4= Symbol.for(id2);  
  
console.log(id1 === id2) //false  
  
console.log(id3 === id4)  
//true
```

! Важно! Символите са уникални и не могат да бъдат равни помежду си, дори ако имат еднакви описания.

Bigint

Позволява работа с числа, които са по-големи от `Number.MAX_SAFE_INTEGER` (9,007,199,254,740,991 или $2^{53} - 1$).

```
let bigNumber = 9007199254740;
```

Референтни типове

Включват **обекти**, **масиви** и **функции**, които се съхраняват в **heap** паметта и могат да бъдат **променяни**. Стойността им се предава по **референция**.

Function

Функциите също са обекти в JavaScript и могат да бъдат съхранявани в променливи и предавани като аргументи.

```
function version1() { console.log("Hello!"); }

let version2 = function() {
  console.log("Hi there!");
};

let version3 = () => { console.log("Hello again!"); };

version2; //
version3(); // Hello again!
```

Object

Обектите представляват колекции от данни и функции (методи).

```
let person = {  
  name: "Alice",  
  age: 30,  
  greet: function() { console.log("Hello,  
" + this.name); }  
};
```

! Важно! Обектите могат да съдържат различни типове данни като стойности (стрингове, числа и т.н.) и други обекти.

Array

Масивите са специален тип обекти за съхранение на поредици от данни.

```
let numbers = [1, 2, "3", () =>
{console.log("Hi!");}, 5];
```

! Важно! Масивите в JavaScript са динамични и могат да съдържат различни типове данни.

Важно!

Когато работим с примитивни типове, можем да бъдем сигурни, че при присвояване на нова променлива *няма* да има неочаквани промени в стойностите.

С референтни типове обаче трябва да внимаваме, когато работим с обекти или масиви, тъй като промените в една променлива ще се отразят и на други.

```
let a = 10;  
// Примитивен тип (Number)  
  
let b = a;  
// Копира стойността на 'a' в  
// 'b'  
  
b = 20;  
// Променяме 'b', но това НЕ  
// засяга 'a'  
  
console.log(a);  
// 10 (остава непроменено)  
console.log(b); // 20
```

VS

```
let obj1 = { name: "Alice" };  
// Референтен тип (Object)  
  
let obj2 = obj1;  
// Присвоява се РЕФЕРЕНЦИЯ към  
// същия обект  
  
obj2.name = "Bob";  
// Променяме стойността в obj2  
  
console.log(obj1.name);  
// "Bob" (obj1 също се променя!)  
console.log(obj2.name); // "Bob"
```



02

Массиви и обекти в JavaScript



Дефиниране и използване на масиви (Array)

Масиви в JavaScript са колекции от елементи, които могат да бъдат от различен тип (числа, низове, обекти и т.н.).

Деклариране:

```
let numbers = [1, 2, 3, 4, 5];  
let fruits = ["ябълка", "портокал", "банан"];  
let arr = [1, "текст", true];
```

Други начини за създаване на масив:

```
let arr = []; // празен масив
```

```
let emptyArr = new Array();
```

```
let threeElementsArr = new Array(3); // Масив с 3 празни слота
```

```
let fiveZerosArr = new Array(5).fill(0); // [0, 0, 0, 0, 0]
```

!!! Няма предимство в използването на **new Array()**, освен ако не създаваш масив с множество елементи наведнъж.

Достъп до елементи в масив

```
let numbers = [1, 2, 3, 4, 5];  
console.log(numbers[0]); // 1  
console.log(numbers[2]); // 3  
console.log(numbers[numbers.length - 1]); // 5
```

Добавяне и премахване на елементи

Добавяне в края на масива:

```
numbers.push(6);    // [1, 2, 3, 4, 5, 6]
```

Премахване на последния елемент:

```
numbers.pop();      // [1, 2, 3, 4, 5]
```

Добавяне в началото:

```
numbers.unshift(0); // [0, 1, 2, 3, 4, 5]
```

Премахване на първия елемент:

```
numbers.shift();    // [1, 2, 3, 4, 5]
```

Методът **splice()** е универсален метод за работа с масиви, защото позволява **премахване, добавяне и замяна** на елементи директно върху оригиналния масив.

```
array.splice(startIndex, numOfElementsToRemove, element1, element2, ...);
```

startIndex

-> Индексът, от който започва промяната

numOfElementsToRemove

-> Колко елемента да бъдат премахнати (0 ако няма да премахваме)

element1, element2, ...

-> Елементи, които да бъдат добавени на мястото на премахнатите
(По желание)

```
let numbers = [10, 20, 40, 50];  
numbers.splice(2, 0, 30); // Добавя 30 на позиция 2  
console.log(numbers); // [10, 20, 30, 40, 50]
```

Дефиниране и използване на обекти (Object)

Обектът в JavaScript е основна структура от данни, която съхранява информация във формат **"ключ → стойност"**.

!!! Ключовете са низове (или символи), а стойностите могат да бъдат всякакви: числа, низове, масиви, функции, други обекти и т.н.

Създаване на обект

```
let person = {  
  name: "Иван",  
  age: 28  
};
```

```
let person = new Object();  
person.name = "Иван";  
person.age = 28;
```

Достъп до свойства

```
console.log(person.name); // Иван  
console.log(person["age"]); // 28
```

Добавяне

```
person.city = "София";
```

Изтриване

```
delete person.city;
```

Обекти с методи

```
let person = {  
  name: "Иван",  
  introduce: function() {  
    console.log("Здравей, казвам се " + this.name);  
  }  
};  
  
person.introduce(); // Здравей, казвам се Иван
```

Разлики между масиви и обекти

Свойство	Масиви	Обекти
Структура	Подредена колекция от елементи	Колекция от свойства
Достъп до елементи	Чрез индекси (<code>arr[0]</code>)	Чрез ключове (<code>obj.key</code> или <code>obj["key"]</code>)
Кога да използваме?	Когато имаме списък от елементи , като имена, числа	Когато имаме структурирани данни , като потребители
Методи	Вградени методи като <code>.push()</code> , <code>.pop()</code> , <code>.map()</code> , <code>.filter()</code>	Може да използва <code>Object.keys()</code> , <code>Object.values()</code> и <code>Object.entries()</code>
Поддържа дължина	Да	Не

Често използвани методи за масиви

```
let numbers = [1, 2, 3, 4];
```

map() – Преобразува всеки елемент

```
let doubled = numbers.map(num => num * 2); // [2, 4, 6, 8]
```

filter() – Филтрира елементи


```
let evenNumbers = numbers.filter(num => num % 2 === 0); // [2, 4]
```

reduce() – Обединява стойности в една

```
let sum = numbers.reduce((acc, num) => acc + num, 0); // 10
```

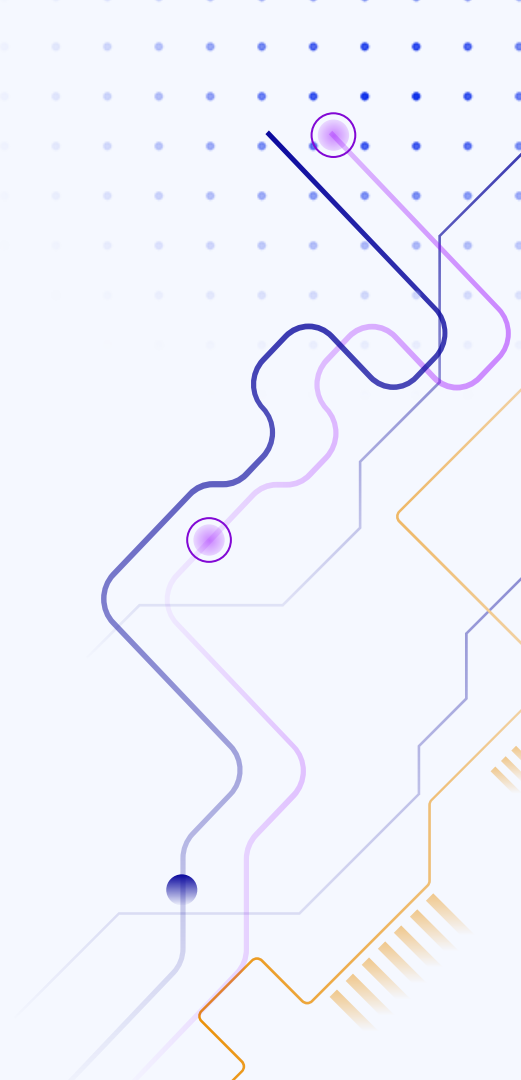
forEach() – Изпълнява функция върху всеки елемент

```
numbers.forEach(num => console.log(num)); // 1, 2, 3, 4
```



03

Итератори и генератори



Какво е **итериране**?

Процесът на обхождане на
елементите на колекция (масив, обект,
множество и др.).

Итератори

Какво е итератор?

Обект, който предоставя начин за последователен достъп до елементите на колекция.

Основни характеристики:

- Имплементира протокола за итериране (Iterable Protocol).
- Има метод `next()`, който връща обект с две свойства: `value` и `done`.

```
const array = [1, 2, 3];
const iterator = array[Symbol.iterator]();

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

Итерируеми обекти

Какво е итерируем обект?

Обект, който имплементира метода `[Symbol.iterator]()`, връщащ итератор.

Примери:

- Масиви
- Низове
- Map и Set

```
const iterable = [10, 20, 30];  
for (const value of iterable) {  
  console.log(value); // 10, 20, 30  
}
```

Генераторите

Какво е генератор?

Специален вид функция, която може да бъде спирана и възобновявана по време на изпълнение.

Синтаксис:

- Дефинира се с `function*`.
- Използва `yield` за връщане на стойности.

```
function* simpleGenerator() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const gen = simpleGenerator();  
console.log(gen.next()); // { value: 1, done: false }  
console.log(gen.next()); // { value: 2, done: false }  
console.log(gen.next()); // { value: 3, done: false }
```

Предимства на генераторите

Лениво изчисление:

Стойностите се генерират само при поискване.

Безкрайни последователности:

Могат да генерират безкрайни потоци от данни.

Пример за безкраен генератор

```
function* infiniteCounter() {  
  let i = 0;  
  while (true) {  
    yield i++;  
  }  
}  
  
const counter = infiniteCounter();  
console.log(counter.next().value); // 0  
console.log(counter.next().value); // 1
```

Практически приложения на итератори и генератори

Итератори:

- Създаване на персонализирани структури от данни.
- Работа с големи или динамични колекции от данни.

Генератори:

- Асинхронно програмиране и управление на потоци от данни.
- Проложения в популярни JavaScript библиотеки (напр. Redux-Saga)

04

Структури от данни Map и Set



Въведение

Това са структури от данни, които осигуряват по-ефективни начини за съхранение и достъп до данни, отколкото обикновените обекти и масиви

Тези структури от данни в Javascript, които ще покажем са:

- Map
- Set

Map

- Колекция от двойки **ключ-стойност**, ключовете са уникални и могат да бъдат от всякакъв тип
- Запазва реда на вмъкване
- Ключовете в **Map** са сравнени по референция

! Важно!

```
let map = new Map();
map.set({ id: 1 }, "Object 1");
console.log(map.get({ id: 1 }));
//undefined различна референция
```

////////////////////////////////////

```
let map = new Map();
let obj = {id: 1};
map.set(obj, "Object 1");
console.log(map.get(obj));
//Object 1 същата референция
```

Set

- Колекция от **уникални стойности**
- Запазва реда на вмъкване
- Проверява уникалността по референция

! Важно!

```
let set = new Set();  
set.add({ id : 1});  
set.add({ id : 1});  
set.add({ id : 1});  
console.log(set.size);  
//3, не 1. Всяко ({id:1}) е нов  
обект в паметта съответно Set ги  
вижда като 3 нови референции.
```

Разлики с обикновените обекти и масиви

Характеристики	Map	Object	Set	Array
Ключове	Всеки тип (object, function, number)	Само string и symbol	Не се използват ключове	Индекс-базиран (0,1,2...)
Ред на елементите	Запазва реда	Няма гаранция	Запазва реда	Запазва реда
Проверка за съществуване	<code>.has(key)</code>	<code>key in object</code>	<code>.has(value)</code>	<code>array.includes(value)</code>
Изпълнение	По-бърз при работа с динамични ключове	По-бавен при големи данни	Оптимизиран за уникални стойности	Подходящ за изброяване
Итерация	<code>forEach()</code> , <code>for..of</code>	<code>for..in</code> , <code>Object.entries()</code>	<code>forEach()</code> , <code>for..of</code>	<code>forEach()</code> , <code>map()</code> , <code>for..of</code>

Map - методи и примери (Part1)

Създаване на Map

```
let userRoles = new Map();
userRoles.set('Alice', 'Admin');
userRoles.set('Bob', 'Editor');
userRoles.set('Charlie', 'Viewer');
console.log(userRoles);
// Map(3) { 'Alice' => 'Admin',
// 'Bob' => 'Editor', 'Charlie' =>
// 'Viewer' }
```

Основни методи

```
let userRoles = new Map();
console.log(userRoles.get('Alice'));
// 'Admin' - Взяма стойността по
// ключ
console.log(userRoles.has('Bob'));
// true - Проверява дали ключ
// съществува
userRoles.delete('Charlie');
// Премахва елемент
console.log(userRoles.size);
// 2 - Брой на елементите
userRoles.clear();
// Изтрива всички стойности
```

Map - методи и примери (Part2)

Итерация

```
let users = new Map([
  ['John', 28],
  ['Jane', 32],
  ['Tom', 40]
]);

for (let [key, value] of users) {
  console.log(`${key} is ${value}
years old.`);
}
// John is 28 years old.
// Jane is 32 years old.
// Tom is 40 years old.
```

Пример за практическо приложение

```
function wordCount(text) {
  let words = text.split(/\s+/);
  let wordMap = new Map();
  words.forEach(word => {
    word = word.toLowerCase();
    wordMap.set(word,
(wordMap.get(word) || 0) + 1);
  });
  return wordMap;
}
console.log(wordCount("Hello world
hello"));
// Map(2) { 'hello' => 2, 'world' => 1}
```

Set- методи и примери (Part1)

Създаване на Set

```
let uniqueNums = new  
Set([1,2,3,3,4,4,5]);  
console.log(uniqueNums);  
// // Set(5) { 1, 2, 3, 4, 5 } -  
Премахнати са дубликатите
```

Основни методи

```
let uniqueNums = new  
Set([1,2,3,3,4,4,5]);  
uniqueNums.add(6); //Добавя елемент  
uniqueNums.delete(2); //Премахва  
елемент  
console.log(uniqueNums.has(3));  
//Проверява дали съществува в случая  
true  
console.log(uniqueNums.size);  
//4 Броя на елементите  
uniqueNums.clear();  
//Изчиства всички елементи
```

Set- методи и примери (Part2)

Итерация

```
let fruits = new
Set(['apple'], ['apple'], ['banana']);

for (let fruit of fruits) {
  console.log(fruit);
}
// apple
// banana
```

Пример за практическо приложение

```
let loggedInUsers = new Set();
function loginUser(username) {
  if(loggedInUsers.has(username)){
    console.log(`${username} вече
е логнат`);
  } else {
    loggedInUsers.add(username);
    console.log(`${username} се
логна успешно`);
  }
}
loginUser('Alice');
loginUser('Bob');
loginUser('Alice');
// Alice се логна успешно
//Bob се логна успешно
//Alice вече е логнат
```



05

WeakMap, WeakSet и Garbage Collection



Какво е WeakMap и WeakSet

WeakMap

Колекция от ключ-стойност, където ключовете са обекти с "слаба референция".

Стойностите могат да бъдат всякакъв тип данни.

WeakSet

Колекция от уникални обекти със слаби референции.

```
// Създаване на нов WeakMap
let weakmap = new WeakMap();
// Създаваме обект, който ще използваме като ключ в
WeakMap
let obj = { name: "John" };
// Добавяме обекта към WeakMap с някаква стойност
weakmap.set(obj, 'Some data');
// Проверяваме дали обектът е в WeakMap
console.log(weakmap.has(obj)); // true, обектът е
добавен
// Сега изхвърляме референцията към обекта
obj = null;
// След като обектът вече не е в употреба,
автоматично ще бъде премахнат от WeakMap

// Проверяваме отново дали елементът е в WeakMap
console.log(weakmap.has(obj)); // false, обектът е
премахнат
```

Разлика между Map/Set и WeakMap/WeakSet

Характеристика	Map/Set	WeakMap/WeakSet
Тип на референцията	Силна референция за ключове/елементи	Слаба референция за ключове/елементи
Изчистване на паметта	Не се изчистват автоматично	Изчистват се, когато няма други референции към тях
Поддръжка на примитивни типове	Могат да бъдат ключове/елементи	Само обекти могат да бъдат елементи или ключове

Защо и кога да използваме WeakMap и WeakSet?

- **Паметна ефективност:** Ако работим с обекти като ключове или елементи, които могат да бъдат изтрети, когато не се използват повече, **WeakMap** и **WeakSet** помагат за избягване на изтичане на паметта.
- **Автоматично изчистване:** За разлика от обикновените Map и Set, които не премахват елементи от паметта при липса на референции, WeakMap и WeakSet гарантират, че обектите ще бъдат премахнати автоматично, когато вече не са необходими.
- **Оптимизация на големи структури от данни:** Когато работим с динамични данни и не искаме да се натоварваме с ръчно изчистване на паметта, тези структури ще помогнат за автоматизация на процеса.

```
let weakmap = new WeakMap();  
let obj = {};  
weakmap.set(obj, 'Some data');
```

```
// След като обектът `obj` не се използва  
никъде другаде, той може да бъде събран от  
Garbage Collector.  
obj = null; // weakmap автоматично изтрива  
елемента
```

```
let weakset = new WeakSet();  
let obj1 = {};  
let obj2 = {};
```

```
weakset.add(obj1);  
weakset.add(obj2);
```

```
// След като обектите не се използват повече, те  
могат да бъдат автоматично премахнати от паметта.  
obj1 = null; // weakset автоматично премахва obj1
```

Автоматично изчистване на паметта (Garbage Collection)

Garbage Collection в JavaScript автоматично изчиства обекти, които вече не се използват от програмата. При **WeakMap** и **WeakSet**, ако обектът, който е част от колекцията, вече не е достъпен извън нея, той ще бъде автоматично премахнат от паметта. Това е важна разлика спрямо стандартните Map и Set, които държат референции към обектите, независимо дали те се използват или не.

```
// Create a new WeakSet
let weakSet = new WeakSet();

// Create objects to use in the WeakSet
let obj1 = { name: "Pranjal" };
let obj2 = { name: "Pranav" };
weakSet.add(obj1);
weakSet.add(obj2);
weakSet.delete(obj1)
obj2=null
console.log(weakSet.has(obj1))//false
console.log(weakSet.has(obj2))//false
```

Благодарим за вниманието!

Имате ли въпроси?

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**

Please keep this slide for attribution