JavaScript po ES6

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

16.10.2025

JavaScript po ES6

Úvod

V této přednášce se podíváme na vývoj JavaScriptu po ES6 (ES2015)

Probereme si:

- Nové funkce a vylepšení jazyka
- Jak se změnil způsob psaní kódu
- Jak se zjednodušilo asynchronní programování
- Jak se zlepšila práce s objekty a třídami

Zaměříme se především na praktické použití nových funkcí

Úvod

Proč je důležité znát nové verze JavaScriptu?

- Modernější a čitelnější kód
- Lepší výkon a efektivita
- Nové možnosti a funkce
- Většina moderních frameworků využívá nové funkce

ES6/ES2015

let a const

ES6 přináší nové způsoby deklarace proměnných: `let` a `const`

```
let x = 10;         // variable that can be changed
const y = 20;       // constant that cannot be changed
```

Hlavní rozdíly oproti `var`:

- Block scope místo function scope
- Nelze redefinovat ve stejném scope
- Temporal Dead Zone
- Přísnější pravidla pro použití

let a const

Příklad problému s `var`:

```
var x = 10;
var x = 20; // OK - can be redefined

function example() {
  console.log(x); // undefined - hoisting
  var x = 30;
}
example();
```

let a const

```
let x = 10;
let x = 20; // Error: Identifier 'x' has already been
declared
```

let a const

```
let x = 10;

function example() {
  console.log(x); // ReferenceError: Cannot access 'x' before
initialization - Temporal Dead Zone
  let x = 30;
}
example();
```

let a const

`const` zabraňuje změně hodnoty u primitivních datových typů a reference u referenčních datových typů

```
const PI = 3.14159;
PI = 3; // TypeError: Assignment to constant variable
```

let a const

Ale pozor na objekty a pole:

```
const user = { name: "John" };
user.name = "Jane"; // OK - changing property, not reference
console.log(user); // { name: "Jane" }
user = { name: "Mike" }; // TypeError: Assignment to constant
variable.
```

```
const arr = [1, 2, 3];
arr.push(4); // OK - modifying array content
console.log(arr); // [1, 2, 3, 4]
arr = [5, 6]; // TypeError: Assignment to constant variable.
```

Block scope a Temporal Dead Zone

Blok kódu je sekvence příkazů uzavřená mezi `{` a `}`

Block scope je typ rozsahu platnosti proměnných, kde jsou proměnné dostupné pouze v bloku, kde byly deklarovány

```
if (true) {
  var x = 10; // function scope
  let y = 20; // block scope
}
console.log(x); // 10
console.log(y); // ReferenceError: y is not defined
```

Block scope a Temporal Dead Zone

To samé platí pro cykly:

```
for (var i = 0; i < 3; i++) {
  // i is available in the whole function scope
}
console.log(i); // 3

for (let j = 0; j < 3; j++) {
  // j is only available in this block scope
}
console.log(j); // ReferenceError: j is not defined
```

Block scope a Temporal Dead Zone

Temporal Dead Zone (TDZ) proměnné je sekce kódu mezi vstupem do scope a deklarací této proměnné

V této době JavaScript ví o proměnné (hoisting), ale zabraňuje jejímu použití v této sekci

```
console.log(x); // undefined - hoisting with var
var x = 1;

console.log(y); // ReferenceError - TDZ with let
let y = 2;
```

Block scope a Temporal Dead Zone

TDZ chrání před použitím proměnných před jejich deklarací:

```
const x = y; // ReferenceError - y is in TDZ
const y = 1;
```

Arrow funkce

Arrow funkce jsou nový způsob zápisu funkcí s kratší syntaxí

```
// Classic function
function add(a, b) {
  return a + b;
}

// Arrow function
const add = (a, b) => {
  return a + b;
}
```

Arrow funkce

```
// Shorthand for simple expressions
const add = (a, b) => a + b;
```

Arrow funkce

Arrow funkce mají několik **speciálních vlastností**:

1. Nemají vlastní `this` - používají `this` z okolního scope
2. Nemají vlastní `arguments` objekt
3. Nemohou být použity jako konstruktory
4. Nemohou být použity jako metody objektů (kvůli `this`)
5. Zkrácený zápis funguje pouze pro jednoduché výrazy

Arrow funkce

Řešení problému s `this` pomocí arrow funkcí:

```
function Person(name) {
  this.name = name;

  // Problem with this in var function
  this.sayHiLater = function() {
    setTimeout(function() {
      // this is not Person
      console.log('Ahoj, ' + this.name);
    }, 1000);
  };
```

Arrow funkce

```
  // Solution with arrow function
  this.sayHiLaterArrow = function() {
    setTimeout(() => {
      // this is Person
      console.log('Ahoj, ' + this.name);
    }, 1000);
  };
}
```

Template literals

Template literals jsou nový způsob zápisu řetězců pomocí zpětných apostrofů (backticks)

```
// Classic way
var name = "John";
var greeting = "Ahoj " + name + "!";

// Template literal
const name = "John";
const greeting = `Ahoj ${name}!`;
```

Template literals

Výhody:

- Interpolace výrazů pomocí `${}`
- Víceřádkové řetězce
- Možnost vnořených výrazů

Template literals

Víceřádkové řetězce bez escapování:

```
// Old way
var html = "<div>\n" +
  "  <h1>Title</h1>\n" +
  "  <p>Paragraph</p>\n" +
  "</div>";
```

Template literals

```
// Template literal
const html = `<div>
  <h1>Title</h1>
  <p>Paragraph</p>
</div>`;
```

Template literals

Interpolace složitějších výrazů:

```
const x = 10;
const y = 20;
console.log(`${x} + ${y} = ${x + y}`);
// "10 + 20 = 30"

const items = ["a", "b", "c"];
console.log(`Items: ${items.join(", ")}`);
// "Items: a, b, c"
```

Template literals

```
const person = { name: "John", age: 30 };
console.log(
  `${person.name} is ${person.age > 18 ? "adult" : "minor"}`
);
// "John is adult"
```

Destructuring

Destructuring umožňuje rozbalit hodnoty z polí nebo vlastnosti z objektů do samostatných proměnných

```
// Array destructuring
const numbers = [1, 2, 3];
const [first, second, third] = numbers;

console.log(first);  // 1
console.log(second); // 2
console.log(third);  // 3
```

Destructuring

```
// Object destructuring
const person = { name: "John", age: 30 };
const { name, age } = person;

console.log(name); // "John"
console.log(age);  // 30
```

Destructuring

Pokročilé možnosti destructuringu:

```
// Variable renaming
const { name: firstName, age: years } = person;
console.log(firstName); // "John"
console.log(years);     // 30

// Default values
const { name, age, country = "USA" } = person;
console.log(country); // "USA"
```

Destructuring

```
// Nested destructuring
const user = {
  id: 1,
  info: { firstName: "John", lastName: "Doe" }
};
const { info: { firstName, lastName } } = user;

console.log(firstName); // "John"
console.log(lastName);  // "Doe"
```

Destructuring

Pokud chcete destrukturovat celý podobjekt a zároveň některé jeho konkrétní vlastnosti, musíte specifikovat oba výrazy

```
const { info, info: { firstName, lastName } } = user;

console.log(info); // { firstName: "John", lastName: "Doe" }
console.log(firstName); // "John"
console.log(lastName);  // "Doe"
```

Destructuring

Destructuring v parametrech funkcí:

```
// Old way
function printUser(user) {
  console.log(user.name + " is " + user.age + " years old");
}
```

Destructuring

```
// With destructuring
function printUser({ name, age }) {
  console.log(`${name} is ${age} years old`);
}

// Call remains the same
printUser({ name: "John", age: 30 });
```

Výhodou přijímání parametrů pomocí jednoho objektu a pomocí destructuringu je, že nezáleží na pořadí parametrů

Default, rest a spread operátory

Defaultní hodnoty parametrů umožňují definovat výchozí hodnoty parametrů

```
// Old way
function greet(name) {
  name = name || "Guest";
  return "Hello, " + name;
}
```

Default, rest a spread operátory

```
// With default parameters
function greet(name = "Guest") {
  return `Hello, ${name}`;
}

console.log(greet()); // "Hello, Guest"
console.log(greet("John")); // "Hello, John"
```

Pomocné funkce pro pole

Ještě před ES6 byly na polích některé užitečné funkce, které si ukážeme

- **map** - transformuje prvky pole (např. vynásobí každý prvek číslem 2)
- **filter** - filtruje prvky pole (vrací pouze prvky, které splňují podmínku)
- **reduce** - redukuje pole na jednu hodnotu (např. součet prvků)
- **some** - testuje, zda alespoň jeden prvek pole splňuje podmínku
- **every** - testuje, zda všechny prvky pole splňují podmínku

Funkce map

Funkce map umožňuje transformovat prvky pole pomocí transformační funkce (předaná jako parametr) a vrátit nové pole s transformovanými prvky

```
const numbers = [1, 2, 3];
const doubled = numbers.map(x => x * 2);
console.log(doubled); // [2, 4, 6]
```

Funkce filter

Funkce filter umožňuje filtrovat prvky pole pomocí filtrační funkce (předané jako parametr)

```
const numbers = [1, 2, 3];
const even = numbers.filter(x => x % 2 === 0);
console.log(even); // [2]
```

Funkce reduce

Funkce reduce umožňuje redukovat pole na jednu hodnotu pomocí redukční funkce (předané jako parametr)

Funkce `reduce` má dva parametry:

- Funkce, která se volá pro každý prvek pole
- Počáteční hodnota

Funkce přijímá dva parametry - aktuální hodnotu a aktuální prvek

Výsledek funkce je akumulační argument pro další volání funkce

Funkce reduce

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce(
  (total, n) => total + n,
  0
);
console.log(sum); // 10
```

Funkce some

Funkce some umožňuje testovat, zda alespoň jeden prvek pole splňuje podmínku

```
const numbers = [1, 2, 3, 4];
const hasEven = numbers.some(x => x % 2 === 0);
console.log(hasEven); // true
```

Funkce every

Funkce every umožňuje testovat, zda všechny prvky pole splňují podmínku

```
const numbers = [1, 2, 3, 4];
const allEven = numbers.every(x => x % 2 === 0);
console.log(allEven); // false
```

Rest parametr

Rest parametr umožňuje zachytit zbývající argumenty do pole

```
// Old way with arguments
function sum() {
  var numbers = Array.prototype.slice.call(arguments);
  return numbers.reduce(function(total, n) {
    return total + n;
  }, 0);
}
```

Rest parametr

```
// With rest parameter
function sum(...numbers) {
  return numbers.reduce((total, n) => total + n, 0);
}

console.log(sum(1, 2, 3, 4)); // 10
```

Rest parametr

```
function sum(a, b, ...numbers) {
  console.log(a); // 1
  console.log(b); // 2
  console.log(numbers); // [3, 4]
  return numbers.reduce((total, n) => total + n, a + b);
}

console.log(sum(1, 2, 3, 4)); // 10
```

Spread operátor

Spread operátor umožňuje rozložit pole nebo objekt

```
// Spread for arrays
const numbers = [1, 2, 3];
console.log(Math.max(...numbers)); // 3

const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4]; // [1, 2, 3, 4]
```

Spread operátor

```
// Spread for objects (ES9/ES2018+)
const defaults = { theme: "dark", lang: "en" };
const user = { ...defaults, name: "John" };
// { theme: "dark", lang: "en", name: "John" }

// Default overridden
const user2 = { ...defaults, theme: "light", name: "Jane" };
// { theme: "light", lang: "en", name: "Jane" }
```

Třídy

Třídy poskytují čistší syntax pro objektově orientované programování

```
// Old way with prototype
function Person(name) {
  this.name = name;
}

Person.prototype.sayHello = function() {
  console.log("Hello, " + this.name);
};
```

Třídy

```
// With classes
class Person {
  constructor(name) {
    this.name = name;
  }

  sayHello() {
    console.log(`Hello, ${this.name}`);
  }
}
```

Třídy

Třídy podporují **dědičnost** pomocí `extends` a `super`

```
class Employee extends Person {
  constructor(name, role) {
    super(name); // Calls parent class constructor
    this.role = role;
  }
  sayHello() {
    super.sayHello(); // Calls parent class method
    console.log(`I am ${this.role}`);
  }
}
```

Třídy

```
const emp = new Employee("John", "developer");
emp.sayHello();
// "Hello, John"
// "I am developer"
```

Třídy

Třídy mohou mít **gettery**, **settery** a **statické metody**

```
class Circle {
  constructor(radius) {
    this._radius = radius;
  }

  // Getter
  get radius() {
    return this._radius;
  }
```

Třídy

```
  // Setter
  set radius(value) {
    if (value <= 0)
      throw new Error("Radius must be positive");
    this._radius = value;
  }
  // Static method
  static createUnit() {
    return new Circle(1);
  }
}
```

Enhanced object literals

ES6 přináší vylepšení pro zápis objektových literálů:

1. Zkrácený zápis vlastností
2. Zkrácený zápis metod
3. Computed property names

Enhanced object literals

Zkrácený zápis vlastností:

```
const name = "John";
const age = 30;

// Old way
const person = {
  name: name,
  age: age
};
```

Enhanced object literals

```
// Shorthand notation
const person = { name, age };
```

Enhanced object literals

Zkrácený zápis metod:

```
// Old way
const calculator = {
  add: function(a, b) {
    return a + b;
  },
  subtract: function(a, b) {
    return a - b;
  }
};
```

Enhanced object literals

```
// Shorthand notation
const calculator = {
  add(a, b) {
    return a + b;
  },
  subtract(a, b) {
    return a - b;
  }
};
```

Computed property names

```
const prefix = "user_";

// Old way
const users = {};
users[prefix + "1"] = "John";
users[prefix + "2"] = "Jane";
```

Computed property names

```
// New way
const users = {
  [prefix + "1"]: "John",
  [prefix + "2"]: "Jane",
  [`${prefix}3`]: "Mike"
};
```

Promises

Promise je objekt reprezentující eventuální dokončení (nebo selhání) asynchronní operace

Při konstrukci Promise se předávají dva callbacky:

- `resolve` - volá se, když operace proběhne úspěšně
- `reject` - volá se, když operace selže

Promises

```
// Creating a Promise
const promise = new Promise((resolve, reject) => {
  // Asynchronous operation
  setTimeout(() => {
    const random = Math.random();
    if (random > 0.5) {
      resolve("Success!");
    } else { reject("Failed!"); }
  }, 1000);
});
```

Promises

S promise se pracuje pomocí metod `then`, `catch` a `finally`:

```
promise
  .then(result => {
    console.log("Success:", result);
  })
  .catch(error => {
    console.log("Error:", error);
  })
  .finally(() => {
    console.log("Finished");
  });
```

Řetězení Promise:

```
fetchUser(1)
  .then(user => fetchUserPosts(user.id))
  .then(posts => renderPosts(posts))
  .catch(error => handleError(error));
```

Promises

Promise kombinátory:

```
// Promise.all - waits for all Promises, none can fail
Promise.all([
  fetchUser(1),
  fetchUser(2),
  fetchUser(3)
]).then(users => console.log(users));
```

Promises

```
// Promise.race - returns first completed Promise
Promise.race([
  fetch("/api/1"),
  fetch("/api/2")
]).then(response => console.log(response));
```

Moduly

ES6 přináší nativní podporu pro **moduly**

Module je **soubor s kódem**, který může obsahovat **exporty** a **importy**

Moduly

Moduly zajišťují:

- **Zapouzdření**: každý modul má svůj vlastní scope - proměnné a funkce definované v modulu nejsou automaticky dostupné v globálním scope
- **Znovupoužitelnost**: kód lze organizovat do samostatných souborů a znovu používat v různých částech aplikace
- **Správa závislostí**: moduly mohou explicitně deklarovat své závislosti pomocí importů

Moduly

```
// math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

export default class Calculator {
  multiply(a, b) {
    return a * b;
  }
}
```

Moduly

```
// main.js
import Calculator, { add, subtract } from './math.js';

console.log(add(2, 3));         // 5
console.log(subtract(5, 2));    // 3

const calc = new Calculator();
console.log(calc.multiply(2, 3)); // 6
```

Moduly

Různé způsoby exportu:

```
// Named exports
export const name = "John";
export function sayHello() { }

// Default export (only one per module)
export default class Person { }
```

Moduly

```
// Export list
const x = 1, y = 2;
export { x, y };

// Export with renaming
export { x as numberX, y as numberY };
```

Moduly

Různé způsoby importu:

```
// Import default export
import Person from './person.js';

// Import named exports
import { name, sayHello } from './utils.js';

// Import with renaming
import { name as userName } from './user.js';
```

Moduly

```
// Import everything
import * as utils from './utils.js';
```

Výsledkem je objekt, který obsahuje všechny exportované položky

```
// Import for side effects
import './polyfills.js';
```

Výsledkem tohoto kódu je spuštění kódu z importovaného souboru v aktuálním kontextu

Iterátory

Iterátor je objekt, který definuje způsob průchodu sekvencí hodnot

Výhodou iterátorů jsou

- Unifikace způsobu průchodu sekvencí hodnot
- **Lazy evaluation** - hodnoty se generují až když jsou potřeba

Iterátory

```
const arr = [1, 2, 3];
const iterator = arr[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 }
```

Iterátory

`Symbol.iterator` je speciální symbol, který se používá k získání iterátoru pro podporované typy

Existuje na Prototype chainu

```
const arr = [1, 2, 3];
console.log(arr.__proto__)
// ...
// Symbol(Symbol.iterator): ƒ values()
// ...
```

Iterátory

Iterátor je pouze **interface** pro průchod sekvencí hodnot

Musí obsahovat funkci `next`, která vrací objekt s vlastnostmi

- `value` indikující aktuální hodnotu
- `done` indikující, zda je iterace dokončena

Generátory

Generátor je funkce, která může být pozastavena a později znovu spuštěna tam, kde byla pozastavena

Označení * před názvem funkce znamená, že se jedná o pozastavitelnou funkci, která automaticky vrací iterátor

```
function* numberGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
```

Generátory

```
const gen = numberGenerator();

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

Ukázka pozastavování

```
function* generator() {
  console.log("generating 1");
  yield 1;
  console.log("generating 2");
  yield 2;
  console.log("generating 2");
  yield 3;
}
```

Ukázka pozastavování

```
const gen = generator();
console.log("consuming 1", gen.next());
// generating 1
// consuming 1 { value: 1, done: false }
console.log("consuming 2", gen.next());
// generating 2
// consuming 2 { value: 2, done: false }
console.log("consuming 3", gen.next());
// generating 3
// consuming 3 { value: 3, done: false }
```

Ukázka pozastavování

```
console.log("consuming 4?", gen.next());
// consuming 4? { value: undefined, done: true }
```

Ukázka pozastavování

Praktické použití generátorů:

```
// Infinite sequence
function* fibonacci() {
  let prev = 0, curr = 1;
  while (true) {
    yield curr;
    [prev, curr] = [curr, prev + curr];
  }
}
```

Ukázka pozastavování

```
const fib = fibonacci();

console.log(fib.next().value); // 1
console.log(fib.next().value); // 1
console.log(fib.next().value); // 2
console.log(fib.next().value); // 3
console.log(fib.next().value); // 5
```

for-of cyklus

for-of cyklus umožňuje iterovat přes iterovatelné objekty

Oproti `for-in` cyklu, který iteruje přes vlastnosti objektu nebo indexy pole, `for-of` cyklus **iteruje přes hodnoty**

for-of cyklus

```
// Array
const numbers = [1, 2, 3];
for (const num of numbers) {
  console.log(num);
}
// 1
// 2
// 3
```

for-of cyklus

```
// String
const str = "Hello";
for (const char of str) {
  console.log(char);
}
// H
// e
// l
// l
// o
```

for-of cyklus

Rozdíl mezi for-in a for-of:

```
const arr = ["a", "b", "c"];

for (const index in arr) {
  console.log(index);
}
// 0 1 2
for (const value of arr) {
  console.log(value);
}
// a b c
```

for-of cyklus

for-of funguje se všemi iterovatelními objekty, které si ukážeme později:

```
// Set
const set = new Set([1, 2, 3]);
for (const num of set) {
  console.log(num);
}
// 1 2 3
```

for-of cyklus

```
// Map
const map = new Map([["a", 1], ["b", 2]]);
for (const [key, value] of map) {
  console.log(key, value);
}
// a 1
// b 2
```

for-of cyklus

```
// Generator
function* gen() {
  yield 1;
  yield 2;
}

for (const value of gen()) {
  console.log(value);
}
// 1 2
```

Symbol

Symbol je nový primitivní datový typ pro unikátní identifikátory

```
// Each Symbol is unique
const sym1 = Symbol();
const sym2 = Symbol();
console.log(sym1 === sym2); // false

// Symbols can have description
const sym3 = Symbol("my symbol");
console.log(sym3.toString()); // "Symbol(my symbol)"
```

Symbol

Použití Symbolů jako klíčů objektů:

```
const MY_KEY = Symbol();
const obj = {
  [MY_KEY]: "secret value"
};

console.log(obj[MY_KEY]); // "secret value"
```

Symbol

```
// Symbols are not visible in regular loops
for (let key in obj) {
  console.log(key); // nothing is printed
}
console.log(Object.keys(obj)); // []
```

Symbol

Well-known Symbols:

```
// Symbol.iterator - defines iterator
const myIterable = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
  }
};
```

Symbol

```
// Symbol.toStringTag - modifies toString() output
class MyClass {
  get [Symbol.toStringTag]() {
    return "MyClass Tag";
  }
}
console.log((new MyClass()).toString());
// [object MyClass Tag]
```

Map, Set, WeakMap, WeakSet

Map je kolekce klíč-hodnota, kde klíče mohou být libovolného typu

Výhody Map oproti objektu:

- Klíče mohou být libovolného typu
- Objekty mohou mít pouze string nebo Symbol
- Map ukládá pořadí uložených hodnot
- Map má dobré pomocné funkce
- Map je škálovatelná a velmi efektivní

Map, Set, WeakMap, WeakSet

```
// Creating a Map
const map = new Map();

// Adding values
map.set("string key", "value");
map.set(42, "number as key");
map.set({}, "object as key");

// Getting values
console.log(map.get("string key")); // value
console.log(map.size); // 3
```

Map, Set, WeakMap, WeakSet

Rozdíly mezi Map a objektem:

```
// Object can only have string/symbol keys
const obj = {
  "1": "string key",
  [Symbol()]: "symbol key"
};
```

Map, Set, WeakMap, WeakSet

```
// Map can have any type of keys
const map = new Map([
  [1, "number key"],
  [{}, "object key"],
  [() => {}, "function key"]
]);
```

Map, Set, WeakMap, WeakSet

Set je kolekce unikátních hodnot

```
// Creating a Set
const set = new Set([1, 2, 2, 3, 3, 3]);
console.log(set.size); // 3

// Adding and removing values
set.add(4);
set.delete(1);
```

Map, Set, WeakMap, WeakSet

```
// Checking existence
console.log(set.has(2)); // true

// Iteration
for (const value of set) {
  console.log(value);
}
// 2 3 4
```

Map, Set, WeakMap, WeakSet

WeakMap a **WeakSet** jsou slabé verze Map a Set

- Klíče musí být objekty
- Reference na objekty jsou "slabé"
  - Garbage collector může objekt uvolnit, pokud na objekt existuje jenom slabá reference
- Nelze iterovat přes hodnoty
- Nemají vlastnost size

Používá se pro privátní data, caching nebo metadata

Map, Set, WeakMap, WeakSet

```
const weakMap = new WeakMap();
let obj = { data: "some data" };

weakMap.set(obj, "metadata");
console.log(weakMap.get(obj)); // "metadata"

obj = null; // object can be garbage collected
// reference in weakmap will also be removed
```

Map, Set, WeakMap, WeakSet

Použití WeakMap pro privátní data:

```
const privateData = new WeakMap();

class Person {
  constructor(name) {
    privateData.set(this, { name });
  }
  getName() {
    return privateData.get(this).name;
  }
}
```

Map, Set, WeakMap, WeakSet

```
const person = new Person("John");
console.log(person.getName()); // "John"
// privateData is automatically cleaned up when person
// is no longer needed
```

Nové metody pro pole

find a **findIndex** - hledání prvků

```
const users = [
  { id: 1, name: "John" },
  { id: 2, name: "Jane" },
  { id: 3, name: "Bob" }
];
```

Nové metody pro pole

```
// find returns the first matching element
const user = users.find(u => u.id === 2);
console.log(user); // { id: 2, name: "Jane" }

// findIndex returns the index of the first matching element
const index = users.findIndex(u => u.name === "Bob");
console.log(index); // 2
```

Nové metody pro pole

includes - kontrola existence prvku (ES7)

```
const numbers = [1, 2, 3];

// Old way
console.log(numbers.indexOf(2) !== -1); // true

// New way
console.log(numbers.includes(2)); // true
console.log(numbers.includes(4)); // false
```

Nové metody pro pole

flat a **flatMap** (ES10) - zploštění vnořených polí

`flat` zploští vnořená pole volitelné hloubky do pole (základní hloubka je 1)

```
const nested = [1, [2, 3], [4, [5, 6]]];
console.log(nested.flat());    // [1, 2, 3, 4, [5, 6]]
console.log(nested.flat(2));   // [1, 2, 3, 4, 5, 6]
```

Nové metody pro pole

`flatMap` Nejdříve transformuje prvky pole a poté je zploští

```
const sentences = ["Hello world", "JavaScript is fun"];
const words = sentences.flatMap(
  s => s.split(" ")
);

console.log(words);
// ["Hello", "world", "JavaScript", "is", "fun"]
```

Nové metody pro objekty

Object.assign - kopírování vlastností objektů

```
const target = { a: 1 };
const source1 = { b: 2 };
const source2 = { c: 3 };

const result = Object.assign(target, source1, source2);
console.log(result); // { a: 1, b: 2, c: 3 }
console.log(target); // { a: 1, b: 2, c: 3 }

// Creating a new object
const clone = Object.assign({}, target);
```

Nové metody pro objekty

Object.fromEntries (ES10) - opak Object.entries

```
const entries = [
  ["name", "John"],
  ["age", 30],
  ["city", "New York"]
];

const obj = Object.fromEntries(entries);
console.log(obj);
// { name: "John", age: 30, city: "New York" }
```

Nové metody pro objekty

```
// Example witn Map
const map = new Map(entries);
const objFromMap = Object.fromEntries(map);
```

ES7/ES2016

ES7 (ES2016) přinesl dvě hlavní novinky:

1. Operátor exponenciace (**)
2. Array.prototype.includes()

Operátor exponenciace (**)

Operátor exponenciace (**) je nový způsob zápisu umocňování

```
// Old way
console.log(Math.pow(2, 3));  // 8

// New way
console.log(2 ** 3);  // 8
```

Operátor exponenciace (**)

Funguje i s přiřazením:

```
let num = 2;
num **= 3;  // num = num ** 3
console.log(num);  // 8
```

Array.prototype.includes

includes je nová metoda pro kontrolu existence prvku v poli

```
const numbers = [1, 2, 3, 4, 5];

// Old way
console.log(numbers.indexOf(3) !== -1);  // true
console.log(numbers.indexOf(6) !== -1);  // false

// New way
console.log(numbers.includes(3));  // true
console.log(numbers.includes(6));  // false
```

Array.prototype.includes

includes má lepší podporu pro `NaN` a funguje s fromIndex parametrem

```
// Problem with NaN in indexOf
const arr = [1, NaN, 2, 3];
console.log(arr.indexOf(NaN));     // -1 (not found)
console.log(arr.includes(NaN));    // true (found)
```

Array.prototype.includes

```
// Using fromIndex
const numbers = [1, 2, 3, 1, 2, 3];

console.log(numbers.includes(2, 3));
// true (searches from index 3)

console.log(numbers.includes(2, -2));
// true (searches last 2 elements)
```

ES8/ES2017

ES8 (ES2017) přinesl několik významných novinek:

1. async/await
2. Object.entries() a Object.values()
3. String padding (padStart, padEnd)
4. Trailing commas v parametrech funkcí
5. Object.getOwnPropertyDescriptors()

Async/Await

async/await je nový způsob práce s Promises, který dělá asynchronní kód čitelnější

Místo řetězení `.then()` lze použít `await` přímo v kódu

Async/Await

```
// Promise way
function fetchUserData() {
  return fetch('/api/user')
    .then(response => response.json())
    .then(user => {
      return fetch(`/api/posts/${user.id}`);
    })
    .then(response => response.json());
}
```

Async/Await

```
// async/await way
async function fetchUserData() {
  const userResponse = await fetch('/api/user');
  const user = await userResponse.json();
  const postsResponse = await fetch(`/api/posts/${user.id}`);
  return postsResponse.json();
}
```

Async/Await

Error handling s async/await:

```
// Promise way
function fetchData() {
  return fetch('/api/data')
    .then(response => response.json())
    .catch(error => {
      console.error('Error:', error);
      throw error;
    });
}
```

Async/Await

```
// async/await way
async function fetchData() {
  try {
    const response = await fetch('/api/data');
    return await response.json();
  } catch (error) {
    console.error('Error:', error);
    throw error;
  }
}
```

Async/Await

Paralelní operace s async/await:

```
// Sequential processing
async function fetchSequential() {
  const user = await fetchUser();
  const posts = await fetchPosts();
  const comments = await fetchComments();
  return { user, posts, comments };
}
```

Async/Await

```
// Parallel processing
async function fetchParallel() {
  const [user, posts, comments] = await Promise.all([
    fetchUser(),
    fetchPosts(),
    fetchComments()
  ]);
  return { user, posts, comments };
}
```

Object.keys

Již v ES5 byla k dispozici metoda `Object.keys`

```
const person = {
  name: 'John',
  age: 30,
  city: 'New York'
};
console.log(Object.keys(person));
// ['name', 'age', 'city']
```

Object.values

Nová metoda **Object.values()** vrací pole hodnot objektu

```
const person = {
  name: 'John',
  age: 30,
  city: 'New York'
};

console.log(Object.values(person));
// ['John', 30, 'New York']
```

Object.values

```
// Practical example
const total = Object.values(person)
  .filter(value => typeof value === 'number')
  .reduce((sum, num) => sum + num, 0);
```

Object.entries

Další nová metoda **Object.entries()** vrací pole dvojic [klíč, hodnota]

```
const person = {
  name: 'John',
  age: 30,
  city: 'New York'
};
```

Object.entries

```
for (const [key, value] of Object.entries(person)) {
  console.log(`${key}: ${value}`);
}
// name: John
// age: 30
// city: New York

// Convert to Map
const map = new Map(Object.entries(person));
```

String padding

padStart a **padEnd** pro zarovnání řetězců

```
// padStart - adds characters to the beginning
console.log('5'.padStart(2, '0'));     // "05"
console.log('123'.padStart(5, '0'));   // "00123"
console.log('5'.padStart(4, '*'));     // "***5"

// padEnd - adds characters to the end
console.log('5'.padEnd(2, '0'));       // "50"
console.log('123'.padEnd(5, '0'));     // "12300"
console.log('5'.padEnd(4, '*'));       // "5***"
```

String padding

Praktické použití string paddingu:

```
// Number formatting
const numbers = [1, 23, 456];
numbers.forEach(num => {
  console.log(num.toString().padStart(3, '0'));
});
// "001"
// "023"
// "456"
```

String padding

```
// Text padding
const items = ['Apple', 'Banana', 'Orange'];
const maxLength = Math.max(
  ...items.map(item => item.length)
);
items.forEach(item => {
  console.log(item.padEnd(maxLength, ' ') + '|');
});
// "Apple  |"
// "Banana |"
// "Orange |"
```

ES9/ES2018

ES9 (ES2018) přinesl několik důležitých vylepšení:

1. Rest/Spread vlastnosti pro objekty
2. Promise.prototype.finally()
3. Asynchronní iterace (for-await-of)
4. Vylepšení regulárních výrazů

Rest/Spread pro objekty

Rest a **spread** operátory nyní fungují i s objekty

```
// Rest pro objekty
const { name, age, ...rest } = {
  name: 'John',
  age: 30,
  city: 'New York',
  country: 'USA'
};
console.log(name); // 'John'
console.log(age);  // 30
console.log(rest); // { city: 'New York', country: 'USA' }
```

Rest/Spread pro objekty

Spread pro objekty:

```
const person = {
  name: 'John',
  age: 30
};

// Copying object
const clone = { ...person };
```

Rest/Spread pro objekty

```
// Extending object
const extended = {
  ...person,
  city: 'New York',
  age: 31  // overrides original value
};

console.log(extended);
// { name: 'John', age: 31, city: 'New York' }
```

Promise.prototype.finally

finally se volá vždy po dokončení Promise, bez ohledu na výsledek

Promise.prototype.finally

```
function fetchData() {
  showLoadingSpinner();

  return fetch('/api/data')
    .then(response => response.json())
    .catch(error => {
      console.error('Error:', error);
      throw error;
    })
    .finally(() => { hideLoadingSpinner(); });
}
```

Asynchronní iterace

for-await-of umožňuje iterovat přes asynchronní iterátory

```
async function* asyncGenerator() {
  yield Promise.resolve(1);
  yield Promise.resolve(2);
  yield Promise.resolve(3);
}
```

Asynchronní iterace

```
for await (const num of asyncGenerator()) {
  console.log(num);
}
// 1
// 2
// 3
```

Asynchronní iterace

Praktické použití s fetch:

```
async function* fetchCommits(repo) {
  let url = `https://api.github.com/repos/${repo}/commits`;
  while (url) {
    const response = await fetch(url);
    const data = await response.json();
    for (let commit of data) { yield commit; }
    // Reads the link header to get the next page
    url = getNextPage(response.headers);
  }
}
```

Asynchronní iterace

```
async function showCommits(repo) {
  try {
    for await (const commit of fetchCommits(repo)) {
      console.log(commit.sha);
    }
  } catch (err) {
    console.error(err);
  }
}
```

ES10/ES2019

ES10 (ES2019) přinesl několik užitečných vylepšení:

1. Array.prototype.flat() a flatMap()
2. Object.fromEntries()
3. String.prototype.trimStart() a trimEnd()
4. Optional catch binding
5. Stabilní Array.sort()

Array.prototype.flat a flatMap

flat zploští vnořená pole do jednoho pole

```
const arr = [1, 2, [3, 4, [5, 6]]];

console.log(arr.flat());     // [1, 2, 3, 4, [5, 6]]
console.log(arr.flat(2));    // [1, 2, 3, 4, 5, 6]
console.log(arr.flat(Infinity)); // [1, 2, 3, 4, 5, 6]

// Removes empty items
const sparse = [1, 2, , 4, 5];
console.log(sparse.flat()); // [1, 2, 4, 5]
```

Array.prototype.flat a flatMap

flatMap kombinuje map a flat do jedné operace

```
// Without flatMap
const sentences = ["Hello world", "JavaScript is fun"];
const words = sentences
  .map(s => s.split(" "))
  .flat();

// With flatMap
const words2 = sentences.flatMap(s => s.split(" "));
console.log(words2); // ["Hello", "world", "JavaScript",
"is", "fun"]
```

```
// Practical example
const numbers = [1, 2, 3];
console.log(numbers.flatMap(x => [x, x * 2]));
// [1, 2, 2, 4, 3, 6]
```

Object.fromEntries

Object.fromEntries je opak Object.entries - převádí pole dvojic na objekt

```
const entries = [
  ['name', 'John'],
  ['age', 30],
  ['city', 'New York']
];
const obj = Object.fromEntries(entries);
console.log(obj);
// { name: "John", age: 30, city: "New York" }
```

Object.fromEntries

```
// Filtering object
const person = {
  name: 'John', age: 30,
  password: '123456'
};

const filtered = Object.fromEntries(
  Object.entries(person)
    .filter(([key]) => !key.includes('password'))
);
console.log(filtered); // { name: "John", age: 30 }
```

Object.fromEntries

```
// Transforming values
const prices = {
  banana: 1, apple: 2, orange: 3
};

const doubled = Object.fromEntries(
  Object.entries(prices)
    .map(([key, value]) => [key, value * 2])
);
console.log(doubled);
// { banana: 2, apple: 4, orange: 6 }
```

String.prototype.trimStart a trimEnd

trimStart a **trimEnd** odstraňují bílé znaky ze začátku/konce řetězce

```
const str = '   Hello world   ';

// Old methods
console.log(str.trimLeft());  // "Hello world   "
console.log(str.trimRight()); // "   Hello world"
// New methods
console.log(str.trimStart()); // "Hello world   "
console.log(str.trimEnd());   // "   Hello world"
// Combination
console.log(str.trimStart().trimEnd()); // "Hello world"
```

ES11/ES2020

ES11 (ES2020) přinesl několik významných novinek:

1. BigInt
2. Nullish coalescing operátor (??)
3. Optional chaining (?.)
4. Promise.allSettled
5. globalThis
6. Dynamic import

BigInt

BigInt je nový datový typ pro práci s velkými celými čísly

```
// Creating BigInt
const bigInt = 9007199254740991n;  // n suffix
const alsoHuge = BigInt(9007199254740991);  // function

// Operations with BigInt
console.log(bigInt + 1n); // 9007199254740992n
console.log(bigInt * 2n); // 18014398509481982n
```

BigInt

```
// Comparison
console.log(1n < 2n);  // true
console.log(2n > 1n);  // true
console.log(2n === 2); // false
console.log(2n == 2);  // true
```

Nullish coalescing operátor (??)

Nullish coalescing operátor (??) poskytuje způsob, jak pracovat s `null` nebo `undefined`

```
// Problem with || operator
const count = 0;
const result = count || 42;  // 42

// Solution with ?? operator
const count = 0;
const result = count ?? 42;  // 0
```

Nullish coalescing operátor (??)

```
const name = null;
const text = name ?? "anonymous";  // "anonymous"
```

Nullish coalescing operátor (??)

Rozdíl mezi ?? a ||:

```
// || returns first "truthy" value
console.log(false || 'default');     // "default"
console.log(0 || 'default');         // "default"
console.log('' || 'default');        // "default"
console.log(null || 'default');      // "default"
console.log(undefined || 'default'); // "default"
```

Nullish coalescing operátor (??)

```
// ?? returns first value that is not null/undefined
console.log(false ?? 'default');     // false
console.log(0 ?? 'default');         // 0
console.log('' ?? 'default');        // ""
console.log(null ?? 'default');      // "default"
console.log(undefined ?? 'default'); // "default"
```

Optional chaining (?.)

Optional chaining (?.) umožňuje bezpečný přístup k vnořeným vlastnostem

```
const user = {
  name: 'John',
  address: {
    street: 'Main St'
  }
};
```

Optional chaining (?.)

```
// Old way (junior level developer)
const zipCode = user ?
                user.address ?
                user.address.zipCode : null : null;

// Old way
const zipCode = user && user.address && user.address.zipCode;

// New way
const zipCode = user?.address?.zipCode;
```

Optional chaining (?.)

Optional chaining s metodami a poli:

```
// Methods
const result = obj.method?.();
// undefined if method doesn't exist

// Array
const item = arr?.[0];
// undefined if arr is null/undefined
```

Optional chaining (?.)

```
// Combination
const zip = user?.address?.zip?.toString();

// Practical example
const userRole =
  user?.permissions?.role?.toUpperCase() ?? 'GUEST';
```

Promise.allSettled

Promise.allSettled čeká na dokončení všech promises, bez ohledu na výsledek - můžou selhat

```
const promises = [
  fetch('/api/user'),
  fetch('/api/posts'),
  fetch('/api/comments')
];
```

Promise.allSettled

```
// Promise.all fails on first error
Promise.all(promises)
  .then(results =>
    console.log('All succeeded:', results)
  )
  .catch(error =>
    console.log('At least one failed:', error)
  );
```

Promise.allSettled

```
// Promise.allSettled waits for all
Promise.allSettled(promises)
  .then(results => {
    results.forEach(result => {
      if (result.status === 'fulfilled') {
        console.log('Success:', result.value);
      } else {
        console.log('Error:', result.reason);
      }
    });
  });
```

ES12/ES2021

ES12 (ES2021) přinesl několik užitečných vylepšení:

1. Logical assignment operátory (&&=, ||=, ??=)
2. Numeric separators
3. String.prototype.replaceAll
4. Promise.any
5. WeakRef a FinalizationRegistry

Logical assignment operátory

Logical assignment operátory kombinují logické operátory s přiřazením

```
// OR assignment (||=)
let x = null;
x ||= 42;  // x = x || 42
console.log(x);  // 42
```

Logical assignment operátory

```
// AND assignment (&&=)
let y = 1;
y &&= 2;  // y = y && 2
console.log(y);  // 2

// Nullish assignment (??=)
let z = null;
z ??= 42;  // z = z ?? 42
console.log(z);  // 42
```

Logical assignment operátory

Praktické použití logical assignment operátorů:

```
const user = {
  name: 'John',
  settings: {
    theme: 'light'
  }
};
```

Logical assignment operátory

```
// Setting default values
user.settings.notifications ??= true;

// Conditional updates
user.isAdmin &&= user.permissions.includes('admin');

// Preserve existing value
user.role ||= 'user';
```

Numeric separators

Numeric separators (_) zlepšují čitelnost čísel

```
// Large numbers
const billion = 1_000_000_000;
const budget = 1_234_567.89;
// Binary numbers
const binary = 0b1010_0001_1000_0101;
// Hexadecimal numbers
const hex = 0xFF_EC_DE_5E;
// BigInt
const bigNumber = 1_234_567_890_123_456_789n;
```

String.prototype.replaceAll

replaceAll nahradí všechny výskyty podřetězce

```
// Old way
const text = 'hello hello hello';
console.log(text.replace(/hello/g, 'hi')); // 'hi hi hi'

// New way
console.log(text.replaceAll('hello', 'hi')); // 'hi hi hi'
```

Promise.any

Promise.any vrátí první úspěšně dokončený Promise

```
const promises = [
  fetch('https://api1.example.com/data'),
  fetch('https://api2.example.com/data'),
  fetch('https://api3.example.com/data')
];
```

Promise.any

```
// Returns first successful result
Promise.any(promises)
  .then(firstSuccess => {
    console.log('First success:', firstSuccess);
  })
  .catch(error => {
    // AggregateError if all fail
    console.log('All promises failed:', error);
  });
```

Promise.race a Promise.any

```
// Promise.race returns first completed (success or error)
Promise.race([
  fetch('/endpoint-1'),
  fetch('/endpoint-2')
]).then(/* first completed */);

// Promise.any returns first successful
Promise.any([
  fetch('/endpoint-1'),
  fetch('/endpoint-2')
]).then(/* first successful */);
```

Promise.race a Promise.any

```
// Practical example - fallback API
Promise.any([
  fetch('https://api.example.com'),
  fetch('https://api.secondary.example.com')
]).then(response => response.json());
```

ES13/ES2022

ES13 (ES2022) přinesl několik významných vylepšení:

1. Top-level await
2. Array.prototype.at()
3. Object.hasOwn()
4. Class fields a private methods
5. Static initialization blocks

Top-level await

Top-level await umožňuje použít await **mimo async funkce**

```
// Before - await only in async functions
async function getData() {
  const response = await fetch('/api/data');
  return response.json();
}

// Now - await directly in module
const response = await fetch('/api/data');
const data = await response.json();
export { data };
```

Top-level await

Praktické použití top-level await:

```
// config.js
const config = await fetch('/api/config');
export default await config.json();

// database.js
const connection = await createConnection();
export { connection };
```

Top-level await

```
// app.js
import { data } from './data.js';
// Module loads after await completes
console.log(data);
```

Array.prototype.at()

at() poskytuje lepší způsob přístupu k prvkům pole pomocí indexu

```
const arr = [1, 2, 3, 4, 5];

// Old way
console.log(arr[arr.length - 1]); // 5
console.log(arr[arr.length - 2]); // 4

// New way
console.log(arr.at(-1)); // 5
console.log(arr.at(-2)); // 4
console.log(arr.at(1));  // 2
```

Class fields a private methods

Private fields a **methods** přinášejí skutečné soukromé vlastnosti do tříd

Class fields a private methods

```
class Counter {
  #count = 0;      // private field
  #increment() {  // private method
    this.#count++;
  }

  getCount() { return this.#count; }

  increment() { this.#increment(); }
}
```

Class fields a private methods

```
const counter = new Counter();

counter.increment();
console.log(counter.getCount()); // 1
console.log(counter.#count); // SyntaxError
```

Singleton pomocí private fields

Statické private fields a methods:

```
class Singleton {
  static #instance;

  constructor() {
    if (Singleton.#instance) {
      throw new Error(
        'Use Singleton.getInstance() instead of new.'
      );
    }
  }
```

Singleton pomocí private fields

```
  static getInstance() {
    if (!Singleton.#instance) {
      Singleton.#instance = new Singleton();
    }
    return Singleton.#instance;
  }
}

const instance = Singleton.getInstance();
const instance2 = Singleton.getInstance();
console.log(instance === instance2); // true
```

ES14/ES2023 a ES15/ES2024

Nejnovější verze ES14 (ES2023) a ES15 (ES2024) přinesly:

ES14/ES2023:

- Array.prototype.toSorted, toReversed, with
- Array.prototype.findLast a findLastIndex
- Shebang podpora

ES15/ES2024:

- Object.groupBy a Map.groupBy
- Promise.withResolvers

Immutable metody polí

Immutable metody polí nemodifikují původní pole, ale vrací nová

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

// Old methods modify original array
numbers.reverse();
console.log(numbers); // [4, 3, 2, 1]
// New methods create new array
const original = [1, 2, 3, 4];
const reversed = original.toReversed();
console.log(original); // [1, 2, 3, 4]
console.log(reversed); // [4, 3, 2, 1]
```

Immutable metody polí

```
// Other new methods
console.log(original.toSorted(
  (a, b) => b - a)
); // [4, 3, 2, 1]

console.log(original.with(1, 10)); // [1, 10, 3, 4]
```

Seskupovací metody

Object.groupBy a Map.groupBy:

```
const items = [
  { type: "fruit", name: "apple" },
  { type: "vegetable", name: "carrot" },
  { type: "fruit", name: "banana" }
];
```

Seskupovací metody

```
// Object.groupBy
const grouped = Object.groupBy(items, item => item.type);
// {
//   fruit: [
//     { type: "fruit", name: "apple" },
//     { type: "fruit", name: "banana" }
//   ],
//   vegetable: [
//     { type: "vegetable", name: "carrot" }
//   ]
// }
```

Seskupovací metody

```
// Map.groupBy
const groupedMap = Map.groupBy(items, item => item.type);
// Map {
//   fruit: [
//     { type: "fruit", name: "apple" },
//     { type: "fruit", name: "banana" }
//   ],
//   vegetable: [
//     { type: "vegetable", name: "carrot" }
//   ]
// }
```

Shrnutí

Vývoj JavaScriptu

1. Lepší syntax a čitelnost kódu
   - let/const, arrow funkce, template literals
2. Moderní funkce pro práci s daty
   - destructuring, spread/rest, Map/Set
3. Vylepšené OOP
   - třídy, private fields, static blocks
4. Asynchronní programování
   - Promises, async/await, top-level await
5. Nové operátory a metody
   - ??, ?., ||=, ??=, at(), groupBy()

Vývoj JavaScriptu

Doporučení pro moderní JavaScript:

1. Používejte nové funkce pro čitelnější kód
2. Věnujte pozornost kompatibilitě prohlížečů
3. Využívejte transpilery (Babel) pro starší prohlížeče
4. Sledujte nové verze a vylepšení
5. Používejte TypeScript pro lepší typovou bezpečnost