Základy JavaScriptu

a JS před ES6

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

Ing. Luboš Zápotočný

02.10.2025

Základy JavaScriptu a JS před ES6

Motivace

Proč je vhodné znát starší verze JavaScriptu?

- Získání **základních znalostí** o JavaScriptu
- **Porozumění novým verzím** JavaScriptu a proč se něco mění
- **Udržování starších projektů**

Vývoj jazyka JavaScript

JavaScript byl vytvořen v roce **1995** společností Netscape, která ho používala pro svůj webový prohlížeč Netscape Navigator

Cílem bylo umožnit programátorům psát skripty pro tento prohlížeč

V roce 1997 byla JavaScript standardizována jako ECMA-262 (ECMAScript) a od té doby se neustále rozvíjí

Vývoj jazyka JavaScript

- ES1 (1997) - první verze JavaScriptu
- ES2 (1998) - nevýznamné změny
- ES3 (2000) - regulární výrazy, `try/catch`, `switch`, `do-while`
- ES4 (2004) - nevydáno
- ES5 (2009) - strict mode, `JSON`, pomocné funkce pro řetězce a pole

Vývoj jazyka JavaScript

- ES6/ES2015 (2015)
  - `let` a `const`
  - arrow funkce
  - `for-of` cyklus
  - třídy
  - moduly
  - `Promise`
  - generátory
  - `Map`, `Set`, `WeakMap`, `WeakSet`
  - ...

Vývoj jazyka JavaScript

- ES7/ES2016 (2016)
  - ** operátor exponenciace
  - `includes` na polích
- ES8/ES2017 (2017)
  - `async` a `await`
  - `Object.entries` a `Object.values`
  - `Object.getOwnPropertyDescriptors`

Vývoj jazyka JavaScript

- ES9/ES2018 (2018)
  - `...` spread a rest operátor
  - `for-await-of`
  - `Promise.prototype.finally`
- ES10/ES2019 (2019)
  - `Array.prototype.flat` a `Array.prototype.flatMap`
  - `Array.prototype.fromEntries`
  - stabilní řazení pomocí `Array.sort`
  - optional catch binding

Vývoj jazyka JavaScript

- ES11/ES2020 (2020)
  - `BigInt` datový typ
  - nullish coalescing operator `??`
  - optional chaining operator `?.`
- ES12/ES2021 (2021)
  - `Promise.any`
  - přiřazovací výrazy `??=`, `&&=` a `||=`
  - `WeakRef` a `FinalizationRegistry`
  - oddělovač v číslech (např. `1_000_000`)

Vývoj jazyka JavaScript

- ES13/ES2022 (2022)
  - top-level `await`
  - `at` na řetězcích a polích
  - `Object.hasOwn`
- ES14/ES2023 (2023)
  - `toSorted` a `toReversed` na polích
  - `with`, `findLast` a `findLastIndex` na polích
  - `toSpliced` na polích
  - podpora shebang `#!` ve spustitelných souborech

Vývoj jazyka JavaScript

- ES15/ES2024 (2024)
  - `Object.groupBy` a `Map.groupBy`
  - `Promise.withResolvers`
- ES16/ES2025 (2025)
  - `Iterator`
  - `Promise.try`
  - rozšíření `Set`

Základní syntaxe JavaScriptu (před ES6)

Proměnné

Proměnné jsou **místa v paměti**, která mohou obsahovat různé hodnoty

JavaScript je **dynamicky typovaný** jazyk

To znamená, že **typ proměnné** se určuje **při běhu programu** a **může se měnit**

Proměnné se **deklarují** pomocí klíčového slova `var`

```
var x;
```

Proměnné

Deklarovaná proměnná má **implicitní hodnotu** `undefined`

```
console.log(x);
```

```
undefined
```

Inicializovat proměnnou můžeme pomocí operátoru `=`

```
x = 10;
console.log(x);
```

```
10
```

Proměnné

Samozřejmě můžeme proměnné **deklarovat** a **inicializovat** najednou

```
var x = 10;

console.log(x);
```

```
10
```

Proměnné

Typ proměnné lze zjistit pomocí operátoru `typeof`

```
console.log(typeof x);
```

```
number
```

Při změně obsahu proměnné se **automaticky** změní i její typ

```
x = "Hello";
console.log(typeof x);
```

```
string
```

Základní datové typy

- Nedefinovaný (`undefined`)
- Číslo (`number`)
- Řetězec (`string`)
- Logická hodnota (`boolean`)
- Objekt (`object`)
- Funkce (`function`)

Zajímavosti o datových typech

Čísla s desetinnou čárkou jsou typu `number`

```
console.log(typeof 1.5); // number
```

`null` je typu **`object`**

```
console.log(typeof null); // object
```

Pole jsou typu **`object`**

```
console.log(typeof []); // object
```

Zajímavosti o datových typech

`undefined` je typu `undefined`

```
console.log(typeof undefined); // undefined
```

`NaN` (Not a Number) je typu `number`

```
console.log(typeof NaN); // number
```

`Infinity` a `-Infinity` jsou typu `number`

```
console.log(typeof Infinity); // number
console.log(typeof -Infinity); // number
```

Primitivní datové typy

Primitivní datové typy

- Číslo (`number`)
- Řetězec (`string`)
- Logická hodnota (`boolean`)
- Null (`null`)
- Undefined (`undefined`)

Primitivní datové typy jsou **immutable** (neměnné)

Jejich hodnoty jsou **kopírovány** při přiřazení

Primitivní datové typy

```
var x = 10;
var y = x;
y = 20;

console.log(x); // 10
console.log(y); // 20
```

Hodnota v proměnné x se nezměnila, protože y je samostatná proměnná a při její inicializaci se hodnota proměnné x zkopírovala

Referenční datové typy

Referenční datové typy

- Pole (`array`)
- Objekt (`object`)
- Funkce (`function`)

Referenční datové typy

```
var x = [1, 2, 3];
var y = x;
y[0] = 10;

console.log(x); // [10, 2, 3]
console.log(y); // [10, 2, 3]
```

Hodnota v proměnné x se změnila, protože y je stejně jako x pouze odkaz na paměťové místo

Referenční datové typy

```
var x = { a: 1, b: 2, c: 3 };
var y = x;
y.a = 10;

console.log(x); // { a: 10, b: 2, c: 3 }
console.log(y); // { a: 10, b: 2, c: 3 }
```

Operátory

Operátory

Klíčové slovo je speciální sekvence symbolů, které mají určitý význam v programovacím jazyce

Operátor je speciální sekvence symbolů, která provádí určitou operaci

Operátory

Aritmetické operátory

```
console.log(1 + 1); // 2
console.log(3 - 1); // 2
console.log(2 * 2); // 4
console.log(10 / 2); // 5
console.log(10 % 3); // 1
```

Operátory

Operátory porovnání

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

Operátory

Rozdíl mezi **dvojitou a trojitou rovností** je, že trojitá rovnost porovnává typ obou porovnávaných hodnot a pokud se typ nerovná, ihned oznamuje výsledek `false`

Dvojitá rovnost se v případě rozdílných typů nejdříve pokusí převést obě strany na stejný typ a až poté porovnat obsah obou stran

Operátory

Logické operátory

```
console.log(true && true); // true
console.log(true && false); // false
console.log(true || false); // true
console.log(!true); // false
```

Operátory

Operátory přiřazení

```
var x = 1;
        console.log(x); // 1
x += 4; console.log(x); // 5
x -= 1; console.log(x); // 4
x *= 2; console.log(x); // 8
x /= 2; console.log(x); // 4
x %= 2; console.log(x); // 0
```

Operátory inkrementace a dekrementace

Prefixové a **postfixové** operátory inkrementace a dekrementace

```
var x = 1;
console.log(x++); // 1
console.log(x); // 2
```

```
var x = 1;
console.log(++x); // 2
console.log(x); // 2
```

Operátory inkrementace a dekrementace

```
var x = 1;
console.log(x--); // 1
console.log(x); // 0
```

```
var x = 1;
console.log(--x); // 0
console.log(x); // 0
```

Podmínky

Podmínky

Podmínky se vyhodnocují pomocí klíčového slova `if`

```
if (condition) {
  // condition is true
}
```

`condition` je výraz, který se vyhodnotí na hodnotu `true` nebo `false`

Kód v bloku `{}` se provede, pokud se `condition` vyhodnotí na `true` a tento blok kódu se nazývá **true branch**

Podmínky

Druhá část podmínky - **false branch** - je **volitelná** a lze ji definovat pomocí klíčového slova `else`

```
if (condition) {
  // condition is true
} else {
  // condition is false
}
```

Podmínky

Podmínky jsou často komplikovanější a je potřeba rozlišovat více než dvě situace, k tomu slouží `else if`

```
if (condition1) {
  // condition1 is true
} else if (condition2) {
  // condition2 is true
} else {
  // condition1 and condition2 are both false
}
```

Podmínky

`else if` **není klíčové slovo**, jedná se pouze o proces **parsování** kódu a je to ekvivalentní následujícímu kódu

```
if (condition1) {
} else {
  if (condition2) {
  } else {
    // condition1 and condition2 are both false
  }
}
```

Alternativy

Alternativa k sekvenci mnoha `else if` je `switch`

Definicí několika `case` hned pod sebou docílíte efektu „propadávání“ (**fallthrough**)

Pokud chcete zamezit propadávání, tak použijte klíčové slovo `break`

Pokud žádný `case` není splněn, tak se provede `default`

Alternativy

```
switch (expression) {
  case value1:
    // code
    break;
  case value2:
  case value3:
    // code
    break;
  default:
    // code
}
```

Ternární operátor

Ternární operátor je operátor, který se skládá ze tří částí: `condition`, `trueExpression` a `falseExpression`

```
condition ? trueExpression : falseExpression
```

Ternární operátor

Většinou se používá pro **přiřazení hodnoty** do proměnné

```
var x = y > 10 ? y - 10 : y;
```

Předchozí kód je ekvivalentní následujícímu:

```
if (y > 10) {
  x = y - 10;
} else {
  x = y;
}
```

Ternární operátor

Druhé časté použití je **v podmínkách**

```
if (userAge >= (isHorrorMovie ? 18 : 12)) {
  // can purchase ticket
} else {
  // cannot purchase ticket
}
```

Cykly

Cykly

Javascript obsahuje standardní cykly: `for`, `while` a `do-while`

```
for (var i = 0; i < 10; i++) {
  // code to be executed
}
```

```
while (condition) {
  // code to be executed
}
```

```
do {
  // code to be executed
} while (condition);
```

Cykly

Obsahuje ale také cyklus `for-in` (ES3) a `for-of` (ES6), který si ukážeme v příští přednášce

Cyklus `for-in` slouží pro iteraci nad **vlastnostmi** objektu

```
var obj = { a: 1, b: 2, c: 3 };
for (var key in obj) {
  console.log(key, "=>", obj[key]);
}
// a => 1
// b => 2
// c => 3
```

Funkce

Základy funkcí

Funkce jsou **znovupoužitelné bloky kódu**

V JavaScriptu jsou funkce **first-class citizens**

To znamená, že s nimi můžeme zacházet jako s hodnotami:

- Přiřadit je do proměnné
- Předat je jako parametr
- Vrátit je z jiné funkce

Deklarace funkcí

Deklarace funkce (function declaration) se skládá z klíčového slova `function`, názvu funkce, parametrů (volitelné) a těla funkce

```
function functionName(parameter1, parameter2) {
  // code to be executed
  return someResult;
}
```

Příklad deklarované funkce

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

var result = add(3, 5);
console.log(result); // 8
```

Funkční výraz

Funkční výraz (function expression) je funkce, která je přiřazena do proměnné

```
var add = function(a, b) {
  return a + b;
}

var result = add(3, 5);
console.log(result); // 8
```

Parametry a argumenty

Parametry jsou **proměnné**, které jsou definovány v deklaraci funkce

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

Argumenty jsou **hodnoty**, které jsou předány funkci při volání

```
add(3, 5); // 3 a 5 are arguments
```

Flexibilita parametrů

JavaScript je velmi **flexibilní** v počtu parametrů a nebo argumentů a „nestěžuje si“, když jich na jedné ze stran je více nebo méně než na druhé

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

console.log(add(3, 5, 7)); // 8
// Third argument is not used at all
```

Hodnoty parametrů

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

console.log(add(3));
```

Jakou výslednou hodnotu byste očekávali? A proč?

Problém nastává, když funkce definuje nějaký počet parametrů, ale volající funkci předá **méně argumentů**

```
console.log(add(3)); // NaN
```

Hodnoty parametrů

JavaScript **není striktně typovanný** jazyk a pokud nepředáme druhý parametr, tak si **nemůže domyslet**, že druhý parametr by měl také být `number` a případně dostadit vhodnou iniciální hodnotu

```
function add(a, b) {
  console.log("a", a, typeof a);
  console.log("b", b, typeof b);
  return a + b;
}
```

Hodnoty parametrů

```
console.log(add(3)); // NaN
// a 3 number
// b undefined undefined
```

Defaultní hodnoty parametrů

V JavaScriptu **před ES6** se tyto situace musely řešit pomocí zajímavého **triku**

```
function add(a, b) {
  a = a || 0;
  b = b || 0;
  return a + b;
}

console.log(add(3)); // 3
console.log(add(3, 5)); // 8
console.log(add()); // 0
```

Vysvětlení triku

Trik je založen na tom, že samotný logický operátor `||` **nevrací přímo boolean hodnoty**, ale vrací levou porovnávanou hodnotu, pokud je **truthy** nebo pravou stranu, pokud je levá strana **falsy**

Truthy hodnoty jsou takové, které se při přetypování na boolean stávají `true`

Falsy hodnoty jsou takové, které se při přetypování na boolean stávají `false`

Příklad truthy a falsy hodnot

```
console.log(true || 42); // true
console.log(false || 42); // 42

console.log([] || "Hello"); // []
console.log({} || "Hello"); // {}

console.log(null || "Hello"); // "Hello"
console.log(undefined || "Hello"); // "Hello"
console.log(0 || "Hello"); // "Hello"
console.log("" || "Hello"); // "Hello"
console.log(NaN || "Hello"); // "Hello"
```

Příklad truthy a falsy hodnot

Stejný „trik“ používá většina frontendových frameworků pro pro renderování podmíněného obsahu

Tam ale používá operátor &&

```
{isLoading && <div>Loading...</div>}
```

Dynamický počet parametrů

JavaScript nekontroluje kolik parametrů funkce má a nebo kolik argumentů funkci při volání předáváme

Ale pomocí speciálního objektu `arguments` můžeme získat potřebné informace o tom, kolik argumentů bylo funkci předáno

Dynamický počet parametrů

```
function varfn() {
  console.log(arguments);
}

varfn(-4, 9, 3);
// [Arguments] { '0': -4, '1': 9, '2': 3 }
//   .length: 3
```

Na tomto speciálním objektu můžeme tedy **získat počet argumentů** funkce pomocí přečtení hodnoty jeho vlastnosti `length` a **pomocí cyklu iterovat** přes tento objekt a číst jednotlivé argumenty

```
function varfn() {
  console.log(arguments.length);
  for (var i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
}

varfn(-4, 9, 3);
// 3
// -4
// 9
// 3
```

Klíčové slovo return a návratová hodnota

Klíčové slovo `return` se používá k ukončení funkce a **vrácení hodnoty**

Pokud funkce nevrací zádnou hodnotu, tak se **implicitně** vrací `undefined`

```
function add(a, b) {
  var result = a + b;
}

console.log(add(3, 5)); // undefined
```

Hoisting

Hoisting

Hoisting, česky „zvedání“, je **mechanismus**, který umožňuje používat funkce ještě předtím, než jsou deklarovány

```
myfn(); // "Hello"

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

Při překládání kódu jsou nalezené funkce přesunuty (**hoisted**) na začátek skriptu

Hoisting

Hoisting **nefunguje s funkčními výrazy**

```
myfn(); // TypeError: myfn is not a function

var myfn = function() {
  console.log("Hello");
}
```

Rozsah platnosti proměnných

Rozsah platnosti proměnných

Rozsah platnosti (scope) proměnných je **oblast**, ve které je proměnná viditelná a nebo přístupná

Javascript **před ES6** umožňoval **pouze dva typy** rozsahu platnosti:

- Global scope
- Function scope

JavaScript **po ES6** přidal možnost **blokové platnosti** (block scope) o které si řekneme v příští přednášce

Global scope

Globální proměnné jsou proměnné definované na nejvyšší úrovni skriptu

```
var counter = 0;

function increment() {
  counter++;
}

increment(); increment(); increment();
console.log(counter); // 3
```

Global scope

Pozor! Globální proměnnou lze založit i uvnitř funkce, pokud zapomeneme použít klíčové slovo `var`, což může být zdrojem chyb

```
function myfn() {
  counter = 10;
}

// console.log(counter);
//   ReferenceError: counter is not defined
myfn();
console.log(counter); // 10
```

Function scope

Funkční proměnné jsou proměnné definované uvnitř funkce a jsou viditelné pouze uvnitř této funkce

Neovlivňují globální proměnné a jejich obsah zaniká po skončení funkce

Function scope

```
var counter = 0;

function myfn() {
  var counter = 10;
  console.log(counter); // 10
}

console.log(counter); // 0
myfn(); // 10
console.log(counter); // 0
```

Řetězení rozsahu platnosti

Rozsah platnosti proměnných se **řetězí** (scope chaining) od nejvnitřnějšího po nejvyšší úrovně

Pokud tedy kód má pracovat s proměnnou, pokusí se ji najít v nejbližším rozsahu platnosti a pokud ji nenajde pokračuje dále až do nejvyšší úrovně (globální)

Pokud proměnnou v nějakém vnitřním rozsahu a stejně pojmenované proměnná existuje také ve vnějším rozsahu platnosti, tak se použije ta vnitřnější a říká se tomu **shadowing** (zastínění)

Řetězení rozsahu platnosti

```
var modifier = 0.5;
function calc() {
  var x = 10;
  function subcalc() {
    var y = 20;
    return y + x * modifier;
  }
  return 1 + subcalc();
}

console.log(calc()); // 26
```

Shadowing

```
var modifier = 0.5;
function calc() {
  var x = 10;
  function subcalc() {
    var x = 20;
    return x + x * modifier;
  }
  return x + subcalc();
}

console.log(calc()); // 40
```

Anonymní funkce

Anonymní funkce

Anonymní funkce je funkce, která nemá název

Takové funkce jsou užitečné pro **předávání jako hodnoty** nebo jako callbacky

Připomenutí: funkce jsou **first-class citizens** a lze je tedy

- Přiřadit do proměnné
- Předat jako parametr
- Vrátit z funkce

Anonymní funkce

Anonymní funkce jsme již viděli ve formě **funkčního výrazu**, pokud byly přiřazeny do proměnné

```
var myfn = function() {
  console.log("Hello");
}
```

Anonymní funkce

Předáváním funkce jako parametr jiné funkcí získáváme takzvané **callbacky**

```
function calc(a, b, callback) {
  var result = a + b;
  callback(result);
}


calc(10, 20, function(result) {
  console.log(result); // 30
});
```

Anonymní funkce

Vrácením funkce z funkce získáváme takzvané **closure**

Closure

Closure je funkce, která může mít i přístup k proměnným, které jsou definovány v jiném **vnějším rozsahu platnosti**

```
function generator() {
  var count = 0;
  return function() {
    return count++;
  }
}
```

Closure

```
var idgen = generator();

console.log(idgen()); // 0
console.log(idgen()); // 1
console.log(idgen()); // 2
```

Closures jako technika pro parciální aplikaci funkcí

Parciální aplikace funkce je **funkce**, která má část parametrů již předem zadané

Closures jako technika pro parciální aplikaci funkcí

```
function multiply(a, b) {
  return a * b;
}

function multiplier(a) {
  return function(b) {
    return multiply(a, b);
  }
}

var double = multiplier(2);
var triple = multiplier(3);
```

Closures jako technika pro parciální aplikaci funkcí

```
console.log(double(10)); // 20
console.log(triple(10)); // 30

console.log(double(triple(10))); // 60
```

`this` a objekty v JavaScriptu

`this`

`this` je speciální proměnná, která je automaticky definována v každé funkci a nebo metodách objektu

V JavaScriptu má `this` **různé hodnoty** v závislosti na tom, jak je funkce nebo metoda **volána**

`this`

Volaná funkce bez kontextu má hodnotu `this` rovnou `window` v případě webového prohlížeče a `global` v případě Node.js

```
function myfn() {
  console.log(this);
}

myfn();
// web browser: window object
// Node.js: global object
```

`this`

Pokud je funkce metodou na objektu a **voláme ji pomocí (přes) tento objekt**, tak se hodnota `this` rovná danému objektu

```
var person = {
  name: "John",
  sayHello: function() {
    console.log("Hello, my name is " + this.name);
  }
}

person.sayHello(); // Hello, my name is John
```

Problematika `this`

A teď to zajímavé!

Funkce jsou referenčním typem a lze je tedy přiřadit do proměnné a zavolat funkci pomocí této proměnné

```
var sayHello = person.sayHello;
sayHello(); // Hello, my name is
```

Problematika `this`

Stejný problém ale nastává při pokusu o zavolání této funkce po nějakém čase - použití této metody jako callbacku

```
setTimeout(person.sayHello, 1000); // Hello, my name is
setTimeout(sayHello, 1000); // Hello, my name is
```

Problematika `this`

Ztratilo se jméno? Proč? Proč je nastaveno na `undefined`?

V javascriptu hodnota `this` **záleží na tom, jak je funkce volána**

Pokud ji voláme pomocí tečky na objektu `obj.method()`, tak se hodnota `this` vždy nastaví na tento objekt

Pokud ale získáme referenci na danou metodu jiným způsobem jako například přeuložením do proměnné nebo předáním jako callback, tak se hodnota `this` nastaví na ukazatel do aktuálního kontextu

Problematika `this`

```
var person = {
  name: "John",
  sayHello: function() {
    console.log("Hello, my name is " + this.name);
  }
};
var alien = { name: 42, sayHello: person.sayHello };

person.sayHello(); // Hello, my name is John
alien.sayHello(); // Hello, my name is 42
```

Tohle je vlastně funkcionalita potřebná pro polymorfismus

Problematika `this`

Jak podobné situace řešit? Potřebujeme „pořádné“ objekty

Třídy byly do JavaScriptu přidány až v ES6, co dělali programátoři předtím?

Měli speciální operátor `new` a funkci reprezentující konstruktor

Objekty a jejich konstruktory

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

var person = new Person("John");
person.sayHello(); // Hello, my name is John
```

Objekty a jejich konstruktory

Tento přístup nicméně stále trpí tím, že se mění hodnota `this`

```
var person = new Person("John");
setTimeout(person.sayHello, 1000); // Hello, my name is
```

Reference na sebe sama

Proto ve starších kódech uvidíte použití `self`, což je proměnná v rozsahu platnosti konstruktoru a je v ní uložena hodnota `this` v době vytváření objektu

Tuto proměnnou poté čtou metody (v tomto kontextu jsou to vlastně closures) a korektně s ní pracují (scope chaining)

Reference na sebe sama

```
function Person(name) {
  var self = this;
  this.name = name;
  this.sayHello = function() {
    console.log("Hello, my name is " + self.name);
  }
}

var person = new Person("John");
console.log(person); // Person {name: 'John', sayHello: ƒ}
person.sayHello(); // Hello, my name is John
setTimeout(person.sayHello, 1000); // Hello, my name is John
```

Binding

Binding (vázání) je proces, jak zaručit korektnost hodnoty `this` po „extrakci“ metody z objektu

Svázání se provádí pomocí funkce `bind` na dané funkci

Nevýhoda bindingu je, že o extrahovanou funkci se musí „starat“ (bindovat jí) ten, kdo ji extrahuje

Objekty nefungují „samostatně“

Binding

```
var person = {
  name: "John",
  sayHello: function() {
    console.log("Hello, my name is " + this.name);
  }
}

var sayHello = person.sayHello.bind(person)
sayHello(); // Hello, my name is John
setTimeout(sayHello, 1000); // Hello, my name is John
```

Prototype

Definovat funkce takto v konstruktoru je **neefektivní** hlavně z paměťového hlediska

Prototype

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

var john = new Person("John");
var jane = new Person("Jane");

console.log(john.sayHello === jane.sayHello); // false
```

Prototype

Funkce jsou referenčními typy a typicky všechny instance (objekty) stejného typu mohou mít referenci na jednu funkci v paměti

Jak toho ale dosáhnout?

JavaScript obsahuje speciální objekt `prototype`, který je **přidružen k danému typu** (**né objektu**)

```
console.log(Person.prototype); // Person {}
```

Prototype lze obohacovat o vlastní metody a tyto metody budou následně dostupné u **všech objektů daného typu**

Prototype functions

Prototype functions jsou metody, které jsou přidruženy k danému typu (**né objektu**)

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

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

Prototype functions

```
var john = new Person("John");
john.sayHello(); // Hello, my name is John

var jane = new Person("Jane");
jane.sayHello(); // Hello, my name is Jane

console.log(john.sayHello === jane.sayHello); // true
```

Prototype functions

V čem je **nevýhoda této techniky**?

Není odolná proti změnám `this`

```
var john = new Person("John");
var sayHello = john.sayHello;
sayHello(); // Hello, my name is
```

Proč?

Aby trik se `self` fungoval, každý objekt musí mít vlastní kopii metody (closure), která má přístup ke svému vlastnímu `self`.

Dědičnost

Dědičnost je proces, kdy jeden objekt (dědící objekt) získává metody a vlastnosti z druhého objektu (rodičovského objektu) a může je stylem rozšiřovat nebo modifikovat

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

Person.prototype.greeting = function() {
  return "Hello, my name is " + this.name;
}
```

Dědičnost

```
function Employee(name, salary) {
  Person.call(this, name); // call parent constructor
  this.salary = salary;
}

// Link prototype to Person's prototype (prototype chain)
Employee.prototype = Object.create(Person.prototype);

// But redefine constructor to be Employee function
Employee.prototype.constructor = Employee;
```

Dědičnost

```
Employee.prototype.greeting = function() {
  // Call parent (Person) greeting method
  var greeting = Person.prototype.greeting.call(this);

  // Append suffix to greeting
  return greeting + " and my salary is " + this.salary;
}

var john = new Employee("John", 10000);
console.log(john.greeting()); // Hello, my name is John and
my salary is 10000
```

Prototype chaining

Prototype chaining, podobně jako scope chaining, je proces, který postupně od nejbližšího typu objektu hledá požadovanou metodu nebo vlastnost

Prototype chaining

Rozšíříme prototype rodičovského typu o novou metodu a pokusíme se ji použít u objektu dědícího z tohoto typu

```
Person.prototype.sayHello = function() {
  console.log(this.greeting());
}

var john = new Employee("John", 10000);
john.sayHello(); // Hello, my name is John and my salary is
10000
```

Prototype chaining

Obsah protoype chainu lze vypsat pomocí metody `__proto__`

```
console.log(john.__proto__);
// {constructor: ƒ Employee, greeting: ƒ}

console.log(john.__proto__.__proto__);
// {constructor: ƒ Person, greeting: ƒ, sayHello: ƒ}
```

Vytvoření prototype chainu

V předchozích příkladech jsme si vytvořili prototype chain pomocí metody `Object.create`

```
Employee.prototype = Object.create(Person.prototype);
```

Jedná se o pomocnou funkci, která za vás vlastně udělá ekvivalent následujícího kódu

```
function F() {}
F.prototype = Person.prototype;
Employee.prototype = new F();
```

Asynchronní programování před ES6

Synchronní vs. asynchronní kód

Synchronní kód je takový kód, který se vykonává postupně = následující příkaz se spustí až po dokončení předchozího

```
console.log("First line");
console.log("Second line");
console.log("Third line");
```

Výstup:

```
First line
Second line
Third line
```

Synchronní vs. asynchronní kód

Asynchronní kód je takový kód, ve kterém některé operace mohou běžet **souběžně**

Jaký bude výstup následujícího kódu?

```
console.log("Start");
setTimeout(function() {
  console.log("After 2 seconds");
}, 2000);
console.log("End");
```

Synchronní vs. asynchronní kód

```
Start
End
After 2 seconds
```

Motivace pro asynchronní programování

JavaScript běží v **jednom vlákně** (single-threaded)

Bez asynchronního programování by JavaScript:

- **Blokoval UI** při dlouhých operacích jako například
  - **Čekání na odpověď ze serveru**
  - **Složité výpočty**
  - **Načítání ostatních skriptů**
  - **Animace** a **časovače**

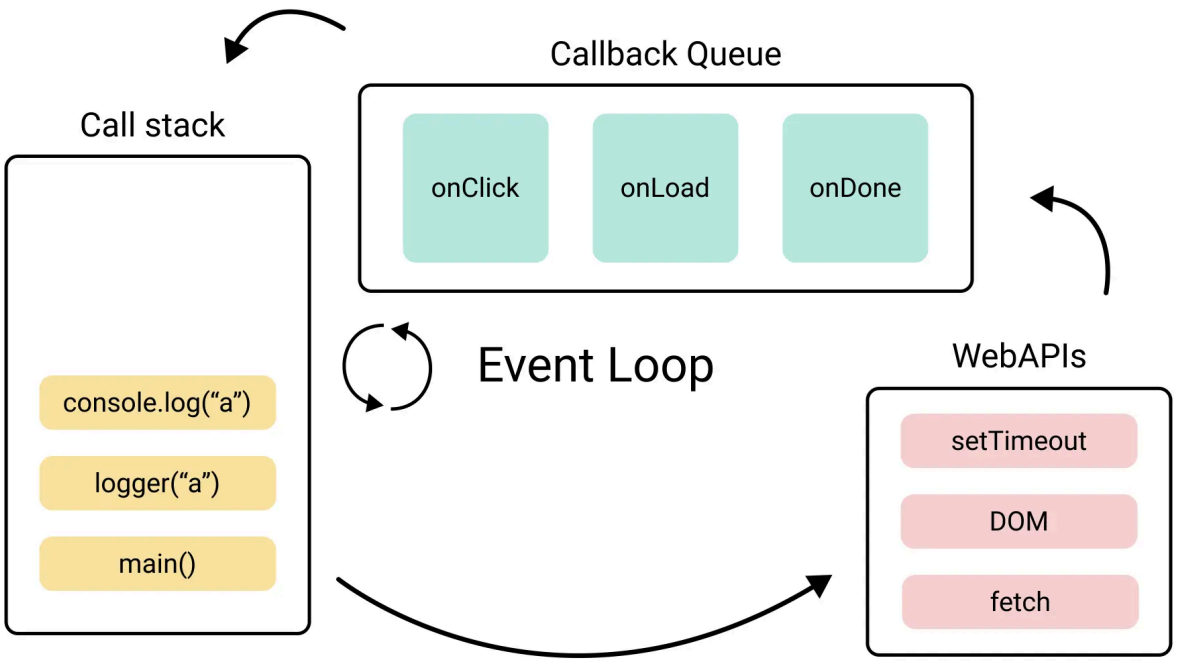
Event Loop

JavaScript používá **Event Loop** pro asynchronní operace

Základní komponenty Event Loopu:

- **Call Stack** - zásobník (LIFO) volání funkcí
- **Web APIs** - poskytují asynchronní funkce
  - `setTimeout`, `fetch`, `DOM events`
- **Callback Queue** - fronta callbacků k zavolání po dokončení asynchronní operace
- **Event Loop** - přesouvá callbacky do call stacku

Event Loop



Event Loop frontendlead.com

Callbacky podruhé

Callback je funkce předaná jako parametr jiné funkci

```
function fetchData(callback) {
  // Simulate data loading
  setTimeout(function() {
    var data = { id: 1, name: "John" };
    callback(data);
  }, 1000);
}
```

Práce s chybami v callbackách

Při načítání dat může dojít k chybě, například na serveru dojde k chybě nebo dojde k chybě při přenosu dat, timeout a podobně

Jak tento proces řešit v callbackách?

Node.js konvence je, že první parametr callbacku je určen pro chybu a druhý nebo případně další jsou určeny pro pozitivní výsledek

Práce s chybami v callbackách

```
function fetchData(callback) {
  // Simulate data loading
  setTimeout(function() {
    var data = { id: 1, name: "John" };
    callback(null, data);
  }, 1000);
}
```

Práce s chybami v callbackách

```
fetchData(function(error, data) {
  if (error) {
    console.log("Error:", error);
  } else {
    console.log("Success:", data);
  }
});
```

Callback Hell

Při více závislých asynchronních operacích vzniká **Callback Hell** nebo také **Pyramid of Doom**

Pro přehlednost na následujícím příkladu nebude řešen error handling

Callback Hell

```
authenticate(credentials, function(token) {
  fetchUser(token.userId, function(user) {
    fetchUserShoppingCart(user.id, function(cart) {
      fetchShoppingCartItems(cart.id, function(items){
        renderShoppingCart(items, function(cart) {
          openShoppingCart();
        });
      });
    });
  });
});
```

Problematika Callback Hell

Jaké vás napadají problémy Callback Hell?

- **Špatná čitelnost kódu**
- **Obtížné debugování**
- **Duplikovaný error handling**
- **Obtížné testování**

Praktické případy s Callback Hell

Asynchronní **AJAX požadavek** na server s callbackem

AJAX (Asynchronous JavaScript and XML) je technologie pro asynchronní načítání dat ze serveru

Pro tyto případy v JavaScriptu (před ES6) existuje například nativní metoda pomocí objektu `XMLHttpRequest`

Pozor, v JavaScriptu před ES6 ještě neexistovala **Promise** nebo metoda `fetch`, která Promise používá

AJAX požadavek s callbackem

```
function makeRequest(url, callback) {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if (xhr.readyState === 4) { // 4 = DONE
      if (xhr.status === 200) { // 200 = OK
        callback(null, JSON.parse(xhr.responseText));
      } else {
        callback(new Error('HTTP ' + xhr.status));
      }
    }
  };
```

AJAX požadavek s callbackem

```
  xhr.open('GET', url);
  xhr.send();
}
```

Příklad použití callbacků

```
makeRequest('/api/users/1', function(error, user) {
  if (error) {
    console.log('Error when loading user:', error);
    return;
  }
  console.log('User loaded:', user.name);

  makeRequest(
    '/api/users/' + user.id + '/posts',
```

Příklad použití callbacků

```
    function(error, posts) {
      if (error) {
        console.log('Error when loading posts:', error);
        return;
      }
      console.log('Number of posts:', posts.length);
    }
  );
});
```

Příklad použití callbacků

Práce se soubory v Node.js

```
var fs = require('fs');

fs.readFile('config.json', 'utf8', function(error, data) {
  if (error) {
    console.log('Error when reading file:', error);
    return;
  }
```

Příklad použití callbacků

```
  try {
    var config = JSON.parse(data);
    console.log('Config loaded:', config);
  } catch (parseError) {
    console.log('Error when parsing JSON:', parseError);
  }
});
```

Alternativy k callbackům

V době před ES6 existovaly **knihovny** pro usnadnění práce s asynchronním kódem:

- **Async.js** - kolekce utilit pro asynchronní kód
- **Q** - implementace Promises
- **RxJS** - reaktivní programování
- **Step**, **Flow-js** - control flow knihovny

Příklad s Async.js

```
var async = require('async');
async.waterfall([
  function(callback) {
    authenticate(callback);
  },
  function(token, callback) {
    fetchUser(token.userId, callback);
  },
  function(user, callback) {
    fetchUserShoppingCart(user.id, callback);
  },
```

Příklad s Async.js

```
  function(cart, callback) {
    fetchShoppingCartItems(cart.id, callback);
  },
  function(items, callback) {
    renderShoppingCart(items, callback);
  },
  function(cart, callback) {
    openShoppingCart(cart, callback);
  }
});
```

Příklad s RxJS

```
var Rx = require('rxjs');
var { from } = Rx;

from(authenticate()).pipe(
  switchMap(token => from(fetchUser(token.userId))),
  switchMap(user => from(fetchUserShoppingCart(user.id))),
  switchMap(cart => from(fetchShoppingCartItems(cart.id))),
  switchMap(items => from(renderShoppingCart(items))),
  tap(cart => openShoppingCart(cart))
).subscribe();
```

Shrnutí asynchronního programování před ES6

Klíčové poznatky

- JavaScript je **single-threaded** s Event Loop
- **Callbacks** jsou základem asynchronního programování
- **Callback Hell** je reálný problém při složitějších operacích
- **Error-first pattern** je konvence pro error handling
- Existují **knihovny** pro zjednodušení práce s callbacky
- ES6+ přináší **Promises** a **async/await** jako elegantnější řešení

Shrnutí asynchronního programování před ES6

Doporučení pro práci s callbacky

- Používejte **pojmenované funkce** místo anonymních
- Dodržujte **error-first callback pattern**
- **Modularizujte** kód do menších funkcí
- Zvažte použití **knihoven** pro complex flow
- Chápejte **Event Loop** a asynchronní chování JavaScriptu