JavaScript - syntaxe, datové typy, funkce, promise

JavaScipt

- JavaScript je více úrovňový, interpretovaný programovací jazyk
- Je jedním ze základních pilířů webového vývoje spolu s HTML a CSS
- Umožňuje interaktivitu na webových stránkách, jako jsou animace, dynamické změny obsahu a komunikace se servery
- **Syntaxe:**
  - **Deklarace proměnných:**
    - **var** - původní způsob, funkcionální rozsah
    - **let** - blokový rozsah, ES6
    - **const** - konstantní hodnota, blokový rozsah, ES6
  - **Datové typy:**
    - **Primitivní:**
      - **String**
      - **Number**
      - **Boolean**
      - **Null**
      - **Undefined**
      - **Symbol**
      - **BigInt**
    - **Složené**
      - **Object (včetně Array, Function)**

- **Operátory:**
  - **Aritmetické:**
    - +, -, *, /, %, **
  - **Porovnávací:**
    - ==, ===, !=, !==, >, < >=, <=
  - **Logické:**
    - &&, ||, !

Příklad operátorů

```
let x = 10;
let y = 5;
console.log(x + y); // 15
console.log(x > y); // true
console.log(x === y); // false
```

- **Podmínky a cykly:**
  - **Podmíněné příkazy:**
    - if, else if, else
    - switch
  - **Cykly:**
    - for
    - while
    - do ... while
    - for ... of, for ... in

Příklad podmínky a cyklu

```
for (let i = 0; i < 5; i++) {
  if (i % 2 === 0) {
    console.log(i + ' is even');
  } else {
    console.log(i + ' is odd');
  }
}
```

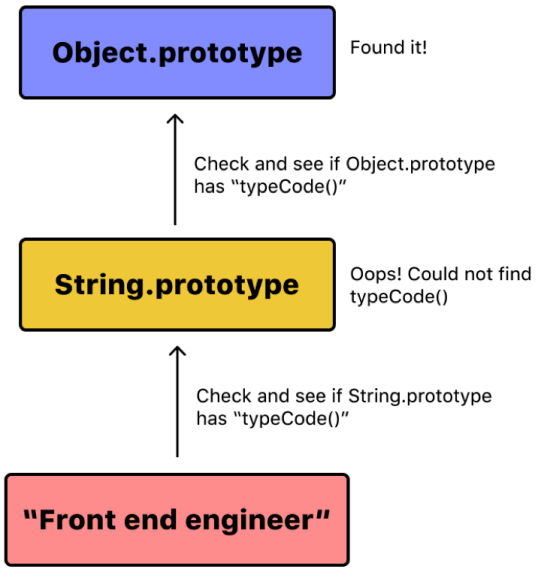
- **Funkce:**
  - **Deklarace funkce:**
    - Klasická syntaxe
    - Anonymní funkce
    - Arrow funkce (ES6)
  - **Parametry**
    - Standardní parametry
    - Výchozí hodnoty
    - Zbytek parametrů

Příklad funkcí

```
// Klasická funkce
function greet(name) {
  return 'Hello, ' + name;
}

// Arrow funkce
const greet = (name) => `Hello, ${name}`;
greet("Ječná");

// Anonymní funkce
var greet = function (name) {
    console.log(`Hello, ${name}`);
};

greet("Ječná");
```

- Prototypy a objekty:

- **Objekty =** kolekce vlastností (klíč-hodnota)
- Každý objekt má prototyp, ze kterého dědí vlastnosti a metody
- **Prototypy =** jsou mechanismus pro dědění vlastností a metod mezi objekty
- Každý objekt má interní odkaz na **prototype**, který může být využit pro sdílení funkcionality
- Prototypy umožňují vytvořit objekty, které dědí od jiných objektů, aniž by bylo potřeba tříd
- **Proč prototypy?:**
  - JS nepoužívá třídní model s dědičností jako Java, C#, C++
  - Prototypový model je jednodušší a flexibilnější pro dynamické jazyky
  - Umožňuje dynamicky přidávat nebo měnit vlastnosti a metody objektů
- **Koncept prototypů:**
  - Objekty dědí vlastnosti přes **Prototype Chain**
  - Když se přistupuje k vlastnosti objektu, JavaScript nejprve hledá vlastnost v samotném objektu
  - Pokus ji nenajde, pokračuje hledáním v prototypu objektu, a tak dále až k Object.prototype



- **Výhody a nevýhody prototypů:**
  - **Výhody:**
    - Flexibilita v dědění a sdílení vlastností
    - Úspora paměti díky sdílení metod mezi objekty
    - Dynamické přidávání a změny vlastností
  - **Nevýhody:**
    - Může být méně intuitivní pro vývojáře zvyklé na jazyky se standardními třídami
    - Složitost při ladění, pokud je řetězec prototypů hluboký
    - Méně formální struktura může vést k nekonzistencím

- Objektový literál:

Objektový literál

```
const person = {
  name: 'Alice',
  age: 30,
  greet: function: function() {
    console.log('Hello!');
  }
};
```

- Konstruktorová funkce:

Konstruktorová funkce

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype.greet = function() {
  console.log('Hello!');
};
```

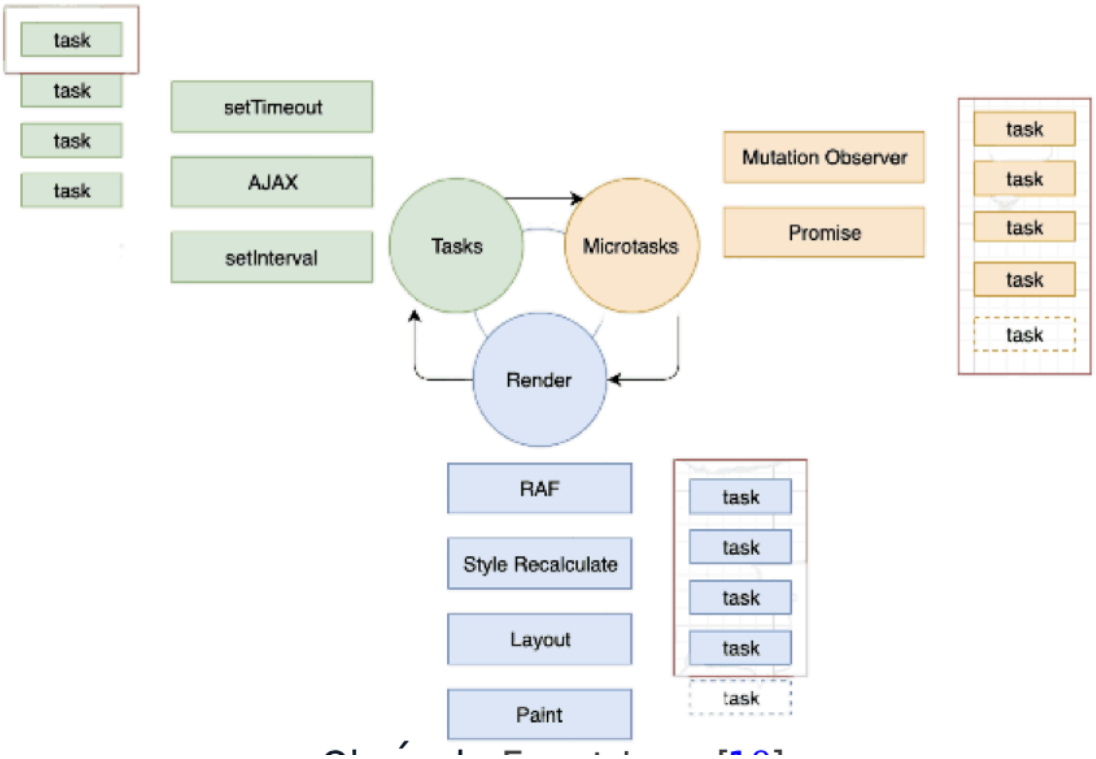
- **Třídy (ES6) a prototypy:**
  - Syntaxe tříd je syntaktický cukr nad prototypy
  - Umožňuje jednodušší zápis objektově orientovaného kódu
  - Pod pokličkou stále funguje prototypové dědění

Příklad třídy

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log('Hello!');
  }
}

const alice = new Person('Alice', 30);
```

- Architektura JavaScriptu:

- JS kód běží v prostředí tzv, **Javascript Runtime Environment** (např. V8 pro prohlížeč - chrome, Node.js pro server)
- **<u>Obecný JS engine se skládá z:</u>**
  - **Call Stack -** zásobník volání
  - **Heap -** halda pro alokaci paměti
  - **Web APIs -** poskytované prohlížečem (DOM, časovače, síťové požadavky)
  - **Event Loop -** pro zpracování asynchronních událostí
  - **Callback Queue a Microtask Queue**



- **Call stack:**
  - Je datová struktura, která sleduje volání funkcí v programu
  - Při volání funkce se přidá její vykonání přidá na vrchol zásobníku
  - Po dokončení funkce odstraní z vrcholu zásobníku
  - Pokud je zásobník prázdný, JavaScript čeká na další úlohy
- **Heap:**
  - Je oblast paměti používaná pro dynamickou alokaci objektů
  - Umožňuje programu alokovat a uvolňovat paměť během běhu
  - JavaScriptové objekty a proměnné jsou ukládány na haldě
  - **Garbage collector** automaticky spravuje uvolňování nepoužívané paměti
  - Důležitý pro výkon aplikace a prevenci úniků paměti
- **Event Loop:**
  - Koordinuje provádění kódu sběr a zpracování událostí a vykonávání dílčích úloh
  - Neustále kontroluje **Call Stack** a **Callback Queue**
  - Pokud je Call Stack prázdný, přesune první úlohu z Call Queue
- **Microtasks a Macrotasks:**
  - **Macrotasks (Makro úlohy):**
    - Hlavní úlohy, jako jsou: **události DOM**, **setTimeout**, **setInterval**
    - Zařazeny do **Callback Queue**
  - **Microtasks (Mikro úlohy):**
    - Krátké úlohy, jako jsou **Promises**, **process.nextTick()** v Node.js
    - Zařazeny do **Microtask Queue**
    - Mají vyšší prioritu než makro úlohy

- **Pořadí instrukcí a zpracování úloh:**
  - JavaScript vykonává kód synchronně po řádcích
  - Asynchronní úlohy jsou předány do Web API (v prohlížeči) nebo jiného API (v Node.js)
  - Po dokončení jsou výsledky asynchronních úloh zařazeny do příslušné fronty (Microtask nebo Callback Queue)
  - Event Loop zajišťuje, že Microtask jsou zpracovány před Macrotask

- **Jednovláknový model a asynchronost:**
  - JS je jednovláknový jazyk
  - Asynchronnost umožňuje zpracování úloh bez blokování hlavního vlákna
  - Využívá Event Loop
  - Důvodem je potřeba zachovat plynulost uživatelského rozhraní a reaktivitu aplikací
  - Jednovláknovost zjednodušuje model programování, vyhýbá se problémům se synchronizací vláken
  - Asynchronní model umožňuje efektivní využití času procesoru
  - Udržuje výkon aplikací tím, že neblokuje hlavní vlákno
  - Využívá se pro operace jako síťové požadavky, časovače, přístup k souborům apod.
  - **Promises:**
    - Jsou objekty reprezentující stav asynchronní operace: dokončení nebo selhání
    - Umožňují lepší práci s asynchronním kódem než callbacky
    - **Stav Promise:**
      - **pending** - čekající - výchozí stav, dosud nedošlo k dokončení
      - **fullfilled** - splnění - operace proběhla úspěšně, promise je splněna s výsledkem
      - **rejected** - zamítnutí - operace skočila chybou, promise je odmítnuta s důvodem (chybou)
    - Promises jsou zařazeny do **Microtask Queue**, což znamená, že mají přednost před **Callback Queue** (makro úlohami)
    - Promises nepřidávají více vláken do JavaScriptu
    - Usnadňují práci s asynchronními úlohami v jednovláknovém prostředí
    - Umožňují řetězení asynchronních operací a lepší chybovou manipulaci

Příklad použití Promises

```
function fetchData() {
  return new Promise((resolve, reject)=>{
    setTimeout(() => {
      resolve('Data loaded');
    }, 1000);
  });
}

fetchData()
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

- **Async/Await:**
  - Pro zjednodušení asynchronního kódu
  - Funkce označené jako async vracejí Promise
  - await umožňuje čekat na Promise a získat její výsledek
  - Zvyšuje čitelnost kódu a usnadňuje zpracování chyb
  - Klíčové slovo **async** označuje funkci, která vždy vrací promise
  - Příkaz **await** pozastaví vykonávání funkce, dokud není vyřešena odpovídající Promise

Základní async/await

```
async function example() {
    let result = await someAsync();
    console.log(result);
}
```

- **Proč tato implementace?:**
  - **Výkon:**
    - Umožňuje efektivní využití času procesoru bez potřeby více vláken
  - **Bezpečnost**
    - Vyhýbá se komplikacím s konkurencí a synchronizací sdílených zdrojů
  - **Jednoduchost**
    - Usnadňuje vývojářům psát asynchronní kód bez nutnosti řešit vícevláknové programování
  - **Kompatibilita**
    - Dědictví z historie JavaScriptu jako jazyka pro prohlížeče, kde jdenovláknovost byla nutností
- **Shrnutí**
  - Asynchronost je řešena pomocí Event Loop, Call Stack, Callback, Queue a Microtask Queue
  - Promises a Async/Await usnadňují práci s asynchronním kódem
  - Jednovláknový model s asynchronními mechanismy poskytuje rovnováhu mezi výkonem a jednoduchostí

Promises

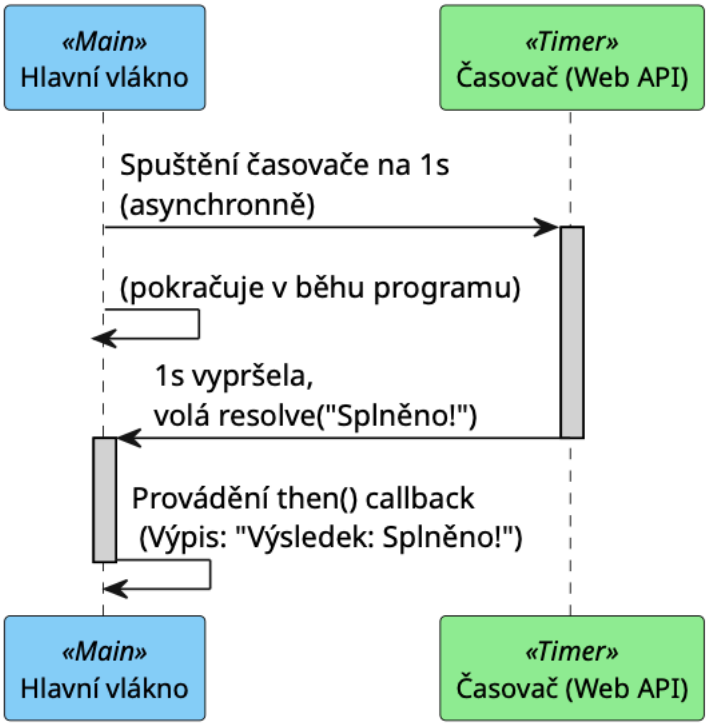
- Základní použití Promise:

- Promise se vytváří pomocí konstruktoru **new Promise(executor)**, kde **executor** je funkce s parametry **resolve** a **reject**
- Asynchronní operace v executor funkci zavolá při úspěchu **resolve(value)** nebo při chybě **reject(error)**
- Na promisu lze navázat následné zpracování pomocí metody **.then()**

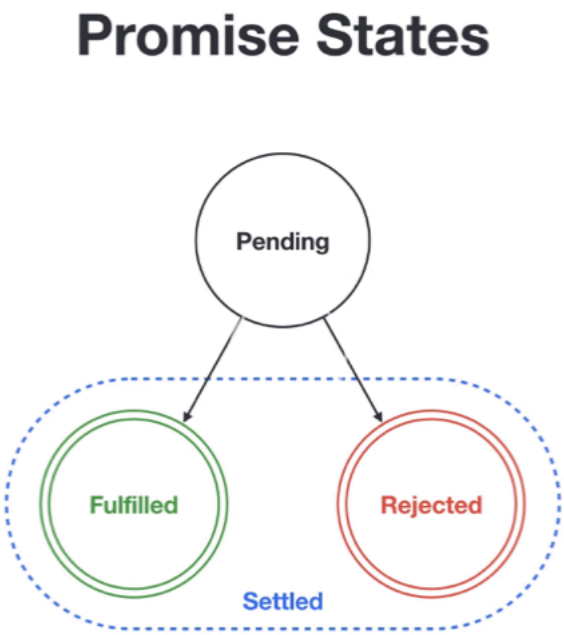
Vytvoření a použití Promise

```
// Vytvoření nové promisy,
// která se splní za 1 sekundu:
let promise = new Promise(
    (resolve, reject) => {
        setTimeout(
    () => resolve("Splněno!"), 1000);
});

// Reakce na splnění promisy:
promise.then(value => {
console.log("Výsledek:", value);
});
```

- **Tok asynchronní operace - Promise:**



- **Stavy promise:**
  - Z pending stavu přechází promise buď do stavu fullfilled, nebo rejected
  - Po přechodu do settled stavu již stav promisy nelze změnit



- **Řetězení a využití Promise:**
  - **Metody then(), catch() a finally():**
    - **then(onFullfilled, onRejected)** = nave zpracování po úspěchu či chybě. Vrací novou promisu pro další řetězení
    - **catch(onRejected)** = zkratka pro **then(undefined, onRejected)**, slouží k zachycení chyby/odmítnutí v řetězci
    - **finally(onFinally)** = zavolá se vždy po ukončení promisy (po všech **then/catch**), neovlivňuje výsledek. Slouží např. k úklidu (obdoba finally v synchronním kódu)

- **Řetězení promisů:**
  - Metoda **then()** umožňuje řetězit navazující asynchronní operace. Každé volaní then vrací novou promisu, která čeká na výsledek předchozího kroku
  - Hodnota vrácená v handleru **then** je předána do dalšího **then** v řetězci
  - Případná chyba se propaguje řetězcem, kde ji lze zachytit v **catch()** na konci

Callback hell (zanořené callbacky)

```
getData(function(a){
  step1(a, function(b){
    step2(b, function(c){
      step3(c, function(d){
        // ... hluboké zanoření ...
      }, onError);
    }, onError);
  }, onError);
}, onError);
```

Řetězení promisů

```
getData()
  .then(a => step1(a))
  .then(b => step2(b))
  .then(c => step3(c))
  .then(d => {
    // ... sekvenční zpracování ...
  })
  .catch(error => {
    // jednotná obsluha chyby
  });
```

  - **catch()** lze připojit na konec řetězce pro zachycení jakékoli chyby, která nastala v kterémkoli předchozím kroku
  - **finally()** se vykoná v každém případě (po **then** i **catch**), typicky pro kód, který se má provést vždy (ukončení načítání, uzavření spojení apod.)

Ukázka použití catch a finally

```
function fetchData() {
return Promise.reject("404 Not Found");
}

fetchData()
.then(data => {
    console.log("Success:", data);
})
.catch(err => {
    console.log("Error:", err);
})
.finally(() => {
    console.log("Dotaz dokončen.");
});
```