Motivace

Objektově orientované programování (dále jen OOP) je způsob návrhu programů - takzvaných programovacích paradigmat. Teorie těchto návrhů je více komplikovaná (a taky je to vysokoškolská tématika) ale zjednodušeně by se dalo říci, že veškeré návrhy by se dali shrnout do tří kategorií:

Datově OP

Návrh se zaobírá ukládáním dat do jednoho místa, kde jsou vždy k dispozici, data nemusí mít jasnou strukturu, a většinou se nepoužívá dědičnost. Kód však bývá velmi nepřehledný a často jsou objekty (data) omezeny svojí velikostí. Jedná se o "začátečnický" způsob programování - nepoužívají se logické celky.

Objektově OP

V tomto návrhu se používají objekty, které reprezentují instance nějakých tříd. Data jsou uložena v určitých krabičkách, které jasně definují, co se v nich nachází a kdo k nim má přístup. Je to nejvíce používaný styl návrhu ale je taktéž nejsložitější na pochopení. Používá se spousta věcí (třídy, instance, přístupnost, statičnost, dědičnost) a v nějakých jazycích (či nástavbách) lze jít ještě více do hloubky. Obecně je se jedná o nejlepší návrh, který aktuálně známe.

Funkcionálně OP

Celý program je vlastně jedna velká vnořená funkce, která přijímá parametry a vrací výstup. Lze si to představit jako sérii krabiček, které vždy vezmou výstup jiné krabičky, data zpracují, a výstup předají další. Tento výstup se potom vypíše. Funkcionální programování v pravém slova smyslu není možno dělat ve všech jazycích; existují pro to speciální jazyky jako Haskell a další.

Tyto paradigmata nemusí fungovat ve všech jazycích a každé se hodí na něco jiného. Obecně je však velmi dobré znát OOP, protože se používá všude a na pochopení FOP je velmi potřeba.

Objekty

V JavaScriptu (dále jen JS) známe objekty jako nějaký způsob dat. Objekty umí uchovávat proměnné (nebo vlastnosti), které vždy mají jméno a hodnotu. Hodnota může být prakticky cokoliv. Lze ukládat například data (datové typy JS externě nemá, proto např. číslo, text, desetinné číslo, ...) a nebo například funkce.

Následujícím kódem definujeme objekt:

```
let human = {
    name: "Matěj Cajthaml",
    favouriteAnimal: "cat",
```

```
    pets: ["Yoko", "Bambi"],

    purchasePet: function(name) {
        pets.push(name);
    }
};
```

K proměnným z tohoto objektu můžeme přistupovat následovně:

```
console.log(human.name);
console.log(human.pets[0]);
console.log(human["pets"]);
human.purchasePet("Yenda");

console.log(human["pets"]);
console.log(human.name);
```

Uvnitř objektů můžeme uchovávat i další objekty.

Třídy

Třídu v JS lze pochopit jako určitou šablonu, které určujeme proměnné a funkce, které bude vždy obsahovat. Těmto proměnným říkáme vlastnosti a funkcím metody. Při vytvoření instancí je garantováno, že tyto vlastnosti a metody bude daná instance třídy obsahovat a to i kdyby byly nastavené na hodnotu `undefined`.

Třídy nám tedy slouží ke konzistenci dat, k tomu, abychom vždy věděli, co se v daném objektu nachází. Z tohoto vyplívá že instance jsou takové objekty, které jsou vytvořeny podle třídy - šablony.

Třídy definujeme pomocí klíčového slova `class`. Třídy musí mít vždy unikátní jméno. Třídy definujeme na začátku souborů.

```
class Cat {
    name; // vlastnosti
    color;
    age;

    meow() { // metody
        console.log(`${this.name} meows`);
    }
}
```

Neoficiální konvence je taková, že jména tříd píšeme na začátku velkým písmenem a dále pomocí tzv. PascalCase - například: `BrownCatPetter`. Vlastnosti vždy definujeme na začátku třídy. Metody následují po vlastnostech. Pro vlastnosti a metody používáme tzv. camelCase - například: `nameOfTheCat`.

Vlastnostem můžeme nastavovat základní hodnotu, která bude nastavena při vytvoření instance.

```
class Cat {
    name = "Yoko";
    color;
    age;
}
```

Pro každou třídu můžeme taktéž definovat konstruktor. Konstruktor (constructor) je speciální metoda, která musí být pojmenována pomocí klíčového slova `constructor`. Tato metoda přijímá různé parametry. Konstruktory se automaticky volají právě tehdy, když se vytváří instance. Při vytváření můžeme určit dané parametry, které se do konstruktoru předají.

```
class School {
    name;

    constructor(name) {
        console.log(`Probíhá registrace školy ${name}.`);
        this.name = name;
    }
}

const sspsag = new School("Smíchovská střední průmyslová škola a gymnázium"); // jméno se p
```

Konstruktory používáme pro konzistenci dat, abychom si byli jistí, že to, co se dostane dovnitř bude validní ihned po tvorbě instance.

Instance

Instance je vždy určena nějaké třídě. Z ní získává povinné vlastnosti a metody, které má k dispozici. Tyto metody můžeme na instanci volat a oni mohou upravovat její data. Vlastnosti můžeme číst a upravovat.

Instance vytváříme pomocí klíčového slova `new` po kterém následuje jméno třídy.

```
class Cat {
   name;
```

```
    color;
    age;

    meow() {
        console.log(`${this.name} meows`);
    }
}

const yoko = new Cat();

console.log(yoko); // vypíše se Cat {name: undefined, color: undefined, age: undefined}

yoko.name = "Yoko";
yoko.color = "zrzavá";
yoko.age = 2;

console.log(yoko); // vypíše se Cat {name: 'Yoko', color: 'zrzavá', age: 2}
```

Ve výše uvedeném kódu říkáme že data uložená v konstantě yoko je instance třídy Cat.

Přístupnost uvnitř třídy

Aktuální vlastnosti, které jsme si ukazovali, jsou veřejné, to znamená, že k nim můžeme přistupovat odkudkoliv, kde máme k dispozici danou instanci. Tyto hodnoty můžeme číst i upravovat.

Často se nám toto chování nehodí a potřebujeme, aby byli vlastnosti neveřejné a mohla k nim přistupovat instance pouze uvnitř sebe (tedy z metod). Metody můžeme taktéž označit jako neveřejné a budou je moci volat jen jiné metody na instanci.

V JS to uděláme tak, že před danou vlastnost napíšeme hashtag - mřížku - `#`.

```
class Notebook {
    name; // je k dispozici všude
    #macId; // jen uvnitř třídy

    getMacId() {
        return `${this.#macId} (${this.#accessDrive()})`;
    }

    #accessDrive() {
       return "/home/private/cats";
    }
```

```
}


const notebook = new Notebook();
console.log(notebook.name);
console.log(notebook.macId);
console.log(notebook.getMacId());
console.log(notebook.#macId); // chyba!
```

Statické věci uvnitř třídy

Do teď byly vlastnosti a metody vázané na dané instance. Nyní si představíme statické vlastnosti a statické metody. Static (v řeči IT) znamená něco neměnného, stálého, nehybného, nezávislého.

Statické vlastnosti a metody tedy nejsou závislé na instance ale třídy. Tudíž nemají přístup k datům instance a ani k jiným nestatickým věcem. Statické vlastnosti často obsahují nějaké údaje, které nechceme mít někde uložené v kódu, ale chceme je mít přehledně umístněné u věcí, se kterými souvisí. Statické metody slouží k složitějším operacím, například vytvoření instancí, připojení k databázi a dalším věcem, které s instanci jednoduše nesouvisí.

Tyto věci v kódu definujeme pomocí klíčového slova `static`. Statické věci se volají na třídě tj. na jménu třídy.

```
class Cat {
    name;
    static name = "Kočka";

    constructor(name) {
        this.name = name;
    }

    static getKnownCats() {
        return [
            new Cat("Yoko"),
            new Cat("Bambi"),
        ];
    }
}

const yenda = new Cat("Yenda");
console.log(yenda.name);

for(let cat of Cat.getKnownCats()) {
```

```
    console.log(`${Cat.name}: ${cat.name}`);
}


// VÝPIS:
// Yenda
// Kočka: Yoko
// Kočka: Bambi
```

Gettery a settery

Často se nám při tvoření třídy hodí si monitorovat kdo, kdy a co nastavuje do různých vlastností. Když chceme zamezit k přístupu, dáme vlastnost neveřejnou pomocí #. Pro čtení a nastavení bychom však museli speciální metody a to se brzo převrhne v neukončitelný boj s přehledností. Proto existují setter a gettery, které přesně vylepšují práci se čtením a nastavováním.

Getter je vlastně taková virtuální vlastnost, která má určenou metodu, která se zavolá právě tehdy, když k dané vlastnosti přistupujeme. Gettery nemají žádné argumenty. Vytvořený getter pro vlastnost lze pouze získávat, nelze jeho hodnotu nastavovat (nic to neudělá).

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    get area() {
        return this.height * this.width;
    }
}

const square = new Rectangle(10, 10);
console.log(square.area);
```

Setter slouží právě k nastavení hodnot. Když nastavujeme vlastnost, která má setter tak se použije definovaná metoda k tomu, aby se data zpracovali a případně se nastavili / zpracovali dále. Setter přijímá jeden argument, což je hodnota, která se nastavuje.

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    set area(value) {
```

```
        this.height = Math.sqrt(value);
        this.width = Math.sqrt(value);
    }
}

const square = new Rectangle(10, 10);
console.log(square.area);
square.area = 25;
console.log(square.width);
console.log(square.height);

// square.area(); nejde
```

Gettery a settery se taktéž hodí například k nějakým logům - monitoringu dané entity.

Dědičnost

Dědičnost je velmi silná a důležitá věc - dovoluje nám sdílet vlastnosti a metody z jiných tříd.

V praxi to hlavně znamená to, že když třída dědí z jiné třídy, máme vždy k dispozici její vlastnosti a metody. A jestli třída, ze které dědíme, dědí z jiné, tak máme k dispozici i její data a tak dále.

Když poté vytvoříme instanci dané třídy, tak je jasné, že bude mít k dispozici vlastnosti ze třídy, které dědíme. Můžeme tedy počítat s tím, že se jedná jen o tuto instanci (data navíce nás nemusí zajímat) a to vede k **abstrakci**.

Pomocí klíčového slova `extends` určíme, z jaké třídy třída dědí. Třída může dědit jen z jedné třídy.

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    speak() {
        console.log(this.name + " makes a noise.");
    }
}

class Cat extends Animal {
    meow() {
        console.log(this.name + " says meow.");
        this.speak();
    }
```

```
}


const cat = new Cat("Yoko");
cat.meow();


// VÝPIS:
// Yoko says meow.
// Yoko makes a noise.
```

Obecně se dědičnost používá k omezení opakování částí kódu. Nemusíme totiž několikrát určovat ty samé vlastnosti a metody, když jednoduše z dané třídy můžeme pouze dědit. Abstrakce je druhý nejdůležitější přístup a ten si teď vysvětlíme.

Abstrakce je způsob přístupu k datům - objektům, které máme uložené na jednom místě. Často totiž potřebujeme například uchovávat všechny zvířata (viz kód výše) v jednom poli. Když ale smícháme různé instance různých tříd, nemůžeme s ničím počítat. Když používáme dědičnost tak si můžeme říct, že v daném poli budou všechny instance tříd, které někdy dědí z třídy Animal a tudíž budou mít k dispozici vlastnost `name` a metodu `speak`.

Často ale na jednotlivých zvířatech chceme získat speciální hodnoty a popř. i volat jejich metody. K tomu, abychom zjistili zda nějaký objekt dědí (je instancí) nějaké třídy, musíme použít operátor `instanceof`.

```
let animals = [ new Cat("Yoko"), new Cat("Bambi"), new Animal("Peopa")];


for (let animal of animals) {
    console.log(animal.name); // protože víme, že všechny data z pole animals dědí z třídy 


    if(animal instanceof Cat) {
        // je-li zvíře kočka, zamňoukej.
        animal.meow();
    }
}
```

Všechny objekty a instance dědí ze třídy `Object`.

Při práci s děděním se nám může stát, že budeme postupně pomocí dědičnosti chtít přidávat věci do konstruktoru - k tomu třídy slouží, k jakési validitě dat. Při definování konstruktoru v nějaké třídě ve které dědíme z jiné, která má taktéž konstruktor, musíme zavolat tzv. `super` metodu. Super metoda definuje rodiče - děděnou třídu. Kdybychom jí s parametry nezavolali, může se stát nekonzistence dat a některé instance by se mohli chovat jinak, než jsme je definovali.

```
class Form {
    name;
    signature;

    constructor(name, signature) {
        this.name = name;
        this.signature = signature;
    }
}

class PatientForm extends Form {
    patientName;
    patientBirthDate;
    patientAddress;
    patientPhone;
    patientInsurance;

    constructor(signature, patientName, patientBirthDate, patientAddress, patientPhone, pati
        super("General patient form", signature);
        this.patientName = patientName;
        this.patientBirthDate = patientBirthDate;
        this.patientAddress = patientAddress;
        this.patientPhone = patientPhone;
        this.patientInsurance = patientInsurance;
    }
}

const patient = new PatientForm('Džonec', 'John', '01/01/2000', '123 Main St', '555-555-555
```