Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Jazyk Dart

Bakalářská práce

Autor: Roman Navrátil
Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Pavel Janečka

Hradec Králové Leden 2016

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 29.4.2016 Roman Navrátil

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Pavlu Janečkovi za metodické vedení práce a odbornou pomoc při zpracování bakalářské práce.

Anotace

Bakalářská práce se zaměřuje na platformou Dart určenou pro vývoj moderních webových aplikací. Společnost Google vyvinula tuto platformu, aby ukázala, že vývoj moderních webových aplikací může být i jednodušší než při použití dostupných technologií v době vydání první stabilní verze platformy. Jazyk JavaScript, který je nativním jazykem webových prohlížečů, skrývá množství nástrah projevujících se hlavně u středně velkých a velkých projektů. Existuje proto celá řada jiných jazyků, které se snaží zpříjemnit vývoj, ale kompilují se do JavaScriptu. Jedním z nich je Dart. Bakalářská práce jej srovnává s dalšími dvěma rozšířenými jazyky a to CoffeeScript a TypeScript. Kapitoly práce představují jednotlivé jazyky a dále porovnávají k nim dodávané vývojové nástroje. Popisují také, jak jednotlivé jazyky odstiňují vývojáře od úskalí JavaScriptu.

Annotation

Title: Programming language Dart

This bachelor thesis focuses on Dart platform which is designated for developing modern web applications. Google company has developed this platform to show that development of modern web applications can be simpler than it was in the first version of Dart platform. JavaScript programming language, which is native programming language of web browsers, hides many pitfalls which appear mainly on medium sized or large projects. Therefore, there are many other programming languages, which try to make more enjoyable web development, but compiles to JavaScript. One of the languages is Dart. The bachelor thesis compares Dart to other two popular languages CoffeeScript and TypeScript. Following chapters familiarise the reader with these languages and compare development tools provided with these languages. It also describes how individual languages help to avoid some pitfalls of JavaScript.

Obsah

1 Úvod 1

2 Dart 2

2.1 Historie 2

2.2 Nástroje 4

2.2.1 Vývojové prostředí 4

2.2.2 Dartium 5

2.2.3 Software Development Kit (SDK) 5

2.2.3.1 Dart VM 5

2.2.3.2 Dartanalyzer 7

2.2.3.3 Pub 7

2.2.3.4 Dart2js 7

2.2.3.5 Dartdoc 8

2.2.3.6 Dartfmt 8

2.3 Syntaxe jazyka Dart 8

2.4 Rozsah platnosti proměnných 10

2.5 Klíčové slovo this 11

2.6 Třídy a dědičnost 12

2.7 Statické atributy a metody 14

2.8 Rozhraní 15

2.9 Abstraktní třída 15

2.10 Výjimky 16

2.11 Mixin 17

2.12 Generické typy 18

2.13 Asynchronní zpracování 19

2.13.1 Future 19
2.13.2 Stream 21
2.13.3 Generátory 22
2.14 Metadata 23
2.15 Reflexe 23
3 Alternativy k jazyku Dart 26
3.1 JavaScript 26
3.1.1 Nástroje 27
3.1.2 Základní syntaxe 28
3.1.3 Rozsah platnosti proměnných 30
3.1.4 Klíčové slovo *this* 31
3.1.5 Třídy a dědičnost 32
3.1.6 Statické atributy a statické metody 34
3.1.7 Rozhraní 34
3.1.8 Abstraktní objekt 35
3.1.9 Výjimky 36
3.1.10 Mixin 36
3.1.11 Generické typy 36
3.1.12 Asynchronní zpracování 36
3.1.13 Metadata 37
3.1.14 Reflexe 37
3.2 TypeScript 38
3.2.1 Nástroje 39
3.2.2 Základní syntaxe 40
3.2.3 Rozsah platnosti proměnných 45

3.2.4 Klíčové slovo *this* 45
3.2.5 Třídy a dědičnost 46
3.2.6 Statické atributy a metody 48
3.2.7 Rozhraní 48
3.2.8 Výjimky 49
3.2.9 Abstraktní třída 49
3.2.10 Mixin 50
3.2.11 Generické typy 51
3.2.12 Asynchronní zpracování 52
3.2.13 Metadata 52
3.2.14 Reflexe 52
3.3 CoffeeScript 53
3.3.1 Nástroje 54
3.3.2 Základní syntaxe 55
3.3.3 Rozsah platnosti proměnných 57
3.3.4 Klíčové slovo *this* 57
3.3.5 Třídy a dědičnost 57
3.3.6 Statické atributy a metody 58
3.3.7 Rozhraní 59
3.3.8 Výjimky 59
3.3.9 Abstraktní třída 59
3.3.10 Mixin 60
3.3.11 Generické typy 60
3.3.12 Asynchronní zpracování 60
3.3.13 Metadata 60

3.3.14 Reflexe 61
4 Shrnutí výsledků 62
5 Závěr a doporučení 64
6 Seznam použité literatury 66
7 Seznam obrázků 68
8 Seznam tabulek 68

1 Úvod

Vznik jazyka JavaScript dovolil přidat webovým stránkám dynamičnost, umožnil tak vznik webových aplikací. Nejdříve se psaly jen krátké skripty, které doplňovaly základní funkčnost aplikace. Postupně se přešlo k až psaní robustních webových aplikací, které ale v JavaScriptu není snadné udržovat. Z toho důvodu se postupně objevovaly i další jazyky, se kterými by přišla lepší udržitelnost. Tvůrci těchto jazyků ale nemohli očekávat, že každý výrobce prohlížeče bude přidávat podporu právě pro jejich jazyk, proto byly vytvořeny nástroje, které kompilovaly aplikační kód do JavaScriptu, aby si zajistily funkčnost ve webovém prohlížeči.

Společnost Google, která se mimo jiné zabývá i vývojem webových aplikací, se rozhodla vytvořit programovací jazyk a sadu podpůrných nástrojů pro tvorbu webových aplikací primárně pro svoje potřeby a tak představila Dart. Dart není pouze nový programovací jazyk, ale také vývojové prostředí Dart Editor a Dartium, což je prohlížeč, který má implementované běhové prostředí pro Dart kód a SDK (Software Developers Kit), který obsahuje základní Dart knihovny, Dart VM[1] a další nástroje. Tyto nástroje mají za cíl co nejvíce usnadnit a zpříjemnit vývoj robustních webových aplikací.

Jazyků se stejným cílem existuje celá řada a každý přišel se svojí syntaxí a řešením typických problémů. Pro potřeby této práce byli vybrány další zástupci – CoffeeScript a TypeScript. S nimi práce porovnává, zda-li Dart nabízí něco navíc.

Práce je rozdělena na dvě části. V první části je představena historie platformy Dart a nástroje patřící do této platformy. Dále je popsána základní syntaxe jazyka Dart, po které následuje popis dalších vlastností jazyka Dart jako jsou třídy nebo rozhraní. Ve druhé části práce jsou krátce představeny alternativy JavaScript, TypeScript a CoffeeScript. Ke každé alternativě jsou uvedeny základní informace, které jsou následovány popisem syntaxe těchto jazyků a jejich dalších vlastností.

[1] Běhové prostředí

2 Dart

Platforma Dart byla představena společností Google v roce 2011. Důvodem, proč společnost Google začala vyvíjet tuto platformu, je snaha o usnadnění a zpříjemnění vývoje komplexních webových aplikací. Do této platformy patří programovací jazyk Dart, Dart VM (Virtual Machine), SDK (Software Development Kit) obsahující základní Dart knihovny, kompilátor z Dartu do JavaScriptu, umožňující spustit aplikaci v prohlížeči bez implementovaneho Dart VM, prohlížeč Dartium s implementovaným Dart VM, pro vývoj bez nutnosti kompilace do JavaScriptu a další nástroje, které budou popsány v této práci.

Nejčastěji je Dart porovnáván s jazykem JavaScript, který stojí za rozvojem webových aplikací. Na malé aplikace a jednoduché skripty se JavaScript používá dobře a není problém kód dlouhodoběji udržovat. Problém nastává při vývoji středních a velkých webových aplikací, na kterých pracuje větší skupina vývojářů. Debuggování a úprava takové aplikace mohou být velmi obtížné. To se Dart snaží změnit. [1, 1]

Tato kapitola se detailně věnuje platformě Dart. Nejprve je nastíněna historie a uvedeni zakladatelé jazyka Dart. Následuje představení základních informací o programovacím jazyce, jeho syntaxi a několik základních nástrojů pro podporu vývoje.

2.1 Historie

Zakladateli Dartu jsou pánové Lars Bak a Kasper Lund. Lars Bak je dánský počítačový inženýr, který je expertem na virtuální běhové prostředí pro programovací jazyky. Sedm let pracoval pro společnost Sun, kde vyvíjel programovací prostředí pro jazyk Self. V roce 1994 začal pracovat pro společnost Longiew Technologie LLC, ve které navrhl a implementoval výkonný virtuální stroj pro jazyky Smalltalk a Java. Po tom, co opustil Sun, založil se studenty z Aarhus University společnost OOVM, kterou pak koupila švýcarská společnost Esmertec, v níž pan Bak strávil dva roky. Nakonec byl kontaktován společností Google s

nabídkou práce na prohlížeči Chrome. V Google vytvořil virtuální stroj pro běh JavaScriptu nazvaný V8[2], který byl přidán do prohlížeče Chrome. [3]

Kasper Lund pochází taktéž z Dánska. Stejně jako Bak pracoval ve společnosti Sun, kde vyvíjel implementaci CLDC (The Connected Limited Device Configuration) Hotspot[3]. Byl spoluzakladatelem společnosti OOVM. Po odkoupení společnosti OOVM společností Esmertec, pracoval dva roky v Esmertec na platformě OSVM[4]. Od roku 2006 do roku 2010 pracoval v Google společně s Bakem na V8. [4]

Nyní pánové Bak a Lund společně pracují na projektu Dart. První veřejnou verzi programovacího jazyka Dart představil Lars Bak 10. října 2011 na oficiálním blogu společnosti Google. O dva roky později 14. listopadu 2013 byla představena první stabilní verze Dart SDK 1.0. [2] Motivací společnosti Google pro vývoj virtuálního stroje V8, bylo zaměření společnosti na webové aplikace s takovým rozsahem, který nebyly schopné existující webové prohlížeče zobrazit a zpracovat. Zároveň šlo o motivaci konkurenčních prohlížečů k dalšímu vývoji a celkovému rozvoji webové platformy. I proto je V8 open source projekt a každý jej může využít.

V případě projektu Dart je důvod stejný. Google se domnívá, že vývoj webových aplikací by mohl být jednoduší a webové aplikace rychlejší. Proto vytvořil Dart, který má pomoci hlavně ve dvou oblastech. Má nabídnout vyšší výkon a vyšší produktivitu. [5] Jedním z cílů Dartu je snadnější znuvupoužitelnost, jednodušší odhalování syntaktických i logických chyb a možnost refaktoringu[5], která v JavaScriptu prakticky neexistuje.

U vývojářů měl Dart po představení úspěch. Kód je lépe čitelný a dokáže běžet ve svém virtuální stroji Dart VM mnohem rychleji než stejný kód v JavaScriptu ve V8. Po prvotním nadšení programátorů se očekávalo, že bude Dart VM přidán do stabilní verze prohlížeče Chrome. Čekalo se dva roky od uvedení verze 1.0 než zakladatelé Dartu vydali správu, že nehodlají Dart VM implementovat do Chrome a raději se

---

[2] V8 je JavaScriptový engine vyvíjený společností Google. V8 kompiluje a zpracovává JavaScriptový kód, alokuje paměť pro objekty a používá garbage collector pro uvolňování paměti. [24]
[3] CLDC je rozhraní a virtuální stroj pro zařízení s omezenými zdroji jako například telefony, pagery nebo PDA (Personal Digital Assistant). [25]
[4] OSVM je platforma pro vestavěné systémy. [4]
[5] Rektoring je systematický proces provádění změn v kódu bez ovlivnění funkčnosti.

budou věnovat zlepšení kompilace do JavaScriptu. "*Chceme nejen pro Google Chrome ale i celý web to nejlepší a proto se budeme soustředit na kompilaci Dartu do JavaScriptu. Rozhodli jsme se neintegrovat Dart VM do Chrome. Naše nová strategie nás staví na cestu, na které poskytujeme funkce, které naši uživatelé potřebují, aby mohli být produktivnější při tvorbě webových aplikací pomocí Dartu*." [26]

2.2 Nástroje

Kapitola představuje vývojové nástroje platformy Dart a samotné běhové prostředí, tzv. Virtual Machine (VM).

2.2.1 Vývojové prostředí

Do verze jazyka Dart 1.11 bylo součástí nástrojů i vývojové prostředí (tzv. IDE) založené na Eclipse speciálně upravené pro vývoj Dart aplikací. Sám tým jazyka Dart však nyní doporučuje používat alternativy v podobě WebStorm nebo IntelliJ od firmy JetBrains, které mají integrovaný plugin pro jazyk Dart. Další možností je použití textového editoru, který umožňuje instalovat doplňky, jako například Sublime Text nebo Atom. Tyto doplňky jsou uvolněny pod svobodnou licencí a nejsou přímo podporovány vývojáři jazyka Dart. Pro seznámení s programovacím jazykem Dart není nutné instalovat vývojové prostředí nebo textový editor. Lze testovat jednoduchý kód přímo ve webovém prohlížeči pomocí online editoru na adrese https://dartpad.dartlang.org/. [7]

Obrázek 1 Ukázka online editoru DartPad. Zdroj: vlastní zpracování

2.2.2 Dartium

Jedná se o speciální sestavení prohlížeče Chromium, které v sobě obsahuje i Dart VM. Účel tohoto prohlížeče je spouštění Dart kódu bez nutnosti kompilace do JavaScriptu. Z důvodu bezpečnosti a stability není doporučeno tento prohlížeč používat na běžné prohlížení internetu. [7]

2.2.3 Software Development Kit (SDK)

Software Development Kit je sada nástrojů pro vývoj obsahující základní knihovny jazyka a doplňkové utility, které jsou spustitelné z příkazového řádku. Součástí SDK je virtuální stroj (VM) jazyka Dart a další nástroje, které budou postupně představeny níže.

2.2.3.1 Dart VM

Kromě webových aplikací lze v jazyce Dart vytvářet i skripty běžící z příkazové řádky a nezávislé na grafickém uživatelském rozhraní. Dart VM je možné nainstalovat na operační systémy Linux, Windows i OS X. [23]

Virtuální stroj Dartu podporuje dva druhy módů, ve kterých je schopný interpretovat:

- checked
- production

Jako výchozí je nastaven mód production. Při tomto nastavení se ignorují datové typy. To znamená, že s proměnnou, která je typována jako String, bude zacházeno, jakoby neměla žádný datový typ uvedený. Tento krok se provádí zejména kvůli rychlosti a dovoluje optimalizovat kompilovaný výstup v jazyce JavaScript. Dart je dynamicky typový jazyk a tím pádem ke svému fungování nepotřebuje znát přesné datové typy.

Během vývoje aplikace je ale vhodné, aby Dart VM kontroloval datové typy. K tomu slouží mode checked. Pokud se v tomto módu programátor pokusí uložit řetězec znaků do proměnné, která je typována jako integer, dojde k syntaktické chybě. Díky módu checked je tedy možné při vývoji předejít chybám v kódu spojených s chybnými datovými typy.

Další možností Dart VM je tvorba tzv. snímků. "Snímek je sekvence bytů, která reprezentuje serializovanou formu jednoho nebo více objektů Dartu"[6] [12]. Dart VM používá tyto snímky pro zrychlování prvního startu aplikace a pro posílání objektů mezi isolates. Isolates umožňují paralelní zpracování kódu. Fungují podobně jako vlákna, ale isolates spolu nesdílí paměť. Komunikace mezi isolates probíhá ve formě zpráv.

Důvod, proč je start aplikace se snímky rychlejší, je ten, že každý snímek obsahuje předpřipravená data pro knihovny jádra a aplikační skripty a tím pádem pak není třeba je při dalším startu znovu předpřipravovat.

Dart VM umí tvořit tři druhy snímků.

- Úplný snímek
- Snímek skriptu
- Snímek objektu

---

[6] A snapshot is a sequence of bytes that represents a serialized form of one or more Dart objects.

Úplný snímek je kompletní obraz celé haldy isolate. Používá se pro rychlý start Dart VM a pro inicializaci základních knihoven. Snímek skriptu je obraz aplikačního skriptu, který se nachází na haldě v isolate, po té, co se skript nahraje do isolate, ale před tím než se skript začne vykonávat. Snímek objektu se používá vždy, když potřebuje jeden isolate poslat zprávu jinému isolate. V takové chvíli Dart VM vytvoří snímek objektu. [1, 7]

2.2.3.2 Dartanalyzer

Dartanalyzer se používá pro nalezení chyb a varování v kódu v příkazovém řádku, podle oficiální specifikace jazyka Dart. [7]

2.2.3.3 Pub

Společnost Google vytvořila pro Dart i vlastní balíčkovací systém, který umožňuje přidávat do aplikace balíčky třetích stran. Všechny balíčky se spravují v souboru pubspec.yaml v kořenovém adresáři projektu. Po přidání balíčku do tohoto souboru a následném uložení se automaticky spustí stahování požadovaných balíčků. Vývojář pak ve skriptu, ve kterém chce použít externí knihovnu na začátku importuje tuto knihovnu.

Na internetové stránce https://pub.dartlang.org/ je pak katalog všech dostupných balíčků, které je možné použít, a jejich kompletní dokumentace. [7]

2.2.3.4 Dart2js

Aby aplikace naprogramované v jazyce Dart mohly fungovat i v prohlížečích, které nemají implementován Dart VM, umožňuje utilita Dart2js kompilovat kód do jazyka JavaScript, který podporují všechny moderní prohlížeče. Tento kompilátor také umí informovat o nepoužívaném kódu, nebo napovědět, jak vylepšit vývojářův kód. Na stránkách dokumentace se nacházejí dva tipy, které mají zlepšit výsledek kompilace do javascriptu.

- Napsat kód tak, aby odvození datových typů bylo jednodušší
- Použít parametr `–minify`, který snižuje množství zkompilovaného kódu za cenu zhoršení jeho čistelnosti

Vývojář se přitom nemusí starat o přidané knihovny. Dart2js provádí optimalizaci, při které vynechává nepoužívané třídy, metody apod. [7]

2.2.3.5 Dartdoc

Od verze 1.12 je součástí Dart SDK i nástroj Dartdoc, který slouží pro generování dokumentace v HTML[7]. Dokumentace se generuje s dokumentačních komentářů do složky kořenový_adresář_projektu/doc/api. [7]

2.2.3.6 Dartfmt

Nástroj slouží pro formátování kódu podle konvencí jazyka Dart. Dartfmt formátuje kód na dvou úrovních

- základní formátování
- transformace kódu

Základní formátování odstraňuje přebytečné odsazení a bílé znaky. Přidává také zalomení řádků na vhodných místech.

Formátování pomocí transformace kódu se provádí za účelem čistění zdrojového kódu. Zabaluje bloky do složených závorek, zkracuje zápis prázdných konstruktorů nebo, kde je to možné, odstraňuje středníky. [7]

2.3 Syntaxe jazyka Dart

Dart není striktně typovaný jazyk, pokud však vývojář na každém místě udává datové typy proměnných, kód velmi připomíná programovací jazyk Java. Následující kód je ukázkou deklarace proměnných s uvedením datového typu i bez jeho uvedení. [6]

```
1  // deklarace proměnné bez udání typu
2  var firstName = 'Roman';
3  // deklarace proměnných s určením datového typu
4  String lastName = 'Navratil';
5  int age = 24;
6  bool student = true;
7  double height = 189.5;
```

[7] Obdobně jako nástroj javadoc pro kód v jazyce Java

Syntaxe podmínek bude většině vývojářů známá.

```
1  int number = 6;
2  if (number % 2 == 0) {
3    print('Sudé číslo');
4  } else {
5    print('Liché číslo');
6  }
7  // Jednořádkový zápis pomocí ternárního operátoru
8  number % 2 == 0 ? print('Sudé číslo') : print('Liché číslo');
```

I u funkcí je možné psát datové typy návratových hodnot a parametrů funkce, ale stejně jako u proměnných to jazyk nevyžaduje. Syntaxe zápisu funkcí je stejná jako v jazyce Java, jen v jednom případě má Dart speciální – zkrácený – zápis funkce, a to když tělem funkce je pouze jeden výraz.

Parametry funkcí mohou být nepovinné pojmenované a nepovinné poziční, jazyk Dart tedy nepoužívá na rozdíl od jazyka Java přetěžování metod. Nepovinné pojmenované parametry se píšou do složených závorek a při deklaraci se každému parametru připíše název. Při volání funkce se pak může v parametrech uvést název parametru a za dvojtečkou jeho hodnota. Nepovinný poziční parametr se liší v tom, že je definován v hranatých závorkách a nemá žádný název. Pokud je třeba tomuto parametru předat hodnotu, na pozici tohoto parametru se při volání funkce udá hodnota. Základní rozdíl mezi těmito dvěma typy parametrů je tedy v pořadí, ve kterém jsou uvedeny v deklaraci funkce. Oba způsoby najednou se nemohou používat. Více v níže uvedených příkladech.

```
1  // Funkce s nepovinnými pozičními parametry
2  String selectQuery(List columns,String table,[String order = 'DESC'] ) {
3    return 'SELECT ' + columns.join(', ') +' FROM '+ table + ' ' + order;
4  }
5  // SELECT first_name, last_name, birthday FROM users DESC
6  String select = selectQuery(
7                  ['first_name', 'last_name', 'birthday'], 'users');
8  // Zkácený zápis funkce. Vrací číslo aktuálního měsíce např.  1
9  int getMonth() => new DateTime.now().month;
10 print(getMonth());
11
12 // Funkce s nepovinnými pojmenovanými parametry
13 DateTime getDateTimeInstance(int year, {int day: 1, int month: 1}){
14   return new DateTime(year, day, month);
15 }
16 print(dateTime.toString()); // 2004-05-08 00:00:00.000
17 print(getDateTimeInstance(2016)); // 2016-01-01 00:00:00.000
18 print(getDateTimeInstance(2016, month: 3)); // 2016-01-03 00:00:00.000
```

Iterace nad množinou dat je možné provádět pomocí kontrukce *for*, *for in*, *while* nebo je možné použít *callback*. Cyklus *for in* je obdobou cyklu *foreach* z jiných jazyků jako je například PHP/Java. [6]

```
1  List months = ['leden', 'únor', 'březen', 'duben', 'květen', 'červen',
2  'červenec', 'září', 'říjen', 'listopad', 'prosinec'];
3  for (var i = 0; i < months.length; i++) {
4    print(months.elementAt(i));
5  }
6  for (var month in months) {
7    print(month);
8  }
9
10 months.forEach((month) => print(month));
11
12 int i = 0;
13 while (i < months.length) {
14   print(months.elementAt(i));
15   i++;
16 }
17
18 int j = 0;
19 do {
20   print(months.elementAt(j));
21   j++;
22 } while (j < months.length);
```

2.4 Rozsah platnosti proměnných

Každá proměnná v kódu má svůj rozsah platnosti[8], ve kterém je možné k takové proměnné přistupovat. Existují dva základní druhy rozsahu platnosti:

- globální
- lokální

Globální proměnná je přístupná odkudkoli. Lokální má platnost pouze v rámci části kódu například v bloku funkce nebo podmínky.

Dart je jazyk, jehož proměnné mají na rozdíl od JavaScriptu pouze lexikální rozsah platnosti, což znamená, že rozsah platnosti je dán strukturou kódu. Pro zjednodušení může sloužit pravidlo složených závorek. Každá proměnná má rozsah platnosti daný složenými závorkami, mezi kterými se nachází například funkce, podmínka nebo cyklus. [6]

```
1  int age = 24; // globální proměnná
2  void scoping() {
3    String name = 'Roman'; // lokální proměnná
```

[8] Anglicky „scope“

```
4     if (true) {
5       String lastname = 'Navratil';
6       print(name); // Vypíše "Roman", protože vnitřní blok má přístup k
    proměnným vnějšího bloku
7       name = 'Lukas';
8       print(lastname); // "Navratil"
9       age = 20;
10      print(age); // "20"
11    }
12    print(lastname); // Chyba. Proměnná lastname má rozsah platnosti pouze
    v rámci podmínky
13    void innerFunction() {
14      print(age); // "20"
15      print(name); // "Lukas"
16      print(lastname); // Chyba
17      outsideFunction();
18    }
19    innerFunction();
20  }
21  void outsideFunction() {
22    double height = 189.5;
23    print(height); // 189.5
24    print(age); // 20
25    print(name); // Chyba
26    print(lastname); // Chyba
27  }
```

2.5 Klíčové slovo this

V jazyce Dart klíčové slovo *this* odkazuje na aktuální instanci, tak jak je to běžné i v jiných jazycích. Existují ale výjimky. Argumenty konstruktoru nadřazené třídy nemají přístup k proměnné *this*. Tovární konstruktory také nemají možnost používat *this*.

V níže uvedené ukázce je znázorněno, že nezáleží na kontextu, ze kterého je volána metoda používající *this*. Nehledě na to, zda je *this* používáno z vnitřní funkce nebo je metoda uložena do proměnné, která je později zavolána jako funkce, *this* vždy odkazuje na aktuální instanci objektu. JavaScript se v těchto případech chová odlišně, což je popsáno v kapitole 3.1.4. [6]

```
1   class Student {
2     String firstname;
3     String lastname;
4     DateTime birthday;
5     Student(firstname, lastname, birthday) {
6       this.firstname = firstname;
7       this.lastname = lastname;
8       this.birthday = birthday;
9     }
10    getAge() {
11      calculateAge() {
12        var currentDate = new DateTime.now();
13        var diff = currentDate.difference(this.birthday);
14        return diff.inDays / 365;
15      }
16      return calculateAge();
```

```
17    }
18    String toString() {
19      return this.firstname + ' ' + this.lastname + ', ' +
20             this.birthday.toString();
21    }
22  }
23  void logger(toString) {
24    print(toString());
25  }
26  void main () {
27    Student s = new Student('Roman', 'Navratil', new DateTime.utc(1991, 8,
    5));
28    print(s.toString());
29  // Roman Navratil, 1991-08-05 00:00:00.000Z
30    logger(s.toString());
31  // Roman Navratil, 1991-08-05 00:00:00.000Z
32    print(s.getAge()); // 24
33    var studentToString = s.toString();
34    print(studentToString());
35  // Roman Navratil, 1991-08-05 00:00:00.000Z
36  }
```

2.6 Třídy a dědičnost

Dart je objektově orientovaný jazyk s dědičností založenou na tzv. mixinech. Každý objekt je instancí třídy a každá třída dědí od třídy Object. Pro vytvoření třídy se používá klíčové slovo *new*. Reference na třídní proměnné nebo metody se provádí pomocí tečky.

```
1   class Student {
2
3     String firstName;
1     String lastName;
2     DateTime birthday;
3
4     Student(String firstName, String lastName, DateTime birthday) {
5       this.firstName = firstName;
6       this.lastName = lastName;
7       this.birthday = birthday;
8     }

9     String toString() {
10      Map data = {
11        'first_name' : this.firstName,
12        'last_name' : this.lastName,
13        'birthday' : this.birthday.toString()
14      };
15
16      return JSON.encode(data);
17    }
18  }
```

Konstruktory v Dartu se zapisují jako funkce, jejíž název je stejný jako název třídy. Takový konstruktor bez argumentů je nazýván výchozí. Třída, která dědí od jiné třídy, nedědí její konstruktor. Existuje několik speciálních typů konstruktorů. Jsou to:

- Pojmenované
- Přesměrující [9]
- Konstantní
- Tovární

Pojmenované konstruktory umožňují definovat třídě několik konstruktorů. Objekt dané třídy se pak může vytvářet několika způsoby podle potřeby.

Přesměrující konstruktor je pojmenovaný konstruktor, který v těle metody provádí pouze přesměrování vykonávání kódu na jiný konstruktor stejné třídy.

Konstantní konstruktor slouží na vytváření objektů, které jsou po vytvoření neměnné (immutable).

Tovární konstruktory je možné použít u tříd, kde konstruktor nevytváří nutně nový objekt. Takový konstruktor může například vracet novou instanci z cache nebo instanci podtypu. Tovární konstruktor ale nemá přístup k *this*.

Dart podporuje abstraktní i statické třídy a metody, rozhraní i dědičnost. Je ale třeba pamatovat na to, že třídy mezi sebou nedědí jiné než výchozí konstruktory. Před spuštěním konstruktoru potomka se pokouší spustit výchozí konstruktor rodičovské třídy bez argumentů. Pokud takový konstruktor neexistuje, je nutné jej explicitně zavolat.

```
1   class Person{
2     String firstname;
3     String lastname;
4     DateTime birthday;
5     Person(this.firstname, this.lastname, this.birthday);
6     int getAge() {
7       DateTime currentDate = new DateTime.now();
8       Duration diff = currentDate.difference(this.birthday);
9       return (diff.inDays / 365).floor();
10    }
11    String toString() {
12      return this.firstname + ' ' + this.lastname + ', ' +
13      this.birthday.toString();
14    }
15  }
16  class Student extends Person {
17    int studentId;
18    String specialization;
19    static String university = 'UHK';
```

[9] Redirecting constructor

```
20   Student(int studentId, String specialization, String firstname,
21           String lastname, DateTime birthday) :
22           super(firstname, lastname, birthday) {
23     studentId = studentId;
24     specialization = specialization;
25   }
26   List getSubjects() {
27     return ['ZMI', 'UOMO', 'TNPW', 'PRO'];
28   }
29   String toString() {
30     Map data = {
31       'first_name' : this.firstname,
32       'last_name' : this.lastname,
33       'birthday' : this.birthday.toString()
34     };
35     return JSON.encode(data);
36   }
37 }
```

Dart podporuje také tzv. callable třídy, což jsou třídy, které se chovají stejně jako funkce. Pro zavolání takové funkce je třeba vytvořit instanci objektu třídy, která obsahuje metodu *call()*, a uložit do proměnné. Pomocí této proměnné je pak možné zavolat třídu jako funkci s potřebnými parametry. [6]

```
1   class AgeCalculator {
2     int call(DateTime birthday) {
3       DateTime currentDate = new DateTime.now();
4       Duration diff = currentDate.difference(birthday);
5       return (diff.inDays / 365).floor();
6     }
7   }
8   main() {
9     var ag = new AgeCalculator();
10    ag(new DateTime(1992, 2, 28)); // 24
11  }
```

2.7 Statické atributy a metody

Pro definici třídních neboli statických atributů a metod v Dartu používá klíčové slovo *static.* Statické metody nejsou definovány objektu ale třídě, tudíž nemají přístup k proměnné *this.* [6]

```
1   class Student {
2     static String university = "UHK";
3     String firstname;
4     String lastname;
5
6     Student(this.firstname, this.lastname);
7   }
8   main () {
9     print(Student.university);
10  }
```

2.8 Rozhraní

Každou třídu je možné použít i jako rozhraní, které jiná třída může implementovat. Je-li třeba, aby třída A obsahovala stejné metody a atributy jako třída B, nemusí od třídy B dědit, ale může ji implementovat. Konstruktory nejsou do rozhraní zahrnuty. Třída může implementovat více než jedno rozhraní.

```
1   class PersonInterface {
2     String firstname;
3     String lastname;
4     DateTime birthday;
5     String toString();
6   }
7   class Person implements PersonInterface {
8     String firstname;
9     String lastname;
10    DateTime birthday;
11    Person(this.firstname, this.lastname, this.birthday);
12    int getAge() {
13      DateTime currentDate = new DateTime.now();
14      Duration diff = currentDate.difference(this.birthday);
15      return (diff.inDays / 365).floor();
16    }
17    String toString() {
18      return this.firstname + ' ' + this.lastname + ', ' +
19      this.birthday.toString();
20    }
21  }
```

Pokud je třída používána jako rozhraní, ale implementuje metodu, implementující třída musí tuto metodu implementovat a je tak přepsána implementace v rozhraní. [6]

2.9 Abstraktní třída

Z abstraktních tříd nelze vytvářet instance. Slouží především k definici metod, které jiné třídy mohou zdědit nebo mají implementovat. Abstraktní třída se značí klíčovým slovem *abstract*. Metody třídy mohou být také abstraktní. Nejsou ale definovány žádným klíčovým slovem. Není-li ve složených závorkách uvedena implementace metody, jedná se o metody abstraktní. [6]

```
1   abstract class Person {
2     String firstname;
3     String lastname;
4     DateTime birthday;
5
6     int getAge() {
7       DateTime currentDate = new DateTime.now();
8       Duration diff = currentDate.difference(this.birthday);
9       return (diff.inDays / 365).floor();
10    }
11
```

```
12   String getName();
13 }
14
15 class Student extends Person {
16   num studentId;
17   String specialization;
18
19   Student(num this.studentId, String this.specialization, String
20           firstname, String lastname, DateTime birthday) {
21     this.firstname = firstname;
22     this.lastname = lastname;
23     this.birthday = birthday;
24   }
25   String getName() {
26     return this.firstname + ' ' + this.lastname;
27   }
28 }
```

2.10 Výjimky

Výjimky jsou chyby, které nastaly při zpracování programu. V kódu, který může skončit chybou, může být vyhozena výjimka, která přeruší zpracování programu, a která uchovává informace o chybě. Dart poskytuje dvě základní třídy pro zpracování chyb *Exception* a *Error*.

Exception třída je určena k předávání informací o chybě uživateli. Slouží jako rozhraní pro další typy vestavěných výjimek v Dartu nebo pro programátorem definované výjimky. V následujícím kódu je ukázáno, jak se definuje vlastní výjimka.

```
1  class CustomException implements Exception {
2    final String message;
3    const CustomException([this.message]);
4  }
```

Tyto výjimky mají být v jedné části aplikace vyhozeny a v jiné odchyceny a zpracovány. K vyvolání výjimky slouží klíčové slovo *throw* následované vytvořením její instance. Pro odchycení výjimky slouží *try…on/catch* blok. V bloku *try* se nachází kód, který může vyhodit výjimku. Zpracování chyby se provádí v bloku *catch* používaném v případě nutnosti pracovat s objektem výjimky, nebo *on* sloužící ke zpracování specifického typu výjimky. Dále je možné přidat nepovinný blok *finally*, v němž je kód, který proběhne vždy bez ohledu na vyhození výjimky. Dart také nabízí klíčové slovo *rethrow*, pokud je třeba zachycenou výjimku znovu vyhodit.

```
1  main () {
2    try {
3        throwException();
4    } on CustomException catch(e) {
5        // zpracování výjimky CustomException
6    } catch (e) {
```

```
7         // zpracování výjimky jiné než CustomException
8     } finally {
9         // kód, který je proveden vždy
10    }
11 }
12 void throwException() {
13   try {
14       throw new CustomException('Error');
15   } on CustomException {
16       rethrow;
17   }
18 }
```

Na ukázce je také vidět, že konstrukce *on* a *catch* lze používat společně pro případ, kdy je třeba odchytit daný typ výjimky a zároveň získat objekt výjimky.

Třída *Error* reprezentuje chybu programu, kterých by se měl programátor vyvarovat například volání funkce s nesprávnými argumenty. Jedná se běhovou výjimku. Dart nevyžaduje tento typ chyb zachycovat a zpracovávat. Nastane-li taková chyba, informace o chybě jsou uloženy do objektu *Error*, ze kterého jsou potom přístupné. [6]

2.11 Mixin

Užitečným nástrojem objektově orientovaného programování jsou také tzv. mixiny. Mixin je třída, jejíž chování je možné předat jiné třídě, aniž by taková třída musela od mixinu dědit. Také je možné několik mixinů přidat k jedné třídě a tím simulovat vícenásobnou dědičnost.

Mixinem může být jakákoliv třída, která nemá definován konstruktor. Aplikace mixinu probíhá připojením k třídě předka, jehož potomek pak může využívat metod mixinu.

Do verze 1.12 bylo u mixinů omezení, že mohly dědit pouze od třídy Object a nemohly volat konstruktor rodičovské třídy pomocí funkce *super()*. Od verze 1.13 je toto omezení zrušeno. Pro použití mixinu se používá klíčové slovo *with* za názvem třídy, za kterým následuje název mixinu. [6]

```
1  class Student extends Person with Specialization {
2    Student(String firstname, String lastname, DateTime birthday)
3        : super(firstname, lastname, birthday) {
4    }
5  }
6
7  class Person {
8    String firstname;
9    String lastname;
```

```
10    DateTime birthday;
11
12    Person(this.firstname, this.lastname, this.birthday);
13  }
14
15  abstract class Specialization {
16    String specialization = 'Applied Informatics';
17
18    void changeToAI() {
19      this.specialization = 'Applied Informatics';
20    }
21
22    void changeToIM() {
23      this.specialization = 'Information Management';
24    }
25  }
26
27  main() {
28    Student student = new Student('Roman', 'Navratil', new DateTime(1991,
    8, 5));
29    print(student.specialization); // Applied Informatics
30  }
```

2.12 Generické typy

Generické typy umožňují definovat třídy, rozhraní i metody s parametrickými datovými typy, což umožňuje psát více abstraktní kód a redukuje duplicitu. Dart umožňuje takové typy definovat. Generické typy bývají zastoupeny velkým písmenem a ohraničeny špičatými závorkami stejně jako například v jazyce Java.

V příkladu je třída *Printer<T>*, kde písmeno T je zástupným znakem pro datový typ, s metodou *printToConsole()*, která přijímá parametr typu T. Konkrétní typ *Person* je uveden ve špičatých závorkách při tvorbě instance třídy *Printer*. Do metody *printToConsole()* je pak možné poslat parametr pouze typu *Person*.

```
1   class Printer<T> {
2     printToConsole(T object) {
3       print(object.toString());
4     }
5   }
6   class Person {
7     toString() {
8       return 'Person information';
9     }
10  }
11  class Notification {
12    toString() {
13      return 'Notification information';
14    }
15  }
16  main() {
17    var p = new Printer<Person>();
18    p.printToConsole(new Person());
19    p.printToConsole(new Notification());
20    // The argument type 'Notification' cannot be assigned to the
21       parameter type 'Person'.
22  }
```

Typickým příkladem použití generických typů je datová struktura seznam, které je tak možné definovat, jakého typu mají být prvky seznamu.

Generický typ lze blíže specifikovat pomocí klíčového slova *extends*. V následující ukázce je třídě *Printer* definován generický typ, který musí být podtypem třídy Person. [6]

```
1  class Printer<T extends Person> {
2    printToConsole(T object) {
3      print(object.toString());
4    }
5  }
```

2.13 Asynchronní zpracování

Asynchronní zpracování umožňuje provádění časově náročných operací na pozadí. Program nečeká na ukončení operace a pokračuje ve zpracování kódu programu. Dart obsahuje dvě API pro práci s asynchronními požadavky Future API a Stream API.

2.13.1 Future

Objekt typu *Future* reprezentuje výsledek asynchronního zpracování. Future představuje potencionální hodnotu nebo chybu, která bude dostupná v předem neznámém čase. Pro zpracování hodnot z Future se používá callback[10] *then()* případně *catchError()* ke zpracování chyb.

V ukázce je použita třída *HttpClient*, která slouží ke stažení obsahu ze serveru přes HTTP protokol. Její metoda *getUrl()* otevírá spojení se serverem. Jelikož taková akce může trvat neznámou dobu, vrací tato metoda *Future*. Po zavolání metody *getUrl()* je dvakrát volána metoda *then()*. První volání slouží pro dodatečné nastavení parametrů požadavku a druhý pro zpracování odpovědi serveru. Na ukázce je také vidět řetězení volání metody *then()*, což je možné za předpokladu, že návratová hodnota této metody je *Future*.

```
1  var url =
2  "https://api.github.com/search/repositories?q=language:dart&in=name";
3  HttpClient client = new HttpClient();
4  client.getUrl(Uri.parse(url))
5    .then((HttpClientRequest request) {
```

[10] Callback je funkce, která je zavolána po dokončení určité operace

```
6
7     })
8     .then((HttpClientResponse response) {
9
10    })
11    .catchError((exception) => print('Error'));
```

Dart nabízí alternativu k psaní asynchronního kódu pomocí klíčových slov *async* a *await*. Pro označení funkce jako asynchronní slouží klíčové slovo *async*, které se při deklaraci funkce píše za její název. Je-li takto označená funkce spuštěna, okamžitě vrací *Future* a tělo funkce je naplánováno na pozdější zpracování. Klíčové slovo *await* slouží k označení výrazu, který má být zpracován asynchronně, a očekává se, že návratová hodnota bude *Future*. Pokud není, návratová hodnota je obalena třídou *Future*. *Await* výraz je nejprve vyhodnocen a poté je pozastaveno zpracování funkce, dokud není připraven výsledek. Výsledkem *await* výrazu je výstup *Future*. Pro zpracování výjimek není potřeba používat funkci *catchError()*, ale kód obalit do *try…catch* bloku. Aby bylo možné používat *await*, musí být funkce definována jako *async*.

Následující ukázka je upravený kód předchozí ukázky s použitím *async* a *await*.

```
1   main() {
2     var url =
3   "https://api.github.com/search/repositories?q=language:dart&in=name";
4     asyncRequest(url);
5   }
6
7   asyncRequest(url) async {
8     try {
9       HttpClient client = new HttpClient();
10      HttpClientRequest request = await client.getUrl(Uri.parse(url));
11      var response = await requestSettings(request);
12      await handleResponse(response);
13    } catch (e) {
14      print(e);
15    }
16  }
17  getResponse(HttpClientRequest request) {}
18  handleResponse(HttpClientResponse response) {}
```

Na rozdíl od běžných funkcí asynchronní funkce mění datový typ vráceného objektu. Pokud je například vrácen řetězec z asynchronní funkce, ve skutečnosti by byl tento řetězec obalen do třídy *Future* a datový typ návratové hodnoty by tak byl *Future<String>*. [6]

2.13.2 Stream

Stream reprezentuje sekvenci událostí. Jednotlivé události mohou posílat nová data nebo hlásit chybu. Dohromady jsou výsledkem jednoho výpočtu. Stream se používá například při čtení souborů nebo pro zpracování HTML událostí. Konec Streamu je dán událostí „done“.

Aby Stream začal generovat události, musí se mu naslouchat funkcí *listen()*.

Streamy je možné používat dvěma způsoby. Buď je jednomu Streamu přiřazen jeden listener nebo je Streamu přiřazeno několik listenerů, kterým jsou zasílána data.

Stream s jedním listenerem začíná vysílat události, jakmile mu je přiřazen listener. Pokud je listener Streamu odebrán, okamžitě přestává posílat události, i kdyby měl nějaké k odeslání. Tyto streamy se používají na čtení velkého množství dat například velký soubor.

Druhý typ streamů s více listenery vysílá události, jakmile nastanou bez ohledu na to, zda existuje listener, který události přijímá.

V následující ukázce je dotazováno API, které vrací data ve formátu JSON. Objekt HttpClientResponse, který obsahuje odpověď serveru, je streamem. Na řádku číslo 11 je tomuto streamu nasloucháno pomocí funkce *listen()*. Data, která stream posílá, jsou ukládány do zásobníku a, jakmile jsou všechna data z odpovědi serveru přečtena, jsou vypsána do konzole.

```
1  main() {
2    var url =
3  "https://api.github.com/search/repositories?q=language:dart&in=name";
4    HttpClient client = new HttpClient();
5    StringBuffer buffer = new StringBuffer();
6    client.getUrl(Uri.parse(url))
7      .then((HttpClientRequest request) {
8        return request.close();
9      })
10     .then((HttpClientResponse response) {
11       response.transform(UTF8.decoder).listen((contents) {
12         buffer.write(contents);
13       }).onDone(() {
14         print(JSON.decode(buffer.toString()));
15       });
16     })
17     .catchError((exception) => print('Error'));
18 }
```

Pro streamy Dart nabízí asynchronní *for* cyklus *await for*, který stejně jako *await* u Future, slouží pro zpřehlednění kódu. Cyklus musí mít na pravé straně od klíčového slova *in* výraz typu *Stream*.

```
1  handleResponse(HttpClientResponse response) async {
2    StringBuffer buffer = new StringBuffer();
3    await for (var content in response.transform(UTF8.decoder)) {
4        buffer.write(content);
5    }
6    return buffer.toString();
7  }
```

Await for cyklus čeká na hodnotu. Jakmile přijde nějaká hodnota, je provedeno tělo cyklu, což se opakuje, dokud není stream uzavřen. Přerušit cyklus je možné pomocí klíčových slov *break* nebo *return*, což zároveň zajistí odhlášení od streamu. [6]

2.13.3 Generátory

Dart disponuje takzvanými generátory, což jsou funkce, které vytvářejí sekvence dat. Jsou dva druhy generátorů synchronní a asynchronní.

Synchronní generátory produkují hodnoty na vyžádání. Hodnota z funkce používající synchronní generátor je dodána pouze v případě, že si o ni jiný kód požádá.

Pro označení funkce, která je synchronním generátorem, se používá klíčové slovo *sync**. Po zavolání této funkce okamžitě vrací hodnotu typu Iterable, ze které je možné získat iterátor. Tělo funkce se začne provádět, jakmile je zavolána metoda iterátoru *moveNext()*, dokud nenarazí na klíčové slovo *yield*. Za tímto slovem je výraz, který určuje hodnotu další položky v řadě. Následně je funkce pozastavena a čeká na další zavolání *moveNext()*. Vlastnost iterátoru *current* umožňuje zjistit je aktuální hodnotu.

```
1  main() {
2    var oddNumbers = oddNumbers(3);
3    var it = oddNumbers.iterator;
4    print(it.current); // null
5    it.moveNext();
6    print(it.current); // 2
7    it.moveNext();
8    print(it.current); // 4
9    it.moveNext();
10   print(it.current); // 6
11   it.moveNext();
12   print(it.current); // null
13 }
14 Iterable oddNumbers(n) sync* {
```

```
15   int start = 0;
16   for (var i = 0; i < n; i++) {
17     yield start += 2;
18   }
19 }
```

Asynchronní generátory generují hodnoty v předem neznámých intervalech a jsou ihned poslány volající funkci. Pro označení generátoru slouží klíčové slovo *async**. Volání takto označené funkce okamžitě vrací hodnotu typu Stream. Jakmile je tomuto streamu přidán listener, začne se provádět tělo funkce generátoru, ve kterém se pomocí klíčového slova *yield* posílají hodnoty do streamu. [6]

```
1  main() async {
2    await for (var value in asyncOddNumbers(1000)) {
3      print(value);
4    }
5  }
6  Stream asyncOddNumbers(n) async* {
7    int start = 0;
8    for (var i = 0; i < n; i++) {
9      yield start += 2;
10   }
11 }
```

2.14 Metadata

Metadata se používají pro doplnění dat o kódu, které začínají znakem zavináče následovaný buď referencí na konstantu, nebo voláním konstantního konstruktoru třídy. Mezi konstanty patří například *@deprecated* a *@override*. Pro definici vlastní metadata anotace je třeba definovat třídu s konstantním konstruktorem.

```
1  class task {
2    final String title;
3    final String description;
4
5    const task(this.title, this.description);
6  }
7
8  @task('Calculate age', 'Create function which return age from birthday
   date')
9  int calculateAge(DateTime birthday) {}
```

Metadata je pak možné získat pomocí reflexe. [6]

2.15 Reflexe

Reflexe slouží ke zkoumání struktury aplikace za běhu programu. Takto prozkoumat lze deklarace třídy, knihovny, instance a další. Dart reprezentuje názvy deklarací instancí třídy Symbol. Symboly slouží jako identifikátor deklarovaný v aplikaci. K použití symbolu slouží znak mřížky, za níž se nachází název symbolu.

Dart používá reflexi založenou na *mirrors*, což jsou zrcadlové odrazy reprezentované třídou, které představují různé entity jazyka Dart. Například pro reflektování třídy je nejprve nutné získat její zrcadlový odraz reprezentovaný třídou ClassMirror. Z této třídy lze získat informace o konstruktorech, metodách i obsažených vlastnostech.

Následující ukázka získává názvy vlastností třídy Person. Nejprve je pomocí funkce *reflectClass()* získána instance třídy ClassMirror. Po té se prochází všemi deklaracemi (metody, vlastnosti, kontsruktory atd.) a vybírají se deklarace pouze typu VariableMirror, který reprezentuje třídní vlastnosti. Posléze jsou získány a vypsány názvy vlastností třídy.

```
1  import 'dart:mirrors';
2  main () {
3    ClassMirror mirror = reflectClass(Person);
4
5    var fields = mirror.declarations.values.where(
6                   (m) => m is VariableMirror
7                 );
8    for (var f in fields) {
9      print(MirrorSystem.getName(f.simpleName));
10   }
11 }
12 class Person {
13   String firstname;
14   String lastname;
15   DateTime birthday;
16   Person(this.firstname, this.lastname, this.birthday);
17   int getAge() {
18     DateTime currentDate = new DateTime.now();
19     Duration diff = currentDate.difference(this.birthday);
20     return (diff.inDays / 365).floor();
21   }
22   String toString() {
23     return this.firstname + ' ' + this.lastname + ', ' +
24     this.birthday.toString();
25   }
26 }
```

Pomocí reflexe lze také vykonávat metody objektů. K tomuto účelu slouží metoda *invoke()* třídy InstanceMirror. Metoda přijímá dva parametry. První parametr specifikuje metodu, která má být spuštěna pomocí symbolu. Druhým parametrem je předán seznam argumentů metody.

Ukázka spouštění metody objektu navazuje na předchozí příklad, kde je definovaná třída Person. Následující kód pomocí reflexe volá metodu třídy Person *getAge()*, jejímž symbolem je název metody bez závorek.

```
1   Person person = new Person(
2       'Roman', 'Navratil', new DateTime(1991, 5, 8)
3   );
4   InstanceMirror imirror = reflect(person);
5   imirror.invoke(#getAge, []);
```

Reflexe v Dartu je autory označená za nestabilní (nedoporučuje se její využití v produkčním prostředí) a stále se pracuje na jejím vývoji. [6]

3 Alternativy k jazyku Dart

V následující kapitole jsou popsány tři programovací jazyky, které se nejčastěji srovnávají s jazykem Dart a jsou často používanými alternativami pro vývoj webových aplikací. Jedná se o jazyky TypeScript, CoffeeScript a JavaScript.

3.1 JavaScript

Jazyk byl představen v květnu roku 1995 nazvaný jako Mocha. V září téhož roku byl přejmenován na LiveScript a v prosinci, z důvodu tehdy nesmírné popularity jazyka Java, byl nazván JavaScript. Od roku 1996 do roku 1997 společnost ECMA (European Computer Manufacturers Association) vytvářela první oficiální specifikaci, která vyšla pod názvem ECMAScript, a JavaScript se stal nejznámější implementací tohoto standardu. Další verze vyšly v roce 1998 a 1999 jako ECMAScript 2 respektive ECMAScript 3. Práce na ECMAScriptu 4 začaly v roce 2004, ale nakonec tento standard nikdy nebyl dokončen kvůli špatné spolupráci firem Microsoft a Netscape a až v roce 2009 vznikla další verze a to rovnou verze 5. Před šestou verzí, která vyšla v roce 2015, vyšla ještě menší verze bez nových funkcí 5.1. [9] V době vypracování této práce nejpopulárnější prohlížeče téměř úplně podporují ECMAScript 5.1[11] a pracují na implementaci standardu ECMAScript verze 6.[12]

Tvůrcem JavaScriptu je Brendan Eich, který na tomto jazyku začal pracovat, když byl přijat do firmy NetScape v roce 1995. Předtím pracoval na operačním systému a síťování ve firmě Silicon Graphics. V roce 1998 spoluzaložil webovou stránku mozilla.org, která si dala za úkol vydat prohlížeč Netscape jako open source. Po té co Nescape koupila a později zrušila firma AOL, pomohl založit prohlížeč Firefox v roce 2004. [10] O deset let později Eich opustil firmu Mozilla. [11] Rok a půl po svém odchodu založil společnost Brave Software, která pracuje na prohlížeči s důrazem na bezpečnost a ochranu soukromí uživatele. [12]

---

[11] http://kangax.github.io/compat-table/es5/
[12] http://kangax.github.io/compat-table/es6/

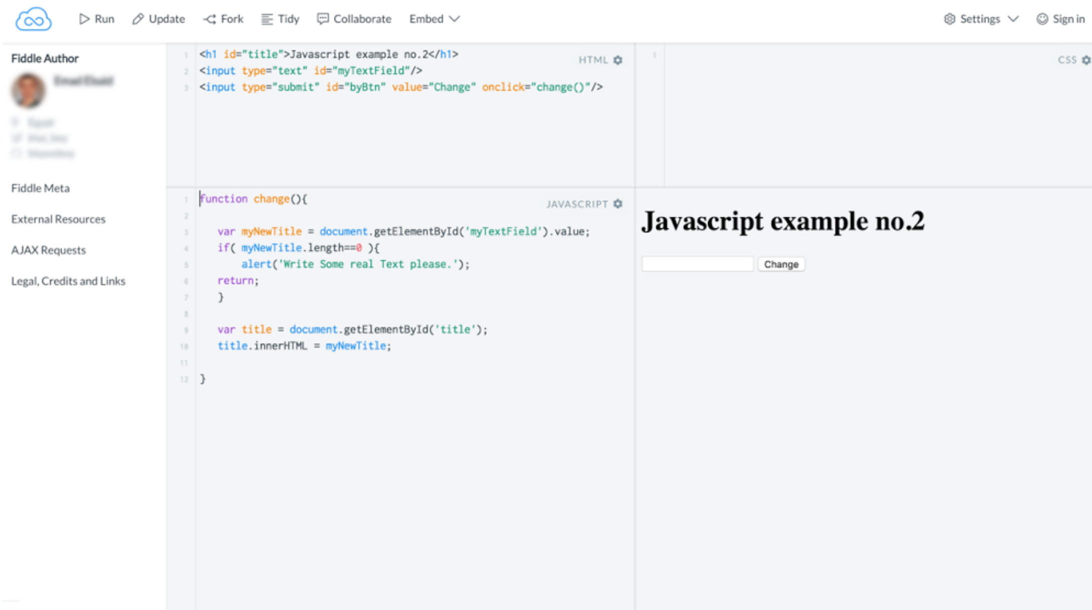
JavaScript byl původně vytvořen jako jednoduchý interpretovaný jazyk, který umožňuje přidávat webovým stránkám interaktivitu. Postupem času se stal oblíbeným programovacím jazykem pro web. V době psaní práce existuje značné množství knihoven a frameworků napsaných právě v JavaScriptu.[13] Díky podpoře všech hlavních prohlížečů a přidávání nových funkcí se vytvářejí i celé aplikace primárně v jazyce JavaScript.

3.1.1 Nástroje

K testování kódu v jazyce JavaScript v podstatě stačí pouze webový prohlížeč, protože dnešní prohlížeče disponují konzolí, kde je možné přímo psát JavaScriptový kód. Dále existují vývojová prostředí jako například WebStorm, který podporuje mimo jiné i jazyk JavaScript, nebo textové editory například Atom nebo Sublime Text. Další možností jsou online konzole (editory), které fungují podobně jako konzole v prohlížečích, ale jsou propracovanější a nabízí více funkcí, jako například jednoduché sdílení kódu. Příkladem online konzole je JSFiddle nebo JSBin.

Kód v jazyce JavaScript lze také testovat spouštěním aplikace z příkazové řádky. To umožňuje například Node.js, což je běhové prostředí pro JavaScript postavené na enginu V8. Po instalaci platformy Node.js je možné spouštět i psát skripty v příkazové řádce. [13]

[13] http://githut.info/



Obrázek 2 Grafické rozhraní nástroje JsFiddle. Zdroj: vlastní zpracování

Kromě JSFiddle lze ukázky kódu jazyka JavaScript zkopírovat do výše uvedených nástrojů a z příkazové řádky spustit. Ukázky používají pro výpis do konsole, ale nástroj JSFiddle jej nepodporuje.

3.1.2 Základní syntaxe

JavaScriptová syntaxe je založena na syntaxi jazyku Java. JavaScript rozlišuje velikost písmen například v názvu proměnné nebo funkce. JavaScript není striktně typový jazyk. To znamená, že datové typy proměnných si odvozuje sám. Explicitně se tedy nedefinují.

```
1  var firstName = 'Roman';
2  var age = 24;
3  var height = 189.5;
4  var months = ['leden', 'unor', 'brezen'];
```

Blok kódu tvoří složené závorky. Syntaxe podmínek je totožná s TypeScriptem i Dartem.

```
1  var number = 6;
2  if (number % 2 == 0) {
3    console.log('Cislo je sude');
4  } else {
5    console.log('Cislo je liche');
6  }
```

Funkcím ve verzi JavaScriptu, která implementuje ECMAScript verze 5.1, nelze definovat výchozí parametry, ale standard ECMAScript 6 dovoluje zapsat i výchozí

parametry funkcí. Datové typy v jazyce JavaScript jsou pouze implicitní, datové typy parametrů ani datový typ návratové hodnoty se nezapisuje.

```
1  function selectQuery(columns, table, order) {
2    return 'SELECT ' + columns.join(', ') + ' FROM ' + table + ' ' +
3          order;
4  }
5  console.log(
6    selectQuery(['first_name', 'last_name', 'birthday'], 'users', 'ASC')
7  );
```

Jelikož jsou paramtery funkce v JavaScriptu nepovinné, neumožňuje přetěžování funkcí. Funkci *selectQuery()* lze zavolat s jedním nebo žádným parametrem, aniž by JavaScript zahlásil chybu, že funkce s těmito parametry neexistuje.

Výchozích parametrů funkcí ve starších verzích JavaScriptu než ECMAScript 6 lze dosáhnout kontrolou, zda jsou parametry inicializované. Pokud nejsou, je jim nastavena výchozí hodnota - *undefined*.

```
1  function selectQuery(columns, table, order) {
2    columns = typeof columns !== 'undefined' ? columns : '*';
3    table = typeof table !== 'undefined' ? table : 'users';
4    order = typeof order !== 'undefined' ? order : 'DESC';
5
6    return 'SELECT ' + columns.join(', ') + ' FROM ' + table + ' ' +
   order;
7  }
```

I u cyklů není mezi Dartem, TypeScriptem a JavaScriptem žádný rozdíl. *For* cyklů má však JavaScript několik typů. Prvním je *for* cyklus, který je podobný *for* cyklu z jiných jazyků například Java. Takový cyklus má tři parametry: inicializátor, podmínku a inkrement. Druhý typ *for* cyklu je *for...in*, který slouží pro iteraci nad názvy vlastností objektu (více o vlastnostech objektů v kapitole 3.1.5). [14]

```
1  var months = ['leden', 'únor', 'březen', 'duben', 'květen', 'červen',
2  'červenec', 'září', 'říjen', 'listopad', 'prosinec'];
3
4  for (var i = 0; i < months.length; i++) {
5    console.log(months[i]);
6  }
7
8  var Building = function (name, height, floorsCount) {
9    this.name = name;
10   this.height = height;
11   this.floorsCount = floorsCount;
12 };
13
14 var UHKBuilding = new Building('UHK', 40, 4);
15
16 for (var p in UHKBuilding) {
17   console.log(p); // vypíše „name“, „height“, „floorsCount“
18 }
19
```

```
20 var i = 0;
21 while (i < months.length) {
22   console.log(months[i]);
23   i++;
24 }
25
26 var j = 0;
27 do {
28   console.log(months[j]);
29   j++;
30 } while (i < months.length);
```

3.1.3 Rozsah platnosti proměnných

JavaScript stejně jako Dart má lexikální rozsah platnosti ale pouze v rámci funkcí.

JavaScript má na rozdíl od Javy několik rozsahů platnosti proměnných, což často mate vývojáře, kteří jazyk tolik neznají. JavaScript má rozsah lexikální, v rámci funkce, globální a další. Popis jednotlivých rozsahů je nad rámec této závěrečné práce. Bližší informace lze nalézt v [28].

V případě, že proměnná je deklarována například v bloku podmínky, tato proměnná je pak přístupná i mimo takovou podmínku. Je-li několik funkcí do sebe vnořených, vnitřní funkce dědí rozsah platnosti od nadřazené funkce. [14]

```
1  var age = 24; // globální proměnná
2  function scoping() {
3    // lokální proměnná pro funkci scoping a všechny vnitřní funkce
4    var name = 'Roman';
5    if (true) {
6      var lastName = 'Navratil';
7      console.log(name); // "Roman"
8      name = 'Lukas';
9      console.log(lastName); // "Navratil"
10     age = 20;
11     console.log(age); // "20"
12   }
13   console.log(lastName); // Není chyba a vypíše se "Navratil"
14   function innerFunction() {
15     var weight = 85; // lokální proměnná funkce
16     console.log(age); // "20"
17     console.log(name); // "Roman"
18     console.log(lastName); // "Navratil"
19     outsideFunction();
20   }
21   console.log(weight); // Chyba
22   innerFunction();
23 }
24 function outsideFunction() {
25   var height = 189.5;
26   console.log(height); // "189.5"
27   console.log(age); // "20"
28   console.log(name); // Chyba
29   console.log(lastName); // Chyba
30 }
31 scoping();
```

3.1.4 Klíčové slovo *this*

Každá funkce v JavaScriptu má, stejně jako objekty, vlastnosti[14]. Kdykoliv je taková funkce zavolána získá vlastnost *this*, či-li proměnnou, jejíž hodnotou je objekt, nad kterým je funkce spuštěna. Jinými slovy je hodnota proměnné *this* závislá na kontextu, ve kterém byla funkce volána. Výjimkou jsou anonymní funkce, které nejsou přiřazeny žádnému objektu. Kvůli takovému chování JavaScriptu může být používání proměnné *this* problematické, protože bude odkazovat na neočekávaný objekt. Problém nastává v těchto případech:

- funkce objektu je předána jiné funkci
- *this* je použito ve vnořené funkci
- funkce je uložena do proměnné

Následuje ukázka kódu tří výše zmíněných situací. [14]

```
1  var Student = function (firstName, lastName, birthday) {
2    this.firstName = firstName;
3    this.lastName = lastName;
4    this.birthday = birthday;
5  };
6  Student.prototype.toString = function () {
7    return this.firstName + " " + this.lastName + ", " + this.birthday;
8  };
9  Student.prototype.getAge = function () {
10   // this použito ve vnitřní funkci
11   var calculateAge = function () {
12     var currentDate = new Date();
13     var diff = currentDate - this.birthday;
14     // this.birthday je undefined
15     return Math.floor(diff / (1000*60*60*24*365));
16   }
17   return calculateAge();
18 }
19 function logger(toString) {
20   console.log(toString());
21 }
22 var s = new Student('Roman', 'Navratil',
23                     new Date(Date.UTC(1991, 7, 5)));
24 console.log(s.toString());
25 // Roman Navratil, Mon Aug 05 1991 02:00:00 GMT+0200 (CEST)
26
27 // Funkce objektu je předána jiné funkci
28 logger(s.toString); // undefined undefined, undefined
29 console.log(s.getAge()); // NaN
30 // Funkce uložena do proměnné
31 var studentToString = s.toString;
32 console.log(studentToString()); // undefined undefined, undefined
```

---

[14] Anglicky property. V jiných jazycích se jedná o atributy.

3.1.5 Třídy a dědičnost

Třídy se v JavaScriptu definují jako funkce přiřazená proměnné. Zároveň je tato funkce konstruktorem, kód této funkce se tedy provede při vytváření instance dané třídy. Chování třídy se definuje jako funkce, které se přiřadí prototypu třídy.

```
1   var Student = function (firstName, lastName, birthday) {
2     this.firstName = firstName;
3     this.lastName = lastName;
4     this.birthday = birthday;
5   }
6
7   Student.prototype.toString = function () {
8     return JSON.stringify({
9       firstName: this.firstName,
10      lastName: this.lastName,
11      birthday: this.birthday
12    });
13  };
14
15  console.log(new Student('Roman', 'Navratil',
16              new Date(Date.UTC(1991, 7, 5))).toString());
```

Na rozdíl od Dartu je objektové programování v jazyce JavaScript založené na prototypech. Každá funkce v JavaScriptu je objektem s vlastností *prototype*, která je ve výchozím stavu prázdná, k níž je možné připojit metody a vlastnosti. Tato vlastnost funkce se používá k dosažení dědičnosti v JavaScriptu. Zatímco ostatní jazyky v této práci používají pro dědičnost klíčové slovo *extend*, JavaScript takové klíčové slovo nemá. Pro přiřazení vlastnosti prototype jednoho objektu druhému se používá funkce *Object.create()*. Object je vestavěný objekt a všechny objekty od něj dědí. Po použití funkce *create()* je ale nastaven konstruktor potomka na konstruktor předka. Z toho důvodu je třeba explicitně přiřadit konstruktoru potomka správný konstruktor. V konstruktoru potomka je jako první zavolán konstruktor předka pomocí funkce *call()*, která zajistí správné nastavení *this*.

V následující ukázce kódu je definován objekt *Person*, jehož vlastnosti *prototype* jsou přiřazeny funkce s názvy *getAge()* a *toString()*. Dále je vytvořen objekt *Student*. Důležitý je řádek 25, kde je vytvořen objekt z prototypu Person a přiřazen vlastnosti *prototype* objektu *Student*. To zajišťuje, že objekt Student zdědí vlastnosti a chování objektu *Person*.

```
1   // Konstruktor rodičovského objektu
2   var Person = function (firstname, lastname, birthday) {
3     this.firstname = firstname;
4     this.lastname = lastname;
5     this.birthday = birthday;
```

```
6  };
7  Person.prototype.getAge = function () {
8    var currentDate = new Date();
9    var diff = currentDate - this.birthday;
10   return Math.floor(diff / (1000*60*60*24*365));
11 };
12 Person.prototype.toString = function () {
13   return this.firstname + ' ' + this.lastname + ', ' + this.birthday;
14 };
15 // Konstruktor potomka
16 var Student = function (studentId, specialization, firstname, lastname,
17                         birthday)
18 {
19   Person.call(this); // volání konstruktoru rodičovského objektu
20   this.studentId = studentId;
21   this.specialization = specialization;
22   this.firstName = firstname;
23   this.lastName = lastname;
24   this.birthday = birthday;
25 };
26 // přiřazení prototypu potomkovi
27 Student.prototype = Object.create(Person.prototype);
28 Student.prototype.constructor = Student; // přiřazení konstruktoru
29 Student.prototype.toString = function () {
30   return JSON.stringify({
31     studentId: this.studentId,
32     name: this.firstName + ' ' + this.lastName,
33     birthday: this.birthday,
34     specialization: this.specialization
35   });
36 };
37 Student.prototype.getSubjects = function () {
38   return ['ZMI', 'UOMO', 'TNPW', 'PRO'];
39 }
```

Druhým použitím prototypů v JavaScriptu je atribut *prototype*, který má každý objekt. [56] Tento atribut lze chápat jako popis rodičovského objektu. Používá se k přístupu k vlastnostem a metodám předka. Je-li potřeba přistoupit k proměnné objektu, JavaScript začíná hledat v daném objektu. Pokud proměnná v objektu není, přes atribut *prototype* se přistoupí k předkovi, ve kterém se JavaScript snaží najít požadovanou proměnnou, a tak dále. Hledání pokračuje, dokud je to možné. Pokud proměnná není nalezena, JavaScript vrátí *undefined*. Tato vlastnost JavaScriptu je nazývána prototypový řetězec[15].

Ukázka kódu navazuje na předchozí příklad. Je vytvořena instance objektu Student a zavolána metoda getAge, která se nachází v předkovi Person. Nejprve se zjišťuje, jestli je metoda getAge v instanci objektu Student. Jelikož není, JavaScript přes

[15] Anglicky prototype chain

atribut *prototype* vyhledá požadovanou metodu v objektu Person, kde ji najde a provede. [14]

```
1  var studentIM = new Student('4321', 'Information Management', 'Lukas',
2  'Kucera', new Date(Date.UTC(1993, 2, 20)));
3  console.log(studentIM.getAge()); // 24
```

3.1.6 Statické atributy a statické metody

Mají-li být atributy a funkce součástí instance objektu, jsou přiřazeny jeho prototypu. Pokud jsou atributy a funkce definovány přímo objektu, je možné k nim přistupovat bez vytvoření instance, tudíž fungují tak, jak statické vlastnosti a metody fungují v jiných jazycích.

```
1  var Student = function (firstname, lastname) {
2    this.firstname = firstname;
3    this.lastname = lastname;
4  };
5  Student.university = 'UHK'; // statická proměnná
6  Student.fromJSON = function (json) { // statická funkce
7    return new Student(json.firstname, json.lastname);
8  };
```

3.1.7 Rozhraní

V JavaScriptu nijak nelze pomocí rozhraní přinutit implementující objekt, aby definoval předepsané funkce. Stejně tak není možné vytvořit abstraktní objekt, jehož potomci by museli implementovat chování předka a dědit jeho vlastnosti. Je však možné toto chování simulovat vytvořením požadovaných objektů a zpětně kontrolovat, zda mají požadované funkce.

Následující ukázka kódu představuje, jak by mohlo být dosaženo chování rozhraní v JavaScriptu. V poli *PersonInterface* je seznam funkcí, které by měl objekt implementovat. Dále je definovaná funkce *implements()*, která prochází názvy funkcí v poli *PersonInterface* a kontroluje, zda funkce s takovým názvem se nachází v kontrolovaném objektu. Objekt *Person*, která má implementovat rozhraní *PersonInterface*, v konstruktoru volá funkcí *implements()*. Pokud tato funkce zjistí, že objekt *Person* neobsahuje funkce z *PersonInterface*, je vyhozena chyba. Při tvorbě instance objektu se tedy hned kontroluje, zda objekt implementuje rozhraní.

```
1  var PersonInterface = ['getName'];
2
3  function implements(object, interface) {
4    for (var i = 0; i < interface.length; i++) {
5      var methodName = interface[i];
```

```
6       if ((typeof object[methodName]) == "function") {
7         continue;
8       } else {
9         return false;
10      }
11    }
12    return true;
13  }
14
15  var Person = function (firstname, lastname) {
16    if (!implements(this, PersonInterface)) {
17      throw Error("Object does not implement interface");
18    }
19    this.firstname = firstname;
20    this.lastname = lastname;
21  };
22  Person.prototype.getName = function () {
23    return this.firstname + ' ' + this.lastname;
24  };
25
26  var person = new Person('Roman', 'Navratil');
27  console.log(person.getName()); // „Roman Navratil"
```

3.1.8 Abstraktní objekt

K docílení abstraktních objektů je postup podobný. Abstraktní objekt by měl sloužit jako základní objekt s definovaným chováním, z něhož nelze vytvořit instanci, ale jiný objekt od základního může dědit chování a vlastnosti. V ukázce je objekt Person, v jehož konstruktoru je vyhozena chyba, tudíž z tohoto objektu nelze vytvořit instance. Tento objekt definuje chování funkce *getName()*. Dále je objekt *Student*, který je potomkem objektu *Person*. Tím pádem od něj dědí funkci *getName()*.

```
1   var Person = function () {
2     this.firstname = undefined;
3     this.lastname = undefined;
4
5     throw new Error("Abstract class cannot be instantiated");
6   };
7   Person.prototype.getName = function () {
8     return this.firstname + " " + this.lastname;
9   };
10
11  var Student = function (firstname, lastname) {
12    this.firstname = firstname;
13    this.lastname = lastname;
14  };
15  Student.prototype = Object.create(Person.prototype);
16  Student.prototype.constructor = Student;
17
18  var student = new Student('Roman', 'Navratil');
19  console.log(student.getName()); // "Roman Navratil"
20
21  var person = new Person();
22  // Error: Abstract class cannot be instantiated
```

3.1.9 Výjimky

JavaScript obsahuje objekt Error, který slouží k vyvolání chyby za běhu aplikace. V konstrukce *try...catch* jsou pak chyby zachytávány a zpracovány. V JavaScriptu nelze zachytit výjimku specifického typu v bloku *catch*, lze ale na typ chyby reagovat podmínkou uvnitř bloku.

V JavaScript je možné definovat vlastní výjimky vytvořením objektu, který dědí od objektu Error. [14]

```
1   function CustomException(message) {
2     this.name = 'CustomException';
3     this.message = message;
4     this.stack = (new Error()).stack;
5   }
6   CustomException.prototype = Object.create(Error.prototype);
7   CustomException.prototype.constructor = Error;
8   try {
9     throw new CustomException('Error');
10  } catch (e) {
11    if (e instanceof CustomException) {
12      console.log(e.name); // zpracování chyb typu CustomException
13    } else {
14      console.log(e.name); // zpracování ostatních chyb
15    }
16  }
```

3.1.10 Mixin

JavaScript pro mixiny nativní podporu nemá, ale dají se simulovat. Vyžaduje to vytvoření funkce, která jako argumenty vyžaduje třídu, které se má přidat mixin, a samotný mixin. Funkce pak prochází vlastnosti objektu mixinu a přidává je prototypu jiné třídy. Taková funkce může vypadat následovně. [44]

```
1   function extend(destination, source) {
2     for (var k in source) {
3       if (source.hasOwnProperty(k)) {
4         destination[k] = source[k];
5       }
6     }
7     return destination;
8   }
```

3.1.11 Generické typy

V JavaScriptu není možné využívat výhod generických typů.

3.1.12 Asynchronní zpracování

JavaScript verze ECMAScript 6 umožňuje asynchronní zpracování pomocí objektu Promise, který reprezentuje hodnotu operace, která bude dokončena někdy

v budoucnu. Konstruktor objektu Promise přijímá callback s parametry *resolve* a *reject*, ve kterém se definuje reakce na výsledek asynchronní operace. Funkce *resolve* vrací objekt Promise s výsledkem asynchronní operace. Funkce reject je volána v případě, že nastala chyba během zpracování a vrací objekt Promise s informacemi o chybě. Promise obsahuje funkce *then()*, která slouží pro zpracování výsledků a umožňuje řetězení dalších asynchronních zpracování, a funkci *catch()* pro zpracování chyb.

```
1  var promise = new Promise(function(resolve, reject) {
2    if (true) {
3      resolve();
4    } else {
5      reject();
6    }
7  });
8  promise.then(function(result) {
9    // zpracování výsledku
10 }).catch(function(error) {
11   // zpracování chyb
12 });
```

Generátory jsou v JavaScriptu definovány funkcí označené klíčovým slovem *function** a vrací hodnoty generované podle výrazu za konstrukcí *yield*. [14]

```
1  function* oddNumbers(n) {
2    var start = 0;
3    for (var i = 0; i < n; i++) {
4      yield start += 2;
5    }
6  }
7  var oddNumber = oddNumbers(5);
8  console.log(a.next().value);
```

3.1.13 Metadata

Doplnění kódu o metadata například pomocí anotací v JavaScriptu nelze. Některé knihovny ale používají pro metadata komentáře.

3.1.14 Reflexe

Vlastnosti objektu a jejich hodnoty lze získat pomocí *for…in* cyklu. Definovat nebo upravovat metody instance objektu je možné funkcí *Object.defineProperty()*.

V příkladu je názorně ukázán výpis vlastností objektu Person a po té je pomocí funkce *Object.defineProperty()* přidána metoda *getFullname()* objektu *person*. [14]

```
1  var Person = function (firstname, lastname, birthday) {
2      this.firstname = firstname;
3      this.lastname = lastname;
```

```
4       this.birthday = birthday;
5   };
6   Person.prototype.getAge = function () {
7   };
8   Person.prototype.toString = function () {
9   };
10  var person = new Person('Roman', 'Navratil', new Date(1991, 7, 5));
11  for (var property in person) {
12    console.log('Name: ' + property + ', Value:' + person[property]);
13  }
14
15  Object.defineProperty(person, 'getFullname', {
16    value: function () {return this.firstname + ' ' + this.lastname;},
17    writable: true,
18    enumerable: true,
19    configurable: true
20  });
21  person.getFullname();
```

3.2 TypeScript

TypeScript není kompletně nový jazyk. Na oficiálních stránkách se píše, že TypeScript je *„nadmnožina JavaScriptu, která umožňuje kompilaci do Javascriptu“* (překlad autora)[16] [15]. Znamená to, že TypeScript doplňuje JavaScript o některé funkce a je s ním plně kompatibilní, tudíž je možné používat čistý Javascript v kódu TypeScriptu.

Největší předností jazyka TypeScript je kontrola a definice vestavěných i vlastních datových typů. Základním principem typové kontroly je definice datových typů pomocí rozhraní, které definuje typ proměnné, funkce nebo pole. TypeScript disponuje také několika speciálními datovými typy. Umožňuje například definici takzvaných union typů, které vymezují, jakých typů může proměnná nabývat. Dalším typem je Type Guard, což je výraz, který provádí kontrolu datového typu v určitém rozsahu nejen při překladu, ale také za běhu programu. Intersection Types říkají, z jakých typů se má objekt skládat. TypeScript také umí dát typům alias. Pomocí řetězcových typů[17] je možné určit, které řetězce může proměnná obsahovat. Posledním příkladem jsou polymorfické *this* typy[18], které reprezentují podtyp třídy nebo rozhraní. V kapitole Základní syntaxe budou tyto typy vysvětleny podrobněji s ukázkami kódu.

---

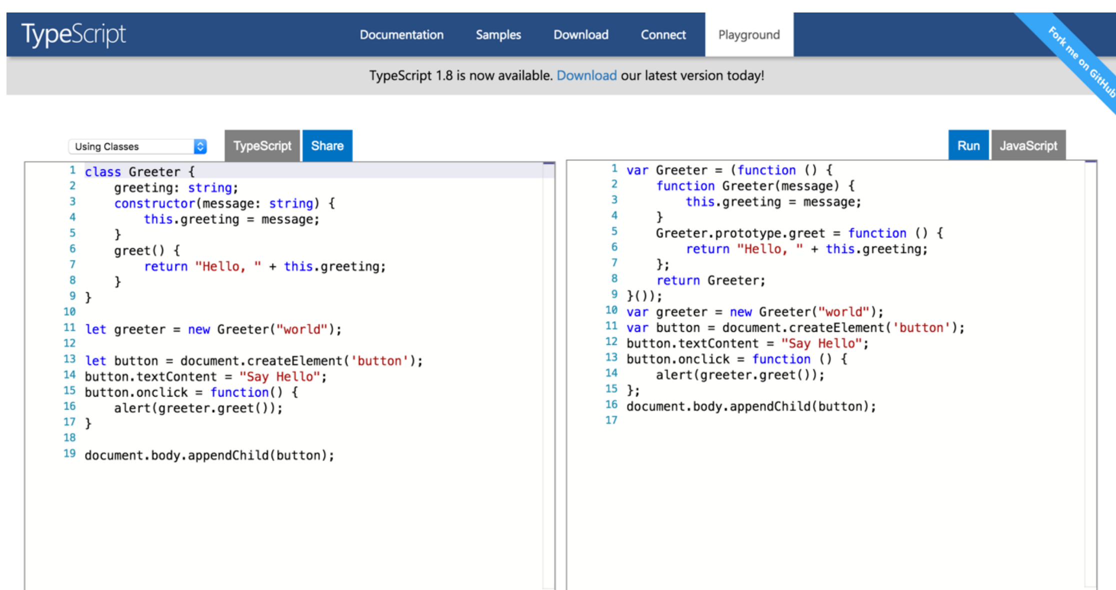
[16] TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
[17] String Literal Types
[18] Polymorphic this types

Další vlastností TypeScriptu je možnost nastavení verze JavaScriptu, do které se má kód kompilovat. Díky tomu je možné zkompilovat kód do starších verzí JavaScriptu a aplikace tak může běžet i ve starších prohlížečích [15]

TypeScript vyvíjí společnost Microsoft jako open source pod licencí Apache 2.0. [16] První veřejná verze byla představena v říjnu roku 2012. [17] Verze 1.0 byla představena 2. Dubna 2014. [18]

3.2.1 Nástroje

S vydáním první veřejné verze nebyl vydán žádný nástroj, který by TypeScript podporoval a to ani v IDE Visual Studio vyvíjené stejnou společností. Až s verzí 1.0 byla přidána nativní podpora do Visual Studio 2013 a jako doplněk pro Visual Studio 2012. Microsoft také nabízí svůj textový editor zvaný Visual Studio Code, který podporuje TypeScript. Také jsou doplňky pro textové editory Atom a Sublime Text. Existují i doplňky pro další populární textové editory vyvinuté komunitou. Dále je možné si TypeScript vyzkoušet přímo v prohlížeči. [15]



Obrázek 3 Online nástroj na vyzkoušení TypeScript bez nutnosti instalace. Zdroj: vlastní zpracování

3.2.2 Základní syntaxe

Ani TypeScript není striktně typový jazyk. Jak již bylo napsáno výše, TypeScript je nadmnožinou Javascriptu a doplňuje jej o další funkce. Jednou takovou funkcí je definice datového typu proměnných. Proměnná může být typu boolean, string, number, array, enum a any. Celá čísla i čísla s plovoucí desetinnou čárkou mají typ number. Proměnná typu any může obsahovat proměnnou, kteréhokoliv typu. Datový typ se definuje za názvem proměnné za dvojtečkou. Definuje-li se datový typ pro pole hodnot, každá tato hodnota musí být daného typu.

```
1   var firstName: string = 'Roman';
2   var lastName = 'Navratil'; // proměnná bez udání datového typu
3   var age: number = 24; // proměnná typu number může obsahovat celé i
    desetinné číslo
4   var height: number = 189.5;
5   var months: string[] = ['leden', 'unor', 'brezen'];
6
7   // promenne nejasneho typu
8   var notSure: any = 4;
9   notSure = "maybe a string instead";
10  notSure = false;
```

Podmínky a cykly se zapisují stejně jako v jazyce JavaScript. TypeScript k nim nic nepřidává, proto nejsou uvedeny žádné příklady.

Funkce je možné doplnit o datové typy parametrů a návratové hodnoty. Ve výchozím stavu jsou všechny parametry povinné. TypeScript ale umožňuje definovat nepovinný parametr a to tak, že se za jeho název napíše znak otazník. Datový typ parametrů se udává stejně jako u proměnných nejprve název a za dvojtečkou datový typ. Typ návratové hodnoty se píše za kulatou závorku v definici funkce za dvojtečku.

```
1   function selectQuery(columns: string[], table: string,
2                        order?: string): string
3   {
4     return 'SELECT ' + columns.join(', ') + ' FROM ' + table + ' ' +
5            order;
6   }
7
8   function getFirstDayOfYear(year: number, day: number = 1,
9                              month: number = 1): any
10  {
11    return new Date(year, month, day);
12  }
```

TypeScript používá pro definici vlastních typů rozhraní, pomocí kterého je možné definovat typy pro proměnné, funkce, pole i třídy. Rozhraní udává tvar, jaký má datový typ mít.

```
1   interface Student {
2     name: string;
3     address: string;
4   }
5   function logStudent(student: Student): void {
6     console.log(student);
7   }
8   var student = {
9     name: "Roman Navratil",
10    address: "Krasna 12, Krasnov 111 22"
11  };
12  logStudent(student);
```

V příkladu je definován typ Student, který musí mít vlastnosti *name* a *address*. Dále je funkce přijímající parametr tohoto typu, a nakonec je definována proměnná student s vlastnostmi *name* a *address*. Tato proměnná může být předána funkci *logStudent()* bez chyb při kompilaci, protože splňuje podmínky datového typu Student. Podmínky udávané rozhraním jsou minimální. To znamená, že objekt je typu Student, pokud má alespoň vlastnosti *name* a *address*, ale může jich mít i více.

Rozhraní také podporují nepovinné vlastnosti, které se značí otazníkem na konci názvu vlastnosti. Takové vlastnosti může, ale nemusí mít.

```
1   interface Student {
2     name: string;
3     address: string;
4     email?: string;
5   }
6   function logStudent(student: Student): void {
7     console.log(student);
8     if (student.email) {
9       console.log(student.email);
10    }
11  }
12  var student = {
13    name: "Roman Navratil",
14    address: "Krasna 12, Krasnov 111 22",
15    birthday: new Date(1991, 7, 5),
16    email: "navraro1@uhk.cz"
17  };
18  logStudent(student);
```

Dále je možné použít rozhraní pro definici tvaru funkce. Takové rozhraní neudává, jaké má mít funkce jméno. V závorkách je seznam parametrů a jejich datových typů a za dvojtečkou se nachází datový typ návratové hodnoty.

```
1  interface LogStudentFunction {
2    (student: Student): void;
3  }
4  var logStudentFunction: LogStudentFunction;
5  logStudentFunction = function(student: Student): void {
6    console.log(student);
7  };
8  logStudentFunction(student);
```

K definici tvaru datového typu pole se v rozhraní definuje, jaký typ má mít index a jakého typu mají být elementy pole. TypeScript umožňuje index pouze typu *number* nebo *string*. Typ elementu může být jakýkoli včetně vlastních typů.

```
1  interface NamesArray {
2    [index: number]: string;
3  }
4  var namesArray: NamesArray;
5  namesArray = ['Roman', 'Lukas', 'Tomas'];
```

V úvodu kapitoly 3.2 bylo zmíněno několik speciálních datových typů, kterými TypeScript disponuje. Jsou jimi union typy, Type Guards, Intersection Types, řetěžcový typy a polymorfické *this* typy.

Union typy umožňují definovat seznam typů, kterých může proměnná nabývat. V následujícím příkladu je použit union typ v parametru funkce a v návratové hodnotě funkce. Výčet datových typů hodnoty, který může být předán funkci, je oddělený svislou čarou. Je možné definovat více než dvě možnosti datového typu a rovněž lze jako union typ použít i vlastní typ. Ukázková funkce přijímá pouze řetězec a číslo a stejně tak může vrátit jen řetězec nebo číslo. Pokud je do takové funkce poslána hodnota jiného datového typu, je při kompilaci vyhozena výjimka.

```
1  function convertGrade(grade: string | number) : string | number {
2    let letterGrades: string[] = ["A", "B", "C", "D", "F"];
3    let numberGrades: number[] = [1, 2, 3, 4, 5];
4    if (typeof grade === "string") {
5      return numberGrades[letterGrades.indexOf(grade)];
6    }
7    if (typeof grade === "number") {
8      return letterGrades[numberGrades.indexOf(grade)];
9    }
10 }
11 console.log(convertGrade("B")); // vypíše 2
12 console.log(convertGrade([1, 2, 3]));
13 // Argument of type 'number[]' is not assignable to parameter of type
   'string | number'.
```

Type Guard je výraz, který provádí kontrolu datového typu v určitém rozsahu za běhu programu. Slouží pro situace při používání union typů, kdy funkce může vracet návratovou hodnotu různých typů. V JavaScriptu se pro kontrolu typu za běhu

programu běžně používájí operátory *typeof* a *instanceof*. TypeScript ke zjednodušení používá type guard. K definici type guard je třeba definovat funkci, jejíž návratovým typem je typový predikát. Kdykoliv je tato funkce zavolána s nějakým parametrem, TypeScript omezí tento parametr na požadovaný typ, pokud je to možné.

V příkladu je funkce *getGrades()* schopná vrátit hodnotu typu *LetterGrade* nebo *NumberGrade*. Pro kontrolu typu hodnoty, který funkce *getGrades()* vrátí, slouží funkce *isLetterGrade()*, v jejíž hlavičce je definován predikát *grade is LetterGrade*. Tato funkce zjišťuje, zda je proměnná grade typu *LetterGrade* či nikoli, která se využije v rozhodovacím bloku podmínky. Díky této funkci je v jednotlivých větvích podmínky jasné, jakého typu proměnná *grades* je.

```
1   interface LetterGrade {
2     letterGrades: string[];
3   }
4
5   interface NumberGrade {
6     numberGrades: number[];
7   }
8
9   function getGrades(): LetterGrade | NumberGrade {
10    return {
11      letterGrades: ["A", "B", "C", "D", "F"],
12    };
13  }
14
15  function isLetterGrade(
16    grade: LetterGrade | NumberGrade): grade is LetterGrade
17  {
18    return (<LetterGrade>grade).letterGrades !== undefined;
19  }
20
21  let grades = getGrades();
22  if (isLetterGrade(grades)) {
23    console.log(grades.letterGrades);
24  } else {
25    console.log(grades.numberGrades);
26  }
```

Intersection type je datový typ, který říká, z jakých typů se má výsledný objekt skládat. Takových typů se nejčastěji dá využít s mixiny. Funkce *extend* přijímá dva různě typované parametry a jako návratovou hodnotu má intersection type (*T & U*). To znamená, že výsledkem této funkce má být objekt s vlastnostmi a chováním objektů předaných v parametrech.

```
1   function extend<T, U>(first: T, second: U): T & U {
2     let result = <T & U> {};
3     for (let id in first) {
4       result[id] = first[id];
5     }
6     for (let id in second) {
```

```
7       if (!result.hasOwnProperty(id)) {
8           result[id] = second[id];
9         }
10      }
11      return result;
12  }
13
14  class Logger {
15    log(logMessage: string): void {
16      console.log(logMessage);
17    }
18  }
19  class LetterGrade {
20    letterGrades: string[] = ["A", "B", "C", "D", "F"];
21  }
22  var gradesLogger = extend(new LetterGrade(), new Logger());
23  gradesLogger.log(gradesLogger.letterGrades.join(", "));
```

TypeScript rovněž umožňuje vytvářet alias pro datové typy. Alias je jiný název pro datový typ, který na daný typ odkazuje. Pro definici aliasu slouží klíčové slovo *type*.

```
1   interface LetterGrade {
2     letterGrades: string[];
3   }
4
5   interface NumberGrade {
6     numberGrades: number[];
7   }
8   type Integer = number;
9   type Grades = LetterGrade | NumberGrade;
```

Jak je vidět na ukázce, je možné vytvořit alias pro několik datových typů. Je možné definovat i alias, který používá generické typy, nebo alias, odkazující sám na sebe ve vlastnosti objektu.

Řetězcové typy specifikují, jakou hodnotu musí řetězec mít. Řetězcové typy jsou praktické zejména v kombinaci s union typy a aliasy. V ukázce je definován řetězcový typ *grades*, který stanovuje, že proměnná tohoto typu může mít hodnotu pouze „A“, „B“, „C“, „D“ nebo „F“. Při pokusu vložit do proměnné jiný řetězec kompilátor hlásí chybu.

```
1   type grades = "A" | "B" | "C" | "D" | "F";
2   function assignGrade(grade: grades) {
3     console.log(grade);
4   }
```

Polymorfický *this* typ lze uplatnit například při řetězení volání funkcí nad instancí objektu.

```
1   class GeometricObject {
2     protected color: string;
3     protected perimeter: number;
4
```

```
5    public setColor(color: string): this {
6      this.color = color;
7      return this;
8    }
9  }
10 class Square extends GeometricObject {
11   x: number;
12
13   public setX(x: number) {
14     this.x = x;
15     return this;
16   }
17
18   public calculatePerimeter() {
19     this.perimeter = 4 * this.x;
20     return this;
21   }
22 }
23 var square = new Square().setX(5).setColor("blue").calculatePerimeter();
```

Bez polymorfického typu *this* by následující kód vyhodil při volání metody *setColor()* chybu, protože *setX()* by vrátilo referenci na instanci třídy Square a ta metodu *setColor()* neobsahuje. TypeScript tento problém odstraňuje pomocí mechanismu odvozování typů, který proměnnou *this* převádí na datový typ zvaný *this*. [15]

3.2.3 Rozsah platnosti proměnných

Rozsah platnosti proměnných v TypeScriptu funguje stejně jako v JavaScriptu.

3.2.4 Klíčové slovo *this*

TypeScript pro správné fungování *this* používá tzv. lambda funkce, což jsou anonymní funkce schopné nastavit proměnnou *this* už při definici metody a ne až po jejím zavolání. [15]

```
1  class Student {
2    private firstname: string;
3    private lastname: string;
4    private birthday: Date;
5    constructor(firstname: string, lastname: string, birthday: Date) {
6      this.firstname = firstname;
7      this.lastname = lastname;
8      this.birthday = birthday;
9    }
10   getAge() {
11     return () => {
12       var currentDate: any = new Date();
13       var diff = currentDate - this.birthday;
14
15       return Math.floor(diff / (1000*60*60*24*365));
16       };
17   }
18   toString() {
19     return () => {
20       return this.firstname + ' ' + this.lastname + ', ' +
21              this.birthday;
```

```
22      }
23    }
24  }
25  function logger(toString) {
26      console.log(toString());
27  }
28  var s = new Student('Roman', 'Navratil',new Date(Date.UTC(1991, 7, 5)));
29  console.log(s.toString()());
30  // Roman Navratil, Mon Aug 05 1991 02:00:00 GMT+0200 (CEST)
31  logger(s.toString());
32  // Roman Navratil, Mon Aug 05 1991 02:00:00 GMT+0200 (CEST)
33  console.log(s.getAge()());
34  var studentToString = s.toString(); // 24
35  console.log(studentToString());
36  // Roman Navratil, Mon Aug 05 1991 02:00:00 GMT+0200 (CEST)
```

3.2.5 Třídy a dědičnost

TypeScript zpřehledňuje syntaxi tříd a doplňuje JavaScript o rozhraní, statické proměnné, statické metody a zjednodušuje dědičnost.

Zatímco JavaScript se soustředí na funkce a prototypovou dědičnost, TypeScript podporuje třídní dědičnost snáze uchopitelnou pro vývojáře. Pro definici třídy se používá klíčové slovo *class*. Konstruktor se definuje pomocí funkce *constructor*.

```
1   class Person {
2     firstname: string;
3     lastname: string;
4     birthday: Date;
5     constructor(firstname: string, lastname: string, birthday: Date) {
6       this.firstname = firstname;
7       this.lastname = lastname;
8       this.birthday = birthday;
9     }
10    toString(): string {
11      return this.firstname + ' ' + this.lastname + ', ' +
12             this.birthday;
13    }
14  }
```

Má-li jedna třída dědit od jiné, slouží k tomu klíčové slovo *extends*. Potomek vždy musí v konstruktoru volat konstruktor předka pomocí funkce *super()*. Klíčové slovo *super* pak odkazuje na předka třídy, přes které je možné volat metody předka. Potomek také může přepisovat metody předka a to tak, že jeho součástí je metoda se stejným názvem jako metoda předka.

```
1   class Person {
2     firstname: string;
3     lastname: string;
4     birthday: Date;
5     constructor(firstname: string, lastname: string, birthday: Date) {
6       this.firstname = firstname;
7       this.lastname = lastname;
8       this.birthday = birthday;
9     }
```

```
10    toString(): string {
11      return this.firstname + ' ' + this.lastname + ', ' +
12             this.birthday;
13    }
14  }
15  class Student extends Person {
16    studentId: number;
17    specialization: string;
18    university:string = 'UHK';
19    constructor(
20      studentId: number,
21      specialization: string,
22      firstname: string,
23      lastname: string,
24      birthday: Date
25    ) {
26      super(firstname, lastname, birthday);
27      this.studentId = studentId;
28      this.specialization = specialization;
29    }
30    toString(): string {
31      return JSON.stringify({
32          studentId: this.studentId,
33          name: this.firstname + ' ' + this.lastname,
34          birthday: this.birthday,
35          specialization: this.specialization
36        });
37    }
38  }
```

TypeScript také umožňuje definovat viditelnost třídních proměnných. Pokud není viditelnost definována, považuje se proměnná za veřejnou. Může být definovaná i jako *private*, kdy je proměnná viditelná pouze v rámci třídy, nebo *protected*, ke které mohou přistupovat i potomci třídy.

Jelikož při vývoji aplikace často nastává situace, kdy se v konstruktoru předávají jeho parametry do proměnných instance třídy, TypeScript umožňuje zkrácený zápis. Je-li před název parametru kontruktoru napsán modifikátor viditelnosti *private*, *protected* nebo *public*, je tím definována třídní proměnná s názvem parametru.

```
1   class Student {
2     constructor(
3       private studentId: number,
4       private specialization: string,
5       protected firstname: string,
6       private lastname: string,
7       private birthday: Date)
8     {}
9     toString(): string {
10      return JSON.stringify({
1         studentId: this.studentId,
2         name: this.firstname + ' ' + this.lastname,
3         birthday: this.birthday,
4         specialization: this.specialization
5       });
6     }
7   }
```

Dále je možné třídám definovat funkce pro získávání a nastavování atributů tzv. gettery a settery. K tomuto účelu slouží klíčová slova *get* a *set* udávaná před název metody. [15]

```
1  class Student {
2    constructor(
3      private _studentId: number,
4      private _specialization: string,
5      private _firstname: string,
6      private _lastname: string,
7      private _birthday: Date
8    ) {}
9
10   set firstname(firstname: string) {
11     this._firstname = firstname;
12   }
13
14   get firstname() {
15     return this._firstname;
16   }
17 }
18 var student = new Student(123, 'AI', 'Roman', 'Navratil', new Date);
19 student.firstname = 'Tomas';
20 student.firstname = true; // Chyba, protože setter přijímá pouze string
```

3.2.6 Statické atributy a metody

Kromě proměnných instance umožňuje TypeScript definovat proměnné a metody třídy neboli statické proměnné a metody pomocí klíčového slova *static*. [15]

```
1  class Student {
2    static university:string = "UHK";
3    constructor(
4      private _studentId: number,
5      private _specialization: string,
6      private _firstname: string,
7      private _lastname: string,
8      private _birthday: Date
9    ) {}
10 }
11 console.log(Student.university);
```

3.2.7 Rozhraní

Vyjma definice datových typů lze v TypeScriptu použít rozhraní společně s třídami, jak je zvykem například v jazyce Java. Rozhraní stanovuje, jaké vlastnosti a metody má mít třída, která toto rozhraní implementuje. [15]

```
1  interface PersonInterface {
2    firstname: string;
3    lastname: string;
4    birthday: Date;
5    toString(): string;
```

```
6   }
7   class Person implements PersonInterface {
8     firstname: string;
9     lastname: string;
10    birthday: Date;
11    constructor(firstname: string, lastname: string, birthday: Date) {
12      this.firstname = firstname;
13      this.lastname = lastname;
14      this.birthday = birthday;
15    }
16    getAge(): number {
17      var currentDate: any = new Date();
18      var diff = currentDate - this.birthday;
19      return Math.floor(diff / (1000*60*60*24*365));
20    }
21    toString(): string {
22      return this.firstname + ' ' + this.lastname + ', ' +
23             this.birthday;
24    }
25  }
```

3.2.8 Výjimky

Výjimky se v jazyce TypeScript zapisují a chovají stejně jako v JavaScriptu. Pro vytvoření vlastní výjimky je možné vytvořit potomka objektu Error. Pro kontrolu typu výjimky v bloku *catch* je vhodné definovat třídní proměnnou s uloženým názvem výjimky. [15]

```
1   class CustomException extends Error {
2     public name:string = 'CustomException';
3     constructor(message: string) {
4       super(message);
5     }
6   }
7   try {
8     throw new CustomException('Chyba');
9   } catch (e) {
10    console.log(e.name);
11  }
```

3.2.9 Abstraktní třída

TypeScript umožňuje definovat i abstraktní třídy z nichž nelze vytvořit instanci, ale mohou obsahovat implementované metody, které jiná třída může zdědit. Abstraktní třídy mohou obsahovat i abstraktní metody, které stejně jako rozhraní stanovují, jak se má metoda jmenovat, jaké jsou její parametry a typ návratové hodnoty. K definici abstraktní třídy i metody se používá klíčové slovo *abstract*. [15]

```
1   abstract class Person {
2     firstname: string;
3     lastname: string;
4     birthday: Date;
5     constructor(firstname: string, lastname: string, birthday: Date) {
6       this.firstname = firstname;
7       this.lastname = lastname;
```

```
8       this.birthday = birthday;
9     }
10    getAge(): number {
11      var currentDate: any = new Date();
12      var diff = currentDate - this.birthday;
13      return Math.floor(diff / (1000*60*60*24*365));
14    }
15    abstract toString(): string;
16  }
17  class Student extends Person {
18    private studentId: number;
19    private specialization: string;
20
21    constructor(
22      studentId: number, specialization: string, firstname: string,
23      lastname: string, birthday: Date
24    ) {
25      super(firstname, lastname, birthday);
26      this.studentId = studentId;
27      this.specialization = specialization;
28    }
29    toString(): string {
30      return JSON.stringify({
31        studentId: this.studentId,
32        name: this.firstname + ' ' + this.lastname,
33        birthday: this.birthday,
34        specialization: this.specialization
35      });
36    }
37  }
```

3.2.10 Mixin

TypeScript nedisponuje klíčový slovem pro označení mixinu. Mixin v TypeScriptu je třída, která je implementována jinou třídou stejně jako rozhraní pomocí klíčového slova *implements*. Implementující třída musí obsahovat stejné metody jako mixin. Jinak by kompilátor hlásil chybu. Nakonec je třeba implementovat funkci, která zkopíruje vlastnosti a chování mixinů do požadované třídy. [15]

```
1   class SpecializationChanger {
2     specialization: string;
3     changeToIM(): void {
4       this.specialization = 'Information Management';
5     }
6     changeToAI(): void {
7       this.specialization = 'Applied Informatics';
8     }
9   }
10
11  class Student implements SpecializationChanger {
12    firstname: string;
13    lastname: string;
14    specialization: string = 'Applied Informatics';
15
16    constructor(firstname: string, lastname: string) {
17      this.firstname = firstname;
18      this.lastname = lastname;
19    }
20    changeToIM: () => void;
21    changeToAI: () => void;
22  }
```

```
23
24 applyMixins(Student, [SpecializationChanger]);
25 var student = new Student('Roman', 'Navratil');
26 student.changeToIM();
27
28 function applyMixins(derivedCtor: any, baseCtors: any[]) {
29   baseCtors.forEach(baseCtor => {
30     Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
31       derivedCtor.prototype[name] = baseCtor.prototype[name];
32     });
33   });
34 }
```

3.2.11 Generické typy

V jazyce TypeScript lze definovat rozhraní, funkce a třídy s parametrickými typy. Popis, jak fungují generické typy, je možné nalézt v kapitole 2.12.

Následující ukázka kódu představuje použití parametrických typů v definici rozhraní a funkce. Proměnná *print* má definovaný datový typ rozhraní *Printer<T>*. Do této proměnné je tedy možné uložit funkci, která má tvar daný rozhraním.

```
1   interface Printer<T> {
2     (arguments: T): void;
3   }
4   function printToConsole<T>(arguments: T): void {
5     console.log(arguments);
6   }
7   var print: Printer<string> = printToConsole;
8   print('TypeScript');
```

Aby třída mohla používat generický typ, musí ve svém názvu ve špičatých závorkách uvést zástupný znak typu a definovat její metody a proměnné. Posléze je třeba vytvořit instanci této třídy s požadovaným typem a implementovat metody, které má mít. [15]

```
1   class Printer<T> {
2     printToConsole: (object: T) => void;
3   }
4   class Person {
5     constructor(private name: string) {};
6     getName() {
7       return this.name;
8     }
9   }
10 var p = new Printer<Person>();
11 p.printToConsole = function (person) {
12   console.log(person.getName())
13 };
14 p.printToConsole(new Person('Roman Navratil')); // v pořádku
15 p.printToConsole(1234); // chyba typu
```

3.2.12 Asynchronní zpracování

TypeScript nedisponuje žádnými funkcemi navíc pro asynchronní zpracování ani pro generátory. Je možné použít Promises stejně jako v JavaScriptu. Podmínkou pro použití Promises je nastavení kompilátoru na kompilaci do ECMAScriptu 6.

```
1   var promise: Promise<string> = new Promise<string>((resolve, reject) =>
    {
2     if (true) {
3       resolve();
4     } else {
5       reject();
6     }
7   });
8   promise.then(() => {
9     console.log('zpracovani vysledku');
10  }).catch(() => {
11    console.log('zpracovani chyby');
12  });
```

3.2.13 Metadata

Metadata se v TypeScriptu definují takzvanými dekorátory, což jsou speciální deklarace, které se připojují ke třídě, metodě, vlastnosti nebo parametru. Dekorátory se skládají ze dvou funkcí. Vnější funkce je tovární funkce na typ Decorator a slouží k předání parametrů anotace. Vnitřní funkce přijímá parametr *target*, ve které je uložena informace o entitě, které se anotace týká. Vytvořením takové funkce je možné definovat anotace s vlastními parametry. Název anotace je pak shodný s názvem vnitřní funkce.

```
1   function task(title: string, description: string) {
2     return function (target) {
3       console.log(target);
4       console.log(title);
5       console.log(description);
6     }
7   }
8
9   @task('Sleep', 'Go to sleep for at least 10 hours')
10  class Test {
11  }
```

Dekorátory jsou označeny jako experimentální funkce TypeScriptu a nedoporučuje se využití v produkčním prostředí. [15]

3.2.14 Reflexe

Dekorátory je možné použít i k úpravě konstruktorů tříd a třídních metod pomocí vlastních anotací s libovolnými parametry. Definici funkce pro zpracování anotace

je kromě vlastních parametrů předán *Decorator*, který se liší podle toho, zda se jedna o anotaci třídy nebo metody.

Třídní *Decorator* je definován před deklarací třídy. Do instance *Decorator* je předán konstruktor. Pokud dekorátor vrací hodnotu, je konstruktor třídy touto hodnotou nahrazen.

Dekorátor pro metody se udávají před deklarací metody a přijímají tři argumenty:

- Prototyp třídy
- Název metody, ke které se instance *Decorator* váže
- *Property Descriptor*

Pokud metoda dekorátoru vrací hodnotu, bude použita jako *Property Descriptor*.

Existují dva typy objektu *Property Descriptor*, datový a přístupový. Datový *Descriptor* uchovává hodnotu vlastnosti objektu. Přístupový *Descriptor* definuje getter a setter vlastnosti objektu.

V následující ukázce je příklad definice dekorátoru pro metodu *getName()*, která vrací hodnotu v proměnné *firstname*. Je-li tato metoda doplněna o anotaci *@fullname()*, je chování funkce *getName()* změněno tak, aby vracela jméno a příjmení osoby. [15]

```
1  class Person {
2    constructor(private firstname:string, public lastname:string) {};
3
4    @fullname()
5    getName() {return this.firstname;}
6
7    toString() {}
8  }
9
10 function fullname() {
11   return function (target: any, propertyKey: string, descriptor:
12     PropertyDescriptor) {
13     descriptor.value = function () {
14       return this.firstname + ' ' + this.lastname;
15     }
16   };
17 }
18 console.log((new Person('Roman', 'Navratil')).getName());
```

3.3 CoffeeScript

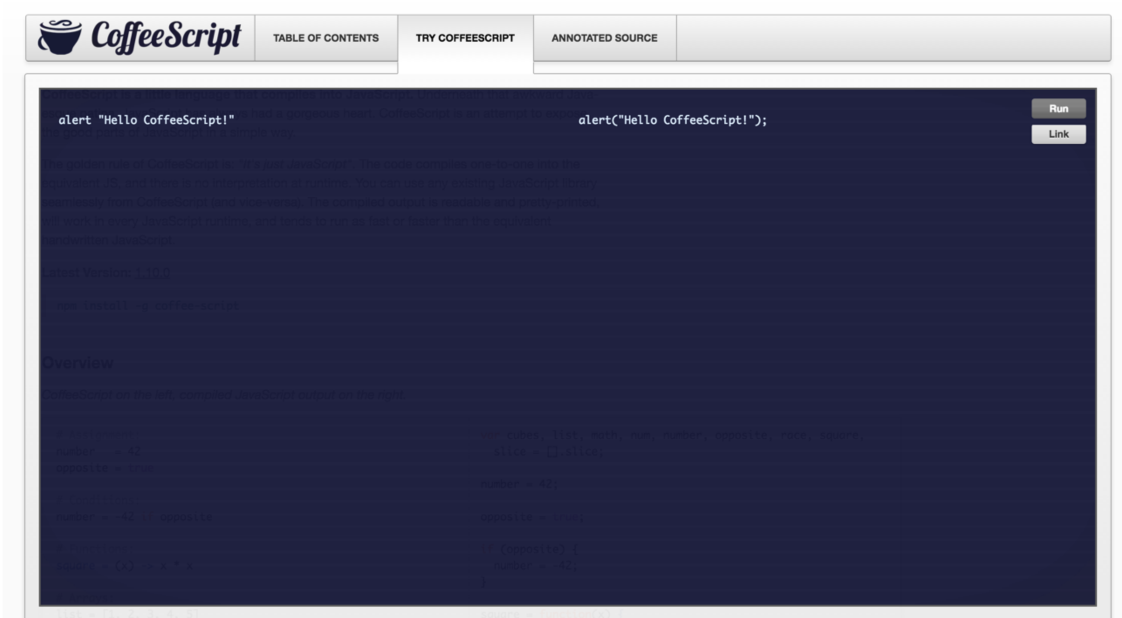
První verze CoffeeScriptu byla vydána 13. prosince 2009. Tato verze používala kompilátor do JavaScriptu napsaný v Ruby a přípona souborů byla *.jaa. [19] V době

psaní této práce se CoffeScript nacházel ve verzi 1.10.0. Přípona souborů je *.coffee a kompilátor je přepsaný do CoffeScriptu.

Autorem CoffeeScriptu je Jeremy Ashkenas autor, JavaScriptového frameworku Backbone.js a JavaScriptové knihovny Underscore.js. Pracoval v grafickém oddělení New York Times. [20]

3.3.1 Nástroje

Pro CoffeeScript neexistuje žádný oficiální nástroj na psaní kódu. Do většiny populárních IDE a textových editorů, ale existují doplňky, které přidávají podporu tomuto jazyku. Stejně jako ostatní jazyky i CoffeeScript si lze vyzkoušet v prohlížeči.



Obrázek 4 Nástroj pro zkoušku CoffeeScriptu v prohlížeči. Nalevo je kód napsaný v CoffeeScriptu a napravo je kód zkompilovaný do JavaScriptu

CoffeScript má jediný nástroj, který funguje pouze v příkazové řádce a slouží převážně na kompilaci do JavaScriptu a na spouštění CoffeeScript souborů.

```
Usage: coffee [options] path/to/script.coffee -- [args]

If called without options, `coffee` will run your script.

  -b, --bare         compile without a top-level function wrapper
  -c, --compile      compile to JavaScript and save as .js files
  -e, --eval         pass a string from the command line as input
  -h, --help         display this help message
  -i, --interactive  run an interactive CoffeeScript REPL
  -j, --join         concatenate the source CoffeeScript before compiling
  -m, --map          generate source map and save as .js.map files
  -n, --nodes        print out the parse tree that the parser produces
      --nodejs       pass options directly to the "node" binary
      --no-header    suppress the "Generated by" header
  -o, --output       set the output directory for compiled JavaScript
  -p, --print        print out the compiled JavaScript
  -r, --require      require the given module before eval or REPL
  -s, --stdio        listen for and compile scripts over stdio
  -l, --literate     treat stdio as literate style coffee-script
  -t, --tokens       print out the tokens that the lexer/rewriter produce
  -v, --version      display the version number
  -w, --watch        watch scripts for changes and rerun commands
```

Obrázek 5 Ukázka nástroje pro CoffeeScript pracují z příkazové řádky. Zdroj: vlastní zpracování

3.3.2 Základní syntaxe

Ze všech uvedených programovacích jazyků se tento nejvíce liší syntaxí. Syntaxe CoffeeScriptu se silně podobá jazykům Ruby nebo Python. K určování bloků kódu se používají neviditelné znaky. Místo znaku středníku k ukončení výrazu se používá odřádkování. Tělo podmínek nebo funkcí není ohraničené složenými závorkami, ale určuje jej odsazení kódu.

CoffeeScript nedisponuje žádnou statickou typovou kontrolou. Nikde se tedy nedefinuje datový typ proměnných, parametrů funkcí nebo návratových hodnot funkcí.

```
1   firstName = 'Roman'
2   lastName = 'Navratil'
3   age = 24
4   height = 189.5
5   months = ['leden', 'unor', 'brezen']
6
7   number = 6
8   if number % 2 == 0
9     console.log 'Cislo je sude'
10  else
11    console.log 'Cislo je liche'
```

Název funkce je definován jako proměnná přiřazená funkci. Nejprve se do jednoduchých závorek udají parametry funkce, pokud jsou potřeba, poté následuje

štíhlá šipka, která značí, že za ní je tělo funkce. Tělo funkce se musí začít psát na dalším řádku a musí být odsazené. Při volání funkce není nutné psát parametry funkce do závorek. Stačí za názvem funkce udělat mezeru a vypsat parametry oddělené čárkou.

```
1  selectQuery = (columns, table, order = 'ASC') ->
2    'SELECT ' + columns.join(', ') + ' FROM ' + table + ' ' + order
3
4  console.log selectQuery ['first_name', 'last_name', 'birthday'], 'users'
   // SELECT first_name, last_name, birthday FROM users ASC
```

Cyklus se v CoffeeScriptu dá napsat dvěma způsoby. První způsob je běžnější a má podobnou syntaxi jako foreach cyklus. Napřed je definována proměnná reprezentující každý element pole, a poté pole, které se má procházet. Druhý způsob používá takzvané comprehensions, které umožňují cyklus zapsat v jednom řádku. Comprehensions jsou výrazy a mohou být vráceny a přiřazeny do proměnné. Fungují jako filtry nad datovou sadou. Ukázka jak takový cyklus může vypadat je v níže uvedených příkladech, kde jsou comprehensions použity k filtraci lichých čísel.

```
1  months = ['leden', 'únor', 'březen', 'duben', 'květen', 'červen',
   'červenec', 'září', 'říjen', 'listopad', 'prosinec']
2  // for cyklus
3  // je-li potřeba znát index dopíše se za čárku za definici proměnné
   elementu
4  for month, i in months
5    console.log months[i]
6  // for cyklus používající comprehensions
7  console.log month for month in months
8  // for cyklus používající comprehensions s podmínkou a přiřazením do
   proměnné
9  numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
10 evenNumbers = (number for number in numbers when number % 2 == 1)
11 console.log evenNumbers
```

I while cyklus lze uložit do proměnné. Poslední výraz v těle while cyklu je uložen do pole a to je vráceno jako výsledek while cyklu. [21]

```
1  i = 0
2  // běžný zápis while cyklu
3  while i < months.length
4    console.log months[i]
5    i++
6  // zkácený zápis while cyklu, který je uložen do proměnné
7  i = 10
8  numbers = while i -= 1
9    i
10 console.log numbers
```

3.3.3 Rozsah platnosti proměnných

Rozsah platnosti proměnných v CoffeeScriptu funguje stejně jako v JavaScriptu.

3.3.4 Klíčové slovo *this*

Kapitola 3.1.4 ukazuje, jak funguje klíčové slovo *this* v JavaScriptu, které funguje jinak než je zvykem v jazycích jako je například Java. CoffeeScript toto chování zjednodušuje speciálním operátorem =>. Tímto operátorem se vytvoří funkce a zároveň je k této funkci připojeno *this* se správným kontextem.

Jedním z případů, kdy *this* je neočekávaným objektem v JavaScriptu je použití vnitřní funkce. V ukázce kódu je třída *Student* s metodou *toString()*. Dále je funkce *logger()*, v jejímž těle je spuštěna funkce a vypsán její výsledek, kterou funkce přijímá jako parametr. Jelikož je funkce *toString()* definovaná pomocí operátoru =>, proměnná *this* bude obsahovat správný kontext *Student* a výsledkem funkce *logger()* bude správný výpis atributů třídy *Student*. Byla-li funkce *toString()* vytvořena pomocí operátoru ->, vypsala by „undefined udefined, undefined", jelikož *this* by mělo nastavený kontext na globální objekt *window*. [21]

```
1  class Student
2    constructor: (@firstname, @lastname, @birthday) ->
3    toString: =>
4      @firstname + ' ' + @lastname + ', ' + @birthday
5
6  logger = (toString) ->
7    console.log toString()
8
9  s = new Student('Roman', 'Navratil', new Date(Date.UTC(1991, 7, 5)))
10 studentToString = s.toString
11 logger(s.toString)
12 # Roman Navratil, Mon Aug 05 1991 02:00:00 GMT+0200 (CEST)
```

3.3.5 Třídy a dědičnost

Pro definici třídy slouží klíčové slovo *class*. Kontruktor je definován funkcí *constructor*. Pokud je třeba přistoupit k třídní proměnné, není nutné psát klíčové slovo *this*, ale pouze přidat znak zavináče před název proměnné. Stejně jako v TypeScriptu není nezbydné zvlášť definovat vlastnosti třídy a poté jim přiřazovat hodnoty v konstruktoru. Pokud je před název parametru konstruktoru přidán znak zavináče, CoffeeScript sám vytvoří třídní vlastnosti pojmenované stejně jako parametry a předá jim hodnotu, kterou dostane v konstruktoru.

```
1  class Student
2    constructor: (@firstName, @lastName, @birthday) ->
3
4    toString: ->
5      JSON.stringify({
6        firstName: @firstName
7        lastName: @lastName
8        birthday: @birthday
9      })
```

Pro dědičnost CoffeeScript také používá klíčové slovo *extends*. Pro volání funkce předka slouží funkce *super()*. Není třeba udávat název volané funkce. Pokud je v metodě potomka zavolána funkce *super()*, je zavolána metoda předka se stejným názvem jako metoda, ze které je tato metoda volána.

V ukázce kódu je kromě dědičnosti tříd vidět i použití funkce *super()*, která ze třídy *Student* volá konstruktor třídy *Person*. Jestliže by byla tato funkce volána v metodě *toString()* třídy *Student*, byla by spuštěna metoda *toString()* třídy *Person*. [21]

```
1  class Person
2    constructor: (@firstname, @lastname, @birthday) ->
3
4    getAge: () ->
5      currentDate = new Date()
6      diff = currentDate - @birthday
7      Math.floor(diff / (1000*60*60*24*365))
8
9    toString: () ->
10     @firstname + ' ' + @lastname + ', ' + @birthday
11
12 class Student extends Person
13   constructor: (@studentId, @specialization, @firstname, @lastname,
   @birthday) ->
14     super(@firstname, @lastname, @birthday)
15
16   getSubjects: () ->
17     ['ZMI', 'UOMO', 'TNPW', 'PRO']
18
19   toString: () ->
20     JSON.stringify {
21       studentId: @studentId,
22       name: @firstname + ' ' + @lastname,
23       birthday: @birthday,
24       specialization: @specialization
25     }
26
```

3.3.6 Statické atributy a metody

CoffeeScript umožňuje definovat i statické proměnné a metody. Definice probíhá stejně jako u vlastností a metod tříd s tím rozdílem, že před názvem je uveden znak zavináče. Jelikož tento znak symbolizuje *this*, ve kterém je reference na aktuální objekt, tyto vlastnosti a metody jsou přiřazeny k objektu a lze je tedy volat bez tvorby instance objektu. [21]

```
1  class Student extends Person
2    @university: "UHK"
3    @getSubjects: () ->
4      ['ZMI', 'UOMO', 'TNPW', 'PRO']
5
6  console.log Student.university
7  console.log Student.getSubjects()
```

3.3.7 Rozhraní

Rozhraní CoffeeScript nepodporuje. Lze ale simulovat stejně jako v JavaScriptu, avšak se syntaxí CoffeeScriptu.

```
1  PersonInterface = ['getName']
2
3  isImplemented = (object, inter) ->
4    for method in inter
5      if ((typeof object[method]) == "function")
6        continue
7      else
8        return false
9    true
10
11 class Person
12   constructor: (@firstname, @lastname) ->
13     if (!isImplemented(@, PersonInterface))
14       throw new Error "Object does not implement interface"
15   getName: () ->
16     @firstname + ' ' + @lastname
17
18 person = new Person('Roman', 'Navratil')
19 console.log person.getName()
```

3.3.8 Výjimky

Zpracování chyb se provádí stejně jako v JavaScriptu, jen se syntaxí CoffeeScriptu.

```
1  class CustomException extends Error
2    constructor: (message) ->
3      @message = message
4      @name = 'CustomException';
5      @stack = (new Error()).stack
6
7  try
8    throw new CustomException('Error')
9  catch e
10   if e instanceof CustomException
11     console.log(e.name)
12   else
13     console.log(e.name)
```

3.3.9 Abstraktní třída

I abstraktních tříd lze dosáhnout pouze simulací jako v JavaScriptu.

```
1  class Person
2    constructor: ->
3      @firstname = undefined
4      @lastname = undefined
5
```

```
6       throw new Error "Abstract class cannot be instantiated"
7     getName: ->
8       @firstname + " " + @lastname
9
10  class Student extends Person
11    constructor: (@firstname, @lastname) ->
12
13  student = new Student('Roman', 'Navratil')
14  console.log student.getName()
15
16  person = new Person()
17  # Error: Abstract class cannot be instantiated
```

3.3.10 Mixin

Ani mixiny nejsou nativně podporované a lze jich dosáhnout funkcí, která zkopíruje vlastnosti jedné třídy do druhé stejně jako v JavaScriptu a TypeScriptu. Taková funkce by mohla vypadat následovně. [22]

```
1   extend = (destination, source) ->
2     destination[name] = method for name, method in source
3     destination
```

3.3.11 Generické typy

CoffeeScript nenabízí možnost používání generických typů.

3.3.12 Asynchronní zpracování

Pro asynchronní zpracování jsou k dispozici Promises jako v jazyce JavaScript.

```
1   promise = new Promise (resolve, reject) ->
2     if true
3       resolve()
4     else
5       reject()
6
7   promise.then (result) ->
8     console.log 'process result'
9   .catch (error) ->
10    console.log 'process error'
```

Generátory mají lehce zjednodušenou syntaxi. Není třeba definovat funkci klíčovým slovem *function**. V CoffeeScriptu je generátorem funkce, která ve svém těle používá klíčové slovo *yield*. [21]

```
1   oddNumbers = (n) ->
2     start = 0
3     while start < n
4       yield start += 2
```

3.3.13 Metadata

V CoffeeScriptu není podpora pro rozšíření kódu o doplňující informace.

3.3.14 Reflexe

CoffeeScript nedisponuje žádnými pomocnými objekty pro podporu reflexe, ale je možné získat informace o objektu stejným způsobem jako v JavaScriptu, jak popisuje kapitola 3.1.14.

```
1  class Person
2    constructor: (@firstname, @lastname, @birthday) ->
3    getAge: ->
4    toString: ->
5
6  person = new Person 'Roman', 'Navratil', new Date(1991, 7, 5)
7  console.log key + ' = ' + value for key, value of person
8
9  Object.defineProperty person, 'getFullName', {
10   value: () -> @firstname + ' ' + @lastname,
11   writable: true,
12   enumerable: true,
13   configurable: true
14 }
15 console.log person.getFullName()
```

4 Shrnutí výsledků

Všechny z probíraných jazyků jsou dobře zdokumentovány. K jednotlivým aspektům jazyka jsou kromě popisů i ukázky kódu s doplňujícími komentáři. TypeScript a Dart nabízí další články a ukázky kódu větších projektů.

Všechny alternativy k JavaScriptu zpřehledňují kód a mají dobré nástroje pro usnadnění vývoje. Platforma Dart jich ale nabízí nejvíc. Zatímco JavaScript, CoffeScript a TypeScript nabízí nástroje pro pohodlnější psaní kódu, Dart disponuje i vlastním virtuálním strojem, prohlížečem pro testování, debuggerem a generátorem dokumentace.

Dart vhodně řeší problémy s rozsahem proměnných nebo proměnnou *this*. CoffeeScript a TypeScript řeší pouze proměnnou *this* ale, i když se jim takový problém daří řešit, nejedná se o to tak elegantní řešení jako v případě Dartu, jelikož je vyžadováno použití speciální syntaktické konstrukce.

JavaScript sice podporuje objektové programování, ale je založen na prototypové dědičnosti, což řadě vývojářů dělá problém, jelikož není tak intuitivní. Dart má dědičnost založenou na třídách a objektové programování funguje tak, jak je většina programátorů zvyklá z jiných jazyků. CoffeeScript a TypeScript sice zavádí zápis tříd tak, aby vypadal běžněji, ale v principu jsou tyto třídy založeny na prototypové dědičnosti. JavaScript a CoffeeScript nepodporují abstraktní třídy, rozhraní ani mixiny. TypeScript je na tom lépe. Stejně jako Dart tyto vlastnosti podporuje.

Pro asynchronní zpracování má Dart Future a Stream API, které jsou hodně využívané v ostatních knihovnách Dartu. Pro přehlednější kód Dart podporuje klíčová slova *async* a *await*. JavaScript má pro zpracování na pozadí objekt Promise, který je sice podobný Future API Dartu, ale je podporovaný až od ECMAScript 6. TypeScript ani CoffeeScript v tomto ohledu nic navíc nepřinášejí.

Generátory nepodporuje pouze TypeScript. JavaScript je umožňuje použít, ale až od verze ECMAScriptu 6.

Reflexe je sice u všech probíraných jazyků podporovaná, ale v Dartu a TypeScriptu se jedná pouze o experimentální vlastnosti.

Zpracování chyb je v alternativách Dartu prakticky stejné. Dart oproti alternativám umožňuje definovat typ výjimky, která má být odchycena, a zpracovat ji.

Shrnutí jednotlivých aspektů jazyků shrnuje následující tabulka, která udává, zda daná vlastnost je v jazyce podporována.

	Dart	**JavaScript**	**TypeScript**	**CoffeeScript**
Datové typy	Ano	*Ne*	Ano	*Ne*
Třídy	Ano	Ano	Ano	Ano
Statické proměnné a metody	Ano	Ano	Ano	Ano
Rozhraní	Ano	*Ne*	Ano	*Ne*
Abstraktní třídy	Ano	*Ne*	Ano	*Ne*
Mixiny	Ano	*Ne*	Ano	*Ne*
Generické typy	Ano	*Ne*	Ano	*Ne*
Asynchronní zpracování	Ano	Ano	*Ne*	*Ne*
Generátory	Ano	Ano	*Ne*	Ano
Metadata	Ano	*Ne*	Ano	*Ne*
Reflexe	Ano	Ano	Ano	Ano

Tabulka 1 Přehled podpory různých vlastností jazyků

5 Závěr a doporučení

Pro programátora, který by přecházel z programovacího jazyku podporujícího třídní dědičnost, je jazyk Dart nejvhodnější volba. Vývojář tak může být plně odstíněn od prototypové dědičnosti JavaScriptu. Dart pro něj bude intuitivní, známý a jednoduchý, takže v něm dokáže tvořit aplikace rychle. Navíc nabízí spoustu nástrojů, které vývoj ještě více urychlí a usnadní. Na druhou stranu nemá tak silnou základnu vývojářů jako samotný JavaScript.

Další nevýhodou Dartu je jeho nejistá budoucnost. Google platformu Dart nijak zvlášť nepropaguje. Ke svým API nenabízí knihovny napsané v Dartu, nenabízí možnost použití Dart serveru v cloudu Googlu a ani podporu knihovny Polymer. Polymer je knihovna pro vývoj webových komponent, která je téměř rok ve stabilní verzi 1.0, je stále ve fází beta. Vývojáři tak na internetu poměrně často řeší, zda se vyplatí platformu studovat, a objevují se obavy ze zastavení vývoje Dartu.[19] Problémem je i podpora balíčků třetích stran. Často jsou podporovány pouze několik měsíců a pak se jejich vývoj zastaví.

Obavy se tým Dartu snaží mírnit na sociálních sítích, kde vytrvale tvrdí, že společnost Google na Dart spoléhá, což potvrzuje fakt, že nová verze služby AdWords, která je přepsaná do jazyka Dart.

Zajímavou alternativou je jazyk TypeScript, doplňující JavaScript. Přináší do JavaScriptu typovou kontrolu, což znamená lepší čitelnost kódu. Společně s dalšími dodanými novými vlastnostmi se mu podařilo docílit zpřehlednění kódu a odstranění základních nedostatků JavaScriptu a tím pádem i zrychlení vývoje.

CoffeeScript je nejvhodnější pro vývojáře znalého jazyka Ruby nebo Pythonu, neboť syntaxe bude povědomá. CoffeeScript dokáže překvapit minimalistickou syntaxí. Programátor, který přichází z jazyků jako je Java, PHP nebo C++, však může mít problémy se studiem tohoto jazyka z důvodu odlišnosti syntaxe a potrvá mu alespoň ze začátku déle něco vytvořit.

---

[19] https://plus.google.com/+TomasZverina/posts/DCkdbejFjzt

JavaScriptu nahrává fakt, že má obrovskou uživatelskou základnu. Existuje tak pro něj neuvěřitelné množství balíčků, knihoven a frameworků, které řeší některé z problémů jazyka.

Pokud se programátor chce vyhnout JavaScriptu, všechny probírané alternativy jsou dobré volby. Přichází-li vývojář s Ruby nebo Pythonu, CoffeeScript pro něj bude nejlepší volba. Pokud by byl znalý JavaScriptu a chtěl si zrychlit a zpřehlednit vývoj, sáhl by po TypeScriptu. V případě, že by chtěl začít psát robustní aplikaci s dobrou podporou objektově orientovaného programování a intuitivní syntaxí, Dart je jednoznačná volba.

6 Seznam použité literatury

[1] WALRATH, Seth LADD. *What is Dart?*. Sebastopol (Kalifornie, Spojené státy americké): O'Reilly Media, Inc., 2012. ISBN 978-1-449-33232-7.

[2] Dart: A language for structured web programming. BAK, Lars. *Chromium Blog* [online]. 2011 [cit. 2014-07-01]. Dostupné z: http://blog.chromium.org/2011/10/dart-language-for-structured.html

[3] Google's Chrome: The Danish Magic Inside. Bloomberg Business [online]. [cit. 2016-01-31]. Dostupné z: http://www.bloomberg.com/bw/stories/2008-11-12/googles-chrome-the-danish-magic-insidebusinessweek-business-news-stock-market-and-financial-advice

[4] Kasper Verdich Lund [online]. [cit. 2016-01-31]. Dostupné z: http://verdich.dk/kasper/

[5] Quick Start. Dart: Scalable, productive app development [online]. [cit. 2016-01-31]. Dostupné z: https://www.dartlang.org/docs/dart-up-and-running/ch01.html

[6] *A Tour of the Dart Language.* [online]. [cit. 2016-01-31]. Dostupné z: https://www.dartlang.org/docs/dart-up-and-running/ch02.html

[7] *Dart Tools* [online]. [cit. 2016-04-28]. Dostupné z: https://www.dartlang.org/tools/

[8] The present and future of editors and IDEs for Dart. Dart News & Updates [online]. 2015 [cit. 2016-01-31]. Dostupné z: http://news.dartlang.org/2015/04/the-present-and-future-of-editors-and.htm

[9] A Short History of JavaScript. W3C [online]. [cit. 2016-01-31]. Dostupné z: https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

[10] Who is Brendan Eich? The Telegraph [online]. 2014 [cit. 2016-01-31]. Dostupné z: http://www.telegraph.co.uk/technology/news/10744011/Who-is-Brendan-Eich.htm

[11] Brendan Eich Steps Down as Mozilla CEO. The Mozilla Blog: News, notes and ramblings from the Mozilla project [online]. 2014 [cit. 2016-01-31]. Dostupné z: https://blog.mozilla.org/blog/2014/04/03/brendan-eich-steps-down-as-mozilla-ceo/

[12] Brave. Brave [online]. [cit. 2016-01-31]. Dostupné z: https://www.brave.com/

[13] Nodejs. Nodejs [online]. [cit. 2016-01-31]. Dostupné z: https://nodejs.org/en

[14] *JavaScript* [online]. [cit. 2016-04-28]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[15] TypeScript. TypeScript [online]. [cit. 2016-01-31]. Dostupné z: http://www.typescriptlang.org/

[16] CopyrightNotice. GitHub [online]. [cit. 2016-01-31]. Dostupné z: https://github.com/Microsoft/TypeScript/blob/master/CopyrightNotice.txt

[17] Announcing TypeScript 0.8.1. TypeScript [online]. [cit. 2016-01-31]. Dostupné z: http://blogs.msdn.com/b/typescript/archive/2012/11/15/announcing-typescript-0-8-1.aspx

[18] Announcing TypeScript 1.0. TypeScript [online]. [cit. 2016-01-31]. Dostupné z: https://blogs.msdn.microsoft.com/typescript/2014/04/02/announcing-typescript-1-0/

[19] Initial commit of the mystery language. GitHub [online]. [cit. 2016-01-31]. Dostupné z: https://github.com/jashkenas/coffeescript/commit/8e9d637985d2dc9b44922076ad54ffef7fa8e9c2

[20] Jereme Ashkenas. Twitter [online]. [cit. 2016-01-31]. Dostupné z: https://twitter.com/jashkenas

[21] *CoffeeScript* [online]. [cit. 2016-04-28]. Dostupné z: http://coffeescript.org/

[22] A fresh look at JavaScript Mixins. JavaScript, JavaScript… [online]. 2011 [cit. 2016-01-31]. Dostupné z: https://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/

[23] Dart on the Server. *Dart: Structured web apps* [online]. [cit. 2016-04-03]. Dostupné z: https://dart-lang.github.io/server/server.html

[24] Chrome V8. *Google Developers* [online]. 2012 [cit. 2016-04-04]. Dostupné z: https://developers.google.com/v8/intro#about-v8

[25] Connected Limited Device Configuration (CLDC); JSR 139. *Oracle* [online]. [cit. 2016-04-04]. Dostupné z: http://www.oracle.com/technetwork/java/cldc-141990.html

[26] Dart for the Entire Web. *Dart News & Updates* [online]. 2015 [cit. 2016-04-06]. Dostupné z: http://news.dartlang.org/2015/03/dart-for-entire-web.html

[27] Classes. *The Little Book on CoffeeScript* [online]. [cit. 2016-04-20]. Dostupné z: https://arcturo.github.io/library/coffeescript/03_classes.html

[28] Everything you wanted to know about JavaScript scope. *Todd Motto: JavaScript and Angular articles* [online]. 2013 [cit. 2016-04-29]. Dostupné z: https://toddmotto.com/everything-you-wanted-to-know-about-javascript-scope/

7 Seznam obrázků

Obrázek 1 Ukázka online editoru DartPad. Zdroj: vlastní zpracování 5
Obrázek 2 Grafické rozhraní nástroje JsFiddle. Zdroj: vlastní zpracování 28
Obrázek 3 Online nástroj na vyzkoušení TypeScript bez nutnosti instalace. Zdroj: vlastní zpracování 39
Obrázek 4 Nástroj pro zkoušku CoffeeScriptu v prohlížeči. Nalevo je kód napsaný v CoffeeScriptu a napravo je kód zkompilovaný do JavaScriptu 54
Obrázek 5 Ukázka nástroje pro CoffeeScript pracují z příkazové řádky. Zdroj: vlastní zpracování..55

8 Seznam tabulek

Tabulka 1 Přehled podpory různých vlastností jazyků...63

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Rokitanského 62, 500 03 Hradec Králové, tel: 493 331 111, fax: 493 332 235

Zadání k závěrečné práci

Jméno a příjmení studenta: **Roman Navrátil**

Obor studia: Aplikovaná informatika
Jméno a příjmení vedoucího práce: **Pavel Janečka**

Název práce:
Jazyk Dart

Název práce v AJ:
Programming language Dart

Podtitul práce:

Podtitul práce v AJ:

Cíl práce: Představení vlastností jazyka a vývojové platformy Dart a jeho následné porovnání s alternativami v prostředí webu.

Osnova práce:
Historie jazyka Dart a motivace jeho vzniku
Vlastnosti jazyka Dart
Vývojová platforma jazyka Dart
Implementace vybraných úloh prezentujících vlastností jazyka
Porovnání s alternativami na platformě webu

Projednáno dne: 15. 10. 2014

Podpis studenta

Podpis vedoucího práce