Obsah

1 Předmluva

Kniha, kterou právě čtete, je úvodem do světa tvorby aplikací. Nečekejte však suchý popis programovacího jazyka. Čtenář se na jednoduchých příkladech naučí, jak při tvorbě programů přemýšlet. Pokud vás v této souvislosti straší slovo „algoritmizace“, kniha vás nezklame. Neobsahuje partie z teoretické informatiky, namísto toho je vše vysvětleno formou příkladů pochopitelných i pro naprosté nováčky ve světě tvorby software.

Tato publikace však návodem, jak při programování přemýšlet, nekončí. Čtenář se dozví, jak používat jazyk Java, jeho pravidla, a v neposlední řadě knihovny, které s tvorbou software pomohou. Tento jazyk je již dlouho osvědčenou volbou pro výuku základů programování, jeho znalost zároveň uplatníte i v praxi. Ani jazykem Java záběr knihy nekončí. V knize se dozvíte o základních technologiích webu a naučíte se tvořit webové aplikace.

Text zabíhá i mimo téma základů nutných pro vytvoření fungujícího programu. Dozvíte se, jak zdrojový kód organizovat, aby byl přehledný a pochopitelný, nebo proč a jak zdrojový kód obohatit o dokumentaci. Jistě oceníte popis aplikací, které s programováním pomohou, například editoru – vývojového prostředí. Seznámíte se i s dlouhodobě osvědčenou a stále používanou technologií, díky které je možné své softwarové projekty efektivně spravovat nebo sdílet s dalšími vývojáři. Kromě příkladů částí programu v textu je součástí knihy i hotová aplikace spolu s návodem na její zprovoznění. Vedle možnosti prohlédnutí všech detailů je tak možné se učit pomocí zkoušení si vlastních úprav.

Přeji příjemné vykročení do oblasti programování při čtení této knihy.

Ing. Ondřej Guth, Ph.D.

4 Základní pojmy

> *Pokud chcete problém vyřešit, musíte jej nejdříve pojmenovat, a řešit příčinu, ne příznaky.*
>
> — Otto Wichterle, český vědec a vynálezce

Většina problémů, se kterými se (nejen) začínající programátoři setkávají, má své řešení někde na internetu, stačí ho najít. Klíčem k nalezení řešení bývá správná formulace vyhledávacího dotazu, ale ten se bez správných pojmů sestavuje jen obtížně.

4.1 Data

V oblasti výpočetní techniky jsou *data* libovolné informace vhodné k počítačovému zpracování. Nejmenší jednotkou dat je *bit*, který obsahuje pouze dva stavy: (**ano**/**ne**), nebo (**pravda**/**nepravda**), nebo třeba (**přítomen**/**nepřítomen**), interpretace mohou být různé. Protože stav lze vyjádřit také pouhou číslicí (`1`/`0`, anglicky *digit*), nazývají se různá zařízení (stroje, přístroje), která interně umí pracovat s čísly, občas výrazem *digitální zařízení* (číslicové stroje, přístroje). Skládáním *bitů* do větších celků vznikají datové útvary, jejichž hodnota opět závisí na interpretaci. Pro vyjádření větší velikosti datového objemu se běžně používá jednotka *bajt* (anglicky *byte*) obsahující 8 *bitů*. U těchto jednotek můžeme použít předpony soustavy SI (*Mezinárodní systém jednotek*) podle vzoru:

- *kilobajt* – označuje tisíc *bajtů*, zkratka je *kB*,
- *megabajt* – označuje milion *bajtů*, zkratka je *MB*,
- *gigabajt* – označuje miliardu *bajtů*, zkratka je *GB*.

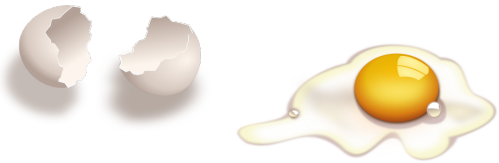
Předpony velkých čísel využijeme třeba při kontrole volného místa na pevném disku.

4.2 Algoritmus

Výklad pojmu *algoritmus* začneme jednoduchou definicí:

> *Algoritmus je návod či postup, kterým lze vyřešit daný typ úlohy. Přepis algoritmu do programovacího jazyka se nazývá programování.*
>
> — inspirováno zdrojem [10]

Jistou podobnost k *algoritmu* lze najít třeba v kuchařském receptu na přípravu pokrmu. Vezměme si například přípravu míchaných vajec pro dvě osoby.



Obrázek 2. Ilustrativní obrázek, upravený zdroj [9]

Postup přípravy lze rozepsat do následujících kroků:

1. Připravíme si suroviny: 4 vejce, šunku, olej, sůl.
2. Na pánvi ohřejeme olej na 160 °C.
3. Přidáme šunku nakrájenou na kostičky.
4. Přidáme vejce.
5. Mícháme po dobu 3 minut.
6. Pokrm podáváme na talíři teplý a osolený.

Tento recept jistě není zcela přesný, ale zkušená kuchařka ví, že vejce je třeba zbavit nejdříve skořápek a sporák je nutné nakonec vypnout; stroj to však neví. Z příkladu je také zřejmé, že pro srozumitelný popis přípravy pokrmu se neobejdeme bez jisté míry abstrakce. Hned první krok by se dal rozepsat samostatným algoritmem:

1. Vyhledáme v lednici potraviny na přípravu míchaných vajec.
2. Pokud některá z potravin chybí (kontrola obsahu lednice), pošleme člena domácnosti na nákup chybějících surovin.
3. Požadované suroviny odneseme na kuchyňskou linku.

Také **proces nakupování** by se dal popsat samostatným algoritmem a tímto způsobem bychom mohli pokračovat směrem ke stále podrobnějším detailům. Pojem *kroky algoritmu* lze chápat jako *příkazy*, které v jazyce Java realizujeme voláním metod. Každý příkaz přebírá odpovědnost za nějakou dílčí část algoritmu. Na našem příkladu vidíme, že součástí algoritmu mohou být také *příkazy* zpracované za určité podmínky (pokud nějaká potravina chybí, *běž nakoupit*).

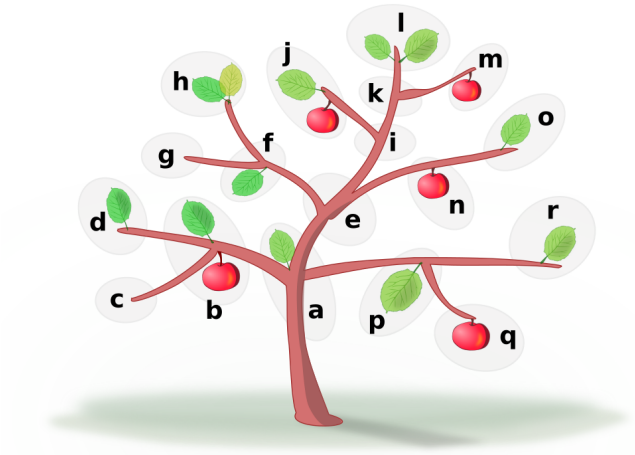
Pokud pošleme člena rodiny na nákup, s dalším postupem přípravy pokrmu pokračujeme až po jeho návratu. V běžném životě však zpravidla nečekáme, až se někdo vrátí, ale třeba nachystáme jídelnu. Čtyřjádrový procesor si můžeme představit jako rodinu se čtyřmi členy a zpracování algoritmů několika jádry

#	Kód	Výsl.	Typ	Poznámka
13.	`Math.max(2, 3)`	3	int	Výraz funguje jako funkce, která vrátí větší ze dvou hodnot uvedených v kulatých závorkách. Výrazům tohoto typu se budeme věnovat podrobněji dále.
14.	`2 + 3 == 6 - 1`	true	boolean	Výraz porovnává výsledek dvou matematických operací, výsledkem je logická hodnota. Aplikování matematického operátoru má *přednost* před operátorem porovnání, proto zde nejsou třeba závorky.
15.	`2 + 3 <= 6 - 2`	false	boolean	Výraz pro porovnání dvou číselných výrazů.
16.	`true || false`	true	boolean	Logický součet se vyznačuje párem svislých znaků (anglicky *pipe*).
17.	`true && false`	false	boolean	Logický součin se vyznačuje párem znaků ampersand (anglicky *ampersand*).
18.	`! false`	true	boolean	Příklad negace logického výrazu

Výše uvedené příklady uvádějí jen vybraný seznam operátorů a jejich možných kombinací, další příklady najdeme v kapitole Programové výrazy a podrobnější výklad tématu třeba zde [22]. Výhodou primitivních typů je nízká paměťová náročnost a podpora `operátorů` (výrazů pro matematické a logické operace).

4.6 Stromová struktura dat

Data lze seskupovat do různých struktur, které mohou urychlit vyhledávání dat, nebo zjednodušit manipulaci. Se **stromovou** strukturou se budeme setkávat poměrně často, hodí se tedy uvést včas základní pojmy.



Obrázek 3. Schéma stromu

Pro lepší představu o tomto datovém modelu si vezměme na pomoc strom jabloně s jejími listy a plody. Místo, kde se větve dělí, se nazývá *uzel* (anglicky *node*), pojmem *uzel* se však označuje obecně každé místo připojující další potomky (na obrázku jabloně to jsou: další větve, plody či listy stromu).

Koncový *uzel*, který **nepřipojuje** další potomky, se označuje termínem *list* (anglicky *leaf*). Každý *uzel* pak má jednoho svého *rodiče* – s výjimkou prvního *uzlu* na kmeni stromu, kterému se říká *kořen* (anglicky *root*).

Každému *uzlu* lze přiřadit *název* a pojmenovat můžeme i všechny *listy*. Pojem *list* se někdy nahrazuje výrazem *koncový uzel* a spojnice mezi *uzly* se nazývá *hrana*. Pokud potřebujeme vyjádřit přesnou adresu určitého *uzlu*, stačí nám zapsat postupně všechny názvy *uzlů* směrem od kořene stromu, oddělovat je můžeme třeba tečkou.

```
a.e.f.h.žlutý_list
```

Podobně bychom mohli lokalizovat i červené jablko vpravo nahoře.

```
a.e.i.k.m.jablko
```

Všimněte si, že *stromovou strukturu* vykazuje také adresa na poštovní obálce, kde kořenem stromu může být jméno státu a oddělovačem *uzlů* je *nový řádek*. Dalším příkladem může být *hierarchická klasifikace organismů* (s dělením na říše, kmeny, třídy a další úrovně) a podobných příkladů by se našla jistě celá řada.

4.7 Model reálného světa

Vysvětlení pojmu *model* uvedeme krátkou definicí:

> *Model je zjednodušená reprezentace určitého objektu reálného světa či systému – pojatá z určitého úhlu pohledu.*
>
> — inspirováno zdrojem [10]

Podívejme se do světa malých dětí, některé si hrávají s modely autíček, jiné s panenkami. Oba typy hraček lze považovat za zjednodušenou napodobeninu předmětu reálného světa. Při výrobě hraček se často preferuje zachování tvaru předlohy, barev, někdy zvuků. Jiné vlastnosti jsou naopak potlačeny v zájmu bezpečí uživatele nebo finanční dostupnosti.

také objekty vytvořené operátorem `new`, pouze je vytváří kompilátor automaticky. Pro úplnost doplňme, že pro zvýšení rychlosti běhu programu kompilátor může jednou vytvořené objekty poskytovat opakovaně. Praktické použití si ukážeme na několika jednoduchých příkladech.

Zdrojový kód 8. Proměna primitivních typů na objekt a zpět

```
Integer objCount = 5; ❶
Character objChar = 'A'; ❷
Boolean objAgreement = true; ❸
int count = objCount; ❹
```

❶ Konverze z primitivního typu na objektový využitím vlastnosti *autoboxing*.

❷ Vytvoření objektu, který reprezentuje znak tabulky Unicode.

❸ Vytvoření objektu, který reprezentuje logickou hodnotu.

❹ Ukázka zápisu objektového typu do primitivního.

Mějme na paměti, že často prováděný *autoboxing* primitivních typů na objekty může běh programu mírně zpomalit.

4.11.4 Neplatné objekty null

Vraťme se na chvíli ještě k programovým proměnným. Připomeňme si, že vložením nového obsahu do proměnné ztratíme automaticky původní obsah. Položme si však legitimní otázku: můžeme obsah proměnné vyprázdnit tak, aby neobsahoval žádnou hodnotu? Správná odpověď závisí na **typu** proměnné:

1. V případě *primitivního* typu obsah obecně vyprázdnit nelze, je možné ho pouze přepisovat jinými platnými hodnotami. Pro úplnost uvádím, že typy s plovoucí čárkou mohou využívat speciální **konstantu** `NaN` (z anglického *not a number*), kterou poskytuje korespondující třída.
2. V případě *objektových* typů lze obsah proměnné obecně vyprázdnit zápisem výrazu `null`, což je klíčové slovo jazyka Java.

Zdrojový kód 9. Příklady zápisu výrazu `null`

```
Integer objCount = null; ❶
Character objChar = null;
Boolean objAgreement = null;
String text = null;
int count = objCount; ❷
double value = Double.NaN; ❸
```

❶ Ukázka vytvoření celočíselné objektové proměnné s prázdným obsahem.

Nedefinovanou hodnotu lze obecně nastavit libovolné objektové proměnné.

❷ Pozor, neplatný objekt neposkytuje žádné objektové metody a nelze ho ani převést na primitivní číslo. Takovou událost však běžný Java kompilátor odhalit nedokáže, a tak za běhu programu by zde došlo k předčasnému ukončení algoritmu.

❸ Primitivní typy s plovoucí desetinnou čárkou mohou využívat speciální konstantu. Za tímto účelem se však využívají častěji *objektové* typy. K neplatné hodnotě mohou vést také některé matematické operace, jako třeba nula dělená nulou.

Jedním z důležitých rozdílů mezi **primitivními** datovými typy a jejich **objektovým** ekvivalentem je schopnost vyjádřit nedefinovanou hodnotu pomocí výrazu `null`, který je pro všechny objektové typy společný.

4.11.5 Vztahy mezi třídami

Pojďme dál a představme si, že máme čtyři instance *třídy* `Car` naplněné daty. Pokud atributy každého objektu vypíšeme do jednoho řádku tabulky, získáme následující přehled:

Tabulka 3. Přehled fiktivních vozidel

Výrobce	Model vozu	Vyrobeno	Výrobní číslo vozu	Pohon	Velikost kufru v litrech	Vlastník
ŠKODA AUTO	Fabia	2005-06-01	123-654-987	benzin	260	Jiří Mexedolid
Škoda Auto	Rapid	2013-01-15	359-985-785	benzin	550	Lucie Zixerfxodf
Toyota	Auris	2010-10-22	987-654-123	Benzin	354	Donald Choresisdi
Ford	Focus	2015-03-30	958-232-497	benzin	375	Donald Choresisdi

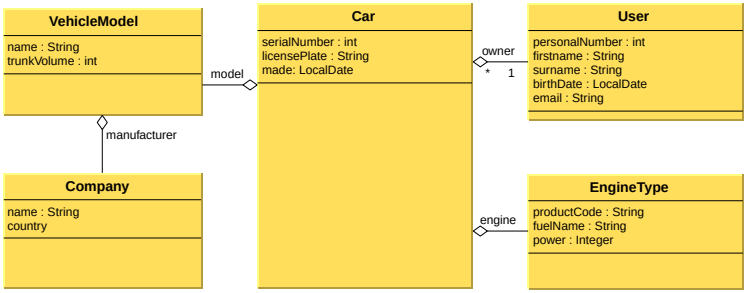
V uvedené tabulce lze postřehnout rizika, která nám mohou zkomplikovat pozdější vyhledávání vozidel…

1. Nelze vyloučit, že jeden **výrobce** vozu bude v různých řádcích zapsán různými názvy, tento problém se týká i **modelu**, **paliva** či jména **majitele**. Tohle může dělat problémy při vyhledávání vozidel podle názvu výrobce.
2. Neumíme rozlišit dva různé majitele se stejným jménem a příjmením.

Můžeme sice namítnout, že odpovědnost za překlepy nese zapisovatel u klávesnice, ale problém naší datové nekonzistence to neřeší. Abychom takovým problémům předešli, rozšíříme *datový model* o další třídy.

1. Začneme třeba **vlastníkem** vozu a pro něho navrhneme novou třídu `User` (česky *uživatel*). Třída dostane jedinečný identifikátor vlastníka (celočíselný atribut `personalNumber`) pro případ duplicitních jmen osob. V diagramu pak vyznačíme vztah spojovací čárou, která bude na straně *třídy* `Car` označena kosočtvercem, což je možné interpretovat slovy: objekt auta **má** vlastníka. Na konci **vazby** (vyznačené čárou) lze vyjádřit **počet** (přesněji multiplicitu) relačních objektů. Pevný počet se uvádí číslem, hvězdička vyjadřuje libovolný počet (včetně nuly), a lze vyjádřit také interval. Doplněný vztah tedy vypovídá, že jedno auto má **právě jednoho** vlastníka, ale jeden vlastník může mít více aut, nebo také žádné auto. Chybějící multiplicita sice formálně nevyjadřuje žádné **výchozí** hodnoty, v této knize to však lze považovat za ekvivalent právě zmíněného vztahu. *Relace* (nebo také vazba či vztah, anglicky *relation*) se zapisuje ve zdrojovém kódu jako běžný *atribut* objektu. Pokud ho však v diagramu zakreslíme jako vztah, už se mezi běžnými *atributy* neuvádí, nemá to význam. *Relace* zakreslená v diagramu může mít svůj název, který se uvádí u spojovací čáry. Pokud název *relace* v diagramu chybí, lze za něj považovat jméno odkazované *třídy* (s malým písmenem na začátku).
2. Podobně vyřešíme také model vozidla (`VehicleModel`) a označíme ho relací, která popisuje, že každý automobil má svůj model auta, ale jeden model auta může být označen u více vozidel. V tomto okamžiku už může být zřejmé, že výrobce automobilů je spíše vlastností továrního modelu než konkrétního automobilu, a proto ten *atribut* přesuneme do odpovídající třídy.
3. Nakonec doplníme třídu `Company` a znázorníme její relaci k modelu auta (auto **má** výrobce).

Po zakreslení všech připomínek do původního diagramu může výsledek vypadat takto:



Obrázek 5. Diagram tříd

Podle diagramu připravíme nové *třídy* a upravíme také tu původní metodu pro vytvoření objektu jednoho auta. Podívejme se na výsledek…

Zdrojový kód 10. Komplexní sestavení objektu

```
public Car createCar() {

    Car car = new Car();
    car.setMade(LocalDate.of(2010, 10, 22)); ❶

    VehicleModel model = new VehicleModel();
    model.setManufacturer(new Company("Toyota", Locale.JAPAN)); ❷
    model.setName("Yaris");
    model.setTrunkVolume(436);
    car.setModel(model); ❸

    User user = new User();
    user.setPersonalNumber(10);
    user.setFirstname("Donald");
    user.setSurname("Choresisdi");
    user.setBirthDate(LocalDate.of(1990, Month.OCTOBER, 30)); ❹
    car.setOwner(user); ❺

    return car;
}
```

❶ Vytvořit objekt pro lokální datum pomocí číselných parametrů už umíme.

❷ Ukázka použití konstruktoru se dvěma parametry, kde výraz `JAPAN` je statický *atribut třídy* `Locale` ze standardní knihovny *Java*. Novou instanci *třídy* není třeba zapisovat vždy do proměnné, ale je možné ji rovnou poslat do argumentu metody, jak ukazuje tento řádek.

❸ Instanci **modelu vozu** zapíšeme do objektu typu `Car`.

❹ Objekt pro lokální datum lze vytvořit i alternativní metodou, kterou předáváme statickou konstantu reprezentující konkrétní měsíc v roce. Konstanta měsíce je opět součástí standardní knihovny *Java*.

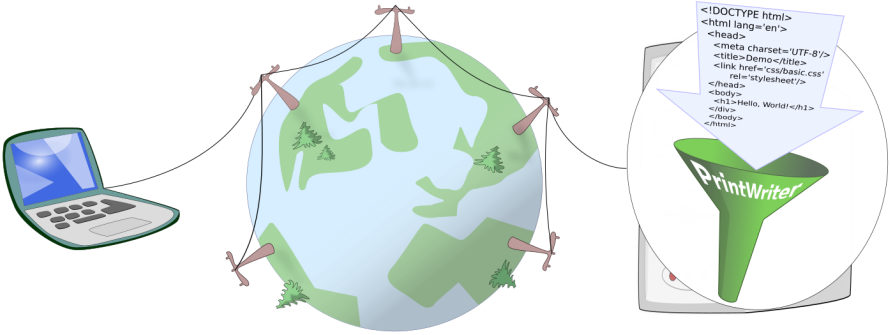
❺ Instanci objektu `User` zapíšeme do objektu typu `Car` jako vlastníka vozu.

Zdrojový kód 11. Řetězení metod

```
    public void printCar() {
    Car car = this.createCar(); ❶

    LocalDate made = car.getMade(); ❷
    String manufacturer = car.getModel().getManufacturer().getName(); ❸
    String modelName = car.getModel().getName(); ❹
    Integer trunkVolume = car.getTrunkVolume();
    String firstname = car.getOwner().getFirstname();
    String surname = car.getOwner().getSurname();
    String owner = firstname + " " + surname; ❺
```

programových objektů, protože potřebné služby zajistí *Java* knihovny, operační systém, síťová karta našeho počítače a další sofistikované technologie na cestě ke vzdálenému počítači. Při zkoušení našich příkladů nám služby vzdáleného serveru napodobí náš počítač, a tak internetové připojení bude potřeba pouze před spouštěním aplikace pro stažení potřebných *Java* knihoven.



Obrázek 20. Schéma odeslání dat třídou PrintWriter

Výše uvedený obrázek znázorňuje schematicky komunikaci klientské *Java* aplikace se vzdáleným serverem.

1. Internetový prohlížeč klienta odešle požadavek na konkrétní webový server prostřednictvím internetové sítě.
2. Server požadavek zpracuje a výsledek pošle zpět ve formátu textové HTML stránky.
3. Po odeslání celé části textu „trychtýř“ vyprázdní metodou `close()`.
4. Internetový prohlížeč zprávu zpracuje (parsuje) a jeho grafickou podobu zobrazí uživateli na monitoru počítače.

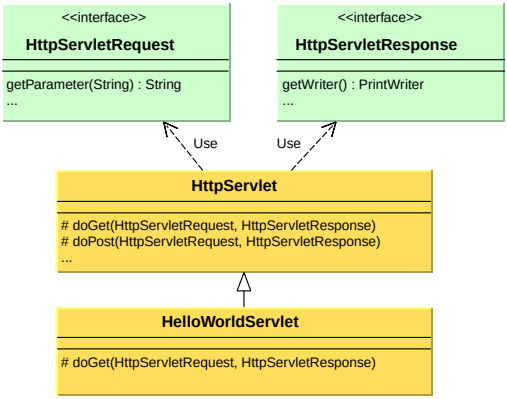
5.2.1 Hello, World!

Prvním naším úkolem bude vypsat (prostředky jazyka *Java*) jednoduchý text „Hello, World!“ do internetového prohlížeče ve formátu HTML, jehož kód by mohl vypadat takto…

Zdrojový kód 37. Obsah výsledné HTML stránky

```
<!DOCTYPE html>
<html lang='en'>
    <head>
        <meta charset='UTF-8'/>
        <title>Demo</title>
        <link href='css/basic.css' rel='stylesheet'/>
    </head>
    <body>
        <h1>Hello, World!</h1>
    </body>
</html>
```

Pro nás to znamená vytvořit třídu (potomka třídy `HttpServlet`) s překrytou metodou `doGet(request, response)`, která bude zapisovat **HTML kód** (uvedený výše) do internetového prohlížeče – prostřednictvím objektu `PrintWriter` získaného z argumentu `response`. Nová třída se může jmenovat třeba `HelloWorldServlet`, podívejme se na diagram tříd:



Obrázek 21. Diagram servletu

V diagramu vidíme dva zeleně vyznačené datové typy, které třída `HttpServlet` *používá* (anglicky *use*) v metodě `doGet()`, jinými slovy lze také říct, že třída `HttpServlet` je na nich *závislá*. Jeden typ reprezentuje žádost o zpracování (anglicky *request*) a druhý jeho odpověď (anglicky *response*). Jejich vztah k servletu je vyznačen přerušovanou čárou zakončenou **otevřenou** šipkou. Nové typy se nazývají *rozhraní* (anglicky *interface*), a tímto anglickým výrazem se označuje také hlavička (stereotyp) grafického prvku.

Připomeňme si krátce, že rozhraní *interface* představuje popis *metod* objektů (včetně parametrů a návratových typů). Aby mohl vzniknout objekt typu tohoto rozhraní, je třeba vytvořit nejdříve třídu, která bude abstraktní *metody* (toho

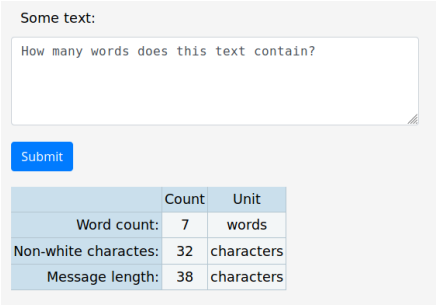
❸ V cyklu projdeme postupně každý znak parametru `message`, řídící proměnná `i` zde reprezentuje zároveň *index* aktuálního znaku.

❹ Znak parametru (na dané pozici) získáme metodou `charAt(i)`. Pro kontrolu mezery (přesněji *bílého znaku*) využijeme statickou metodu `Character.isWhitespace()` ze standardní knihovny *Java*.

❺ Pokud dojde ke změně typu znaku, změníme obsah logické proměnné `lastWhitespace`.

❻ Pokud předcházející znak **nebyl** bílým znakem, přičteme k výsledku jedničku. Připomeňme si, že negace logického typu se provádí operátorem `!` (vykřičník), čímž se hodnota `false` změní na `true` a naopak.

Implementace metody `countNonWhiteCharacters()` stojí na podobných principech. HTML stránka by mohla po spuštění vypadat takto:



Obrázek 29. Formulář počítání slov

Řešení celého servletu najdeme v ukázkových příkladech.

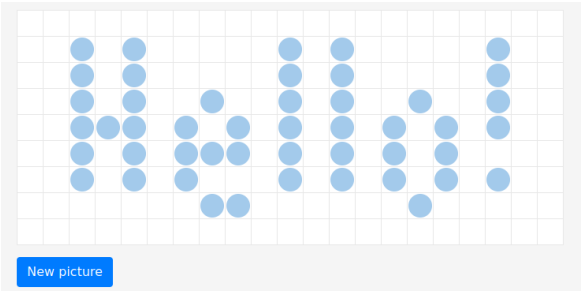
Závěrem proveďme ve zdrojovém kódu vlastní změny: Doplňme tabulku výsledků o počet **velkých písmen**, variantou je počítat pouze **číslovky**. Metodu vhodnou pro rozpoznávání potřebných znaků najdeme v dokumentaci *JavaDoc* ke třídě `Character`.

5.5 Zábava

V této sekci najdeme dvě interaktivní aplikace a jednoduchou strategickou hru.

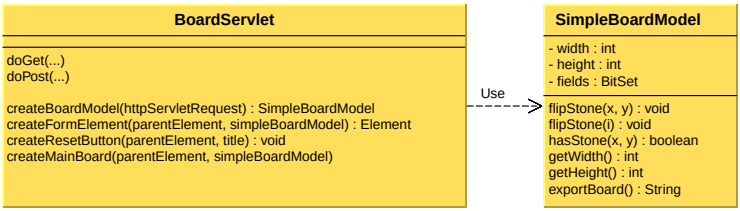
5.5.1 Bodové kreslení

V následujícím řešení si vyzkoušíme sestavení dvoubarevného obrazu, kde jednotlivé body budeme zakreslovat do mřížky kliknutím myši, výsledný vzhled by se měl blížit následujícímu obrázku.



Obrázek 30. Obrazovka bodového kreslení

Mřížku vykreslíme pomocí HTML elementu `table`, kde do každé buňky umístíme jedno tlačítko `button` typu `submit` (bez popisu) a do atributu `name` zapíšeme jednoznačný identifikátor tlačítka, celá tabulka musí být uvnitř elementu `form`. Pro práci s mřížkou budeme potřebovat *datový model*, který postavíme na třídě `SimpleBoardModel`. Podívejme se na diagram tříd servletu s modelem:



Obrázek 31. Diagram tříd bodového kreslení

Třída `SimpleBoardModel` obsahuje šířku a výšku kreslicí desky (v počtu buněk) a nějaký objekt na popis stavu jednotlivých bodů mřížky. Zde se také nabízí použít **dvourozměrné** pole primitivních logických hodnot `boolean[][]`, my však půjdeme jinou cestou a využijeme objekt typu `BitSet`. Jeho výhodou je snadná manipulace s daty a menší paměťové nároky, nevýhodou je jeho **jednorozměrnost**, kterou však lze napravit v aplikačním rozhraní (API) modelu mřížky. Každému bodu lze přiřadit **identifikační číslo**, začneme vlevo nahoře hodnotou `0` a pokračujeme směrem doprava a na konci řádku přejdeme na řádek nižší. Tímto způsobem očíslujeme celou mřížku. Abychom však práci s body modelu typu `SimpleBoardModel` zpřehlednili, nabídneme metody s parametry obou souřadnic, které v implementaci přepočítáme na identifikátory buněk. Využijeme pro to operátor `%` (modulo). Popišme si jednotlivé metody modelu:

1. `flipStone(x, y)` – metoda změní logickou hodnotu buňky v mřížce, kterou si můžeme představit jako kámen tmavé barvy umístěný do mřížky: prvním zavoláním metody kámen do mřížky vložíme, druhým zavoláním kámen zase odstraníme. Buňka v tabulce je určena souřadnicemi `x` a `y`,