Obsah

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`), tak různá zařízení (stroje, přístroje), která interně umí pracovat s čísly, se nazývají občas *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*.

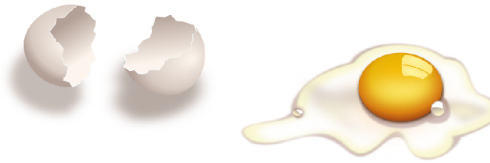
Předpony velkých čísel využijeme třeba při kontrole volného místa na pevném disku.

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

procesoru, které umožňuje *paralelní zpracování*. Toto téma však přesahuje plánovaný rámec knihy, a tak budeme dál uvažovat o všech algoritmech pouze v režii jediného jádra procesoru. Dopad na praktický život by to mělo ten, že nákup chybějících potravin zajistí kuchař sám.

Někdy nastane situace, že kuchař není schopen připravit požadovaný počet porcí najednou — třeba kvůli omezenému objemu kuchyňského nádobí. V tom případě můžeme postup vaření (*algoritmus*) opakovat tak dlouho, dokud počet porcí neodpovídá očekávání (*podmínka*). Procesu opakování se říká *iterace*. Protože však nakupování potravin, během každé iterace, by neúměrně zpomalilo čas přípravy, společný nákup lze provést jen jednou. Procesu urychlení pak můžeme říkat *optimalizace* algoritmu.

Programové *příkazy* se provádějí běžně shora dolů (nebo zleva doprava, pokud je více *příkazů* na jednom řádku) a každý *příkaz* se provádí běžně až po dokončení *příkazu* předchozího. Pro tuto chvíli pomiňme příkazy jazyka pro *opakované* či *souběžné* zpracování na více procesorech.

Jak už jsme naznačili před chvílí, algoritmy se v jazyce *Java* zapisují (v textovém formátu) do útvarů zvaných *metody*. Každá metoda má svůj **název**, může vyžadovat seznam **parametrů** a může vracet návratovou hodnotu. Parametry umožňují *předávání dat* do metody z místa, kde je metoda volána svým jménem. Nákup potravin by vedl k návrhu metody s těmito vlastnostmi:

1. název metody: „běž nakoupit“
2. parametr metody: „seznam potravin“
3. návratový objekt: „nákupní taška se zakoupenými potravinami“

Z pohledu kuchařky je však popis *algoritmu* nákupu nezajímavý. Kuchařka potřebuje pouze vědět, jak se *metoda* pro poskytování chybějících surovin jmenuje a jak ji má použít.

Zapamatujme si, že při počítačovém zpracování se využívají *algoritmy* (například recept na přípravu pokrmů) a *data* (nákupní seznam, potraviny, přepravní taška). Průchod algoritmem lze (za nějaké podmínky) opakovat pomocí *iteračních* příkazů programovacího jazyka.

4.3 Proměnná

Pojem *proměnná* označuje *symbolické jméno* pro úschovu informace za běhu programu. Proměnnou si můžeme pro začátek představit jako krabici na úschovu předmětů: pro uložení sezonních bot lze použít krabici na boty opatřenou štítkem „Petrovy zimní boty“. V případě potřeby lze krabici podle štítku snadno vyhledat a obsah použít. Jedna krabice na body však pojme právě jeden pár bot a vložením nového páru vede ke ztrátě původního obsahu. Do krabice s Petrovými botami se tedy nevejdou boty od tchyně — pokud ovšem nechceme o ty Petrovy boty přijít. Pro úschovu toaletního mýdla si zvolíme raději krabičku menší. K té podobnosti s krabicí mají nejblíže *primitivní* datové typy, které si vysvětlíme v následující kapitole.

Při tvorbě programů používáme *proměnné* k úschově dat pro pozdější použití. Pamatujme, že využití proměnné má jen omezenou platnost v rámci nějakého programového bloku, detaily vysvětlíme později.

4.4 Primitivní datové typy a přetypování

V jazyce *Java* rozlišujeme tři základní skupiny datových typů:

- primitivní typy,
- třídy objektů a
- pole.

V jazyce *Java* má každá proměnná svůj pevně daný datový typ, který se uvádí při její deklaraci, a doplňme, že od verze *Java* 10 existuje ještě deklarace typu proměnné prvním přiřazením hodnoty. Pod pojmem *deklarace* si lze představit nějaké **prohlášení** o něčem, při programování jde často právě o datový typ. Tahle kapitola se zabývá popisem **primitivních** datových typů, k těm ostatním se dostaneme postupně. Primitivní typy obsahují číslo z vymezeného intervalu, případně jeho speciální interpretaci, jako je *znak* nebo *logická hodnota*. Rozsah povolených hodnot primitivního datového typu je součástí specifikace jazyka a jejich počet je omezen – na rozdíl od *tříd*, které si můžeme vytvářet prakticky v neomezeném množství. *Java* nabízí celkem osm primitivních typů, jejich názvy začínají vždy *malým písmenem*. Slovo *literál* ve zdrojovém kódu označuje zápis konkrétní hodnoty nějakého konkrétního typu. Například celočíselný *literál* s hodnotou **deset** lze ve zdrojovém kódu zapsat výrazem `10`, *literál* desetinného čísla lze zapsat podle vzoru: `65.731F`; Literály uvedené v následující tabulce však nejsou jedinou platnou formou zápisu, více informací je třeba tady [19]. Číselné limity zapsané (v tabulce) v exponenciálním tvaru (neboli *vědeckou notací*) byly zaokrouhleny. Jazyk *Java* nabízí prostředky, kterými lze získat (za běhu programu)

přesné mezní hodnoty. Význam posledního sloupce tabulky si vysvětlíme později — v kapitole o vlastnosti zvané Autoboxing.

Tabulka 1. Přehled primitivních datových typů jazyka Java

#	Typ	Literál/ Konstanta	Rozsah	Nejmenší hodnota	Největší hodnota	Objektový ekvivalent
1.	`boolean`	`true`	1 bit	false	true	`Boolean`
2.	`byte`	`(byte) 65`	8 bitů	-128	127	`Byte`
3.	`short`	`(short) 65`	2 bajty	-32 768	32 767	`Short`
4.	`char`	`'A'`	2 bajty	0	65 536	`Character`
5.	`int`	`2_000_000`	4 bajty	-2.1e9	2.1e9	`Integer`
6.	`long`	`65L`	8 bajtů	-9.2e18	9.2e18	`Long`
7.	`float`	`65.7313F`	4 bajty	-3.4e38	3.4e38	`Float`
8.	`double`	`65.731`	8 bajtů	-1.8e308	1.8e308	`Double`

Popis hodnot

1. `boolean` – logická hodnota; může obsahovat jen dva stavy: `true` a `false`,
2. `byte` – nejmenší číselný typ (česky **bajt**) obsáhne *znakovou sadu* **ASCII** s 256 znaky, do této sady se však nevejdou ani všechny národní znaky. Velikost datové informace se uvádí často jako násobek tohoto datového typu. *Literál* se vytváří *přetypováním*, což je změna datového typu vyznačená v kulatých závorkách. Příklad použití je vidět v tabulce, podrobnější informace jsou uvedeny dále,
3. `short` – krátké celé číslo. Ani tento datový typ nemá vlastní *literál*, hodnotu tohoto typu je třeba vytvořit přetypováním,
4. `char` – datový typ reprezentující nějaký **grafický symbol**, který může vyjadřovat *písmeno*, *číslici*, *mezeru*, ale také *piktogram* či jiný *speciální znak*. Úplný výčet povolených znaků je popsaný sadou *Unicode* [11]. Hodnota tohoto typu ve skutečnosti obsahuje jen pořadové číslo znaku z této tabulky. Při **zobrazování** je však podstatné, že na monitoru (či jiném výstupním zařízení) dojde k vykreslení odpovídajícího grafického symbolu. Pro lepší představu lze nahlédnout na řešený příklad z kapitoly o sestavování textu,
5. `int` – nejčastěji využívané celé číslo v jazyce *Java*. Hraniční hodnoty uvedené v tabulce jsou jen orientační, přesné hodnoty jsou třeba zde [19]. Hodnota se uvádí jako číslo bez desetinné části. Pro lepší čitelnost lze číslice primitivních typů prokládat (ve zdrojovém kódu) znakem podtržítka (_), na hodnotu čísla to však nemá vliv,
6. `long` – velké celé číslo, jeho číselná hodnota se doplňuje znakem `L`, přípustný je i malý znak, který se však kvůli podobnosti s jedničkou příliš nepoužívá,

7. `float` – typ čísla s (plovoucí) desetinnou čárkou, jeho hodnota se doplňuje znakem `F`, přípustný je i malý znak. Přesnost obsaženého čísla je omezená (svým typem), a tak při výpočtu může docházet k zaokrouhlení,
8. `double` – typ **velkého** čísla s (plovoucí) desetinnou čárkou, jeho hodnotu lze doplnit (volitelně) znakem `D`, přípustný je i malý znak. Přesnost obsaženého čísla je omezená (svým typem), a tak při výpočtu může docházet k zaokrouhlení.

Zdrojový kód 1. Ukázka zápisu primitivní hodnoty do proměnné

```
int intValue = 65; ❶
double doubleValue = 65.82; ❷
char charValue = 'A'; ❸
```

❶ Zápis čísla do (nové) proměnné s názvem `intValue`. Operátorem přiřazení je v jazyce *Java* znak: rovnítko (`=`). Nalevo od něj se nachází **datový typ** s **názvem proměnné**, na opačné straně je uveden číselný literál odpovídajícího typu. Přiřazovací příkaz se zakončuje středníkem (`;`).

❷ Literál typu `double` zapíšeme do proměnné s názvem `doubleValue`.

❸ Literál typu `char` zapíšeme do proměnné s názvem `charValue`.

Občas je třeba změnit hodnotu ze stávajícího datového typu na jiný, takové operaci se pak říká *typová konverze*, nebo zkráceně *přetypování*. Číselné hodnoty lze bez větších problémů přiřadit do proměnných s **přesnějším** primitivním datovým typem. Pokud však hrozí riziko ztráty původní informace, je třeba použít přetypování explicitní, které se provádí uvedením kulatých závorek s cílovým typem.

Zdrojový kód 2. Příklad přetypování čísla typu `int`

```
long largeValue = 65; ❶
byte smallValue = (byte) 65; ❷
short shortValue = (short) 654321; ❸
int floatValue = (int) 65.82; ❹
```

❶ Literál typu `int` přetypujeme na číselný typ s **větším** rozsahem automaticky — tedy bez jakýchkoliv formalit.

❷ Pro přetypování hodnoty typu `int` do typu s **menším** rozsahem je třeba výrazu předřadit cílový typ uzavřený v kulatých závorkách `()`.

❸ Literál typu `int` přetypujeme explicitně na číslo s **menším** rozsahem, ale vlivem odříznutí části významných bitů dostane proměnná nekorektní hodnotu `-1039`.

❹ Literál typu `double` přetypujeme explicitně na **celé** číslo, čímž dojde k odříznutí desetinné části.

4.5 Java výrazy a příkazy

Pojem *programový výraz* si lze představit jako (matematický) **vzorec**, kde dosazením hodnot získáme nějaký **výsledek**. Přesněji bychom mohli říci, že *programový výraz* ve zdrojovém kódu označuje *literál*, *proměnnou*, volání *metody*, nebo jejich kombinaci (spojovanou *operátory*) v souladu se syntaxí jazyka tak, aby jejich vyhodnocením za běhu programu vznikla **hodnota**. Pokud hodnotu výrazu zapíšeme do *proměnné*, vytvoříme *přiřazovací příkaz* a ten se zakončuje středníkem, jak už zaznělo v předchozí kapitole. Vedle *přiřazovacích příkazů* rozeznáváme ještě *příkazy volání metod* a *příkazy jazyka*.

Již jsme zmínili, že zápis hodnoty do proměnné se skládá ze dvou částí oddělených rovnítkem (`=`) a že levá strana obsahuje název proměnné, do které chceme výsledek zapsat. Při prvním použití se název proměnné uvádí datovým typem, ale možností je více a přípustná je i deklarace proměnné bez přiřazení hodnoty. Pravá část popisuje programový *výraz* odpovídajícího typu.

Zapamatujme si, že pokud je programový kód (v těle metody) zakončen středníkem, jedná se o **příkaz**. Java však nabízí také příkazy jazyka, kde se středník neuvádí.

Ve sloupci **Kód** následující tabulky najdeme vybrané ukázky *výrazů* a *příkazů*, ukázky bez středníku reprezentují *výraz*. Zalamování řádků (na pozici mezer) je pro kompilaci nevýznamné.

Tabulka 2. Ukázka výrazů a jejich zápisu do proměnných

#	Kód	Výsl.	Typ	Poznámka
1.	`boolean agreement = true;`	true	boolean	Přiřadí hodnoty logického *literálu* do nové proměnné `agreement` (česky souhlas).
2.	`char myCharacter = 'A';`	A	char	Zápis znaku `A` do proměnné s názvem `myCharacter`, hodnota znakového literálu se ohraničuje jednoduchou uvozovkou.
3.	`int count = 5;`	5	int	Zápis číselné hodnoty `5` do nové proměnné `count` (počet).
4.	`int goodsPrice = count * 100;`	500	int	Do proměnné `goodsPrice` (cena zboží) zapíšeme součin dvou čísel. Znak `*` představuje operátor násobení, znak `/` je určen pro dělení.

#	Kód	Výsl.	Typ	Poznámka
5.	`int myCash = 2 * 100 + 3 * 20 + 1;`	261	int	Do nové proměnné `myCash` (moje hotovost) zapíšeme výsledek jednoduchého výpočtu. Znak `+` představuje operátor sčítání, znak `-` je určen pro odečítání. Operátor násobení má obecně *přednost* (anglicky *precedence*) před sčítáním, jinou preferenci lze vynutit kulatými závorkami.
6.	`boolean shopping = myCash >= goodsPrice;`	false	boolean	Do logické proměnné `shopping` (nákup) zapíšeme výsledek porovnání, zdali naše hotovost stačí na nákup zboží. V tomto případě nestačí a hodnota proměnné bude `false`. Velikost hodnot primitivních typů lze porovnávat pomocí operátorů: `<` (je menší než), `>` (je větší než), `<=` (menší či rovno), `>=` (větší či rovno), `==` (rovná se), `!=` (nerovná se). Výsledkem porovnání je logická hodnota typu `boolean`.
7.	`myCash = myCash + 1_000;`	1261	int	Tento zápis změní hodnotu dříve deklarované proměnné `myCash`. Všimněme si, že výraz na pravé straně používá stejnou proměnnou pro získání své původní hodnoty. Uvedený *příkaz* zvýší hotovost v proměnné `myCash` o `1000` korun. Znak podtržítka v čísle můžeme použít pro grafické oddělení řádu tisíců (například), jeho použití však není povinné.
8.	`myCash += 1_000;`	2261	int	Uvedený výraz představuje stručnější zápis toho předchozího, k naší hotovosti přičte dalších `1000` korun, aktuální hotovost tedy bude `2261` korun. Analogický přístup lze využít také pro odečítání, násobení a dělení.
9.	`shopping = myCash >= goodsPrice;`	true	boolean	Nová kontrola hotovosti pro nákup zboží, výsledek změní původní hodnotu proměnné `shopping` na `true`.
10.	`myCash -= goodsPrice;`	1761	int	Po zaplacení zboží zbude v proměnné `myCash` hotovost 1761 korun.
11.	`myCash <= Short.MAX_VALUE`	true	boolean	Příklad srovnání aktuální hotovosti s maximální hodnotou typu `short`. Minimální hodnota se vyjadřuje výrazem `Short.MIN_VALUE`, mezní hodnoty ostatních primitivních typů se zapisují analogicky.
12.	`boolean evenNumber = (myCash % 2) == 0;`	false	boolean	Pro porovnání shody primitivních datových typů se používá operátor `==`, vyhodnocení rozdílu se provádí operátorem `!=`, operátor `%` slouží pro výpočet **modulo** (zbytku po celočíselném dělení). Operátor `%` má přednost před `==`, pokud si však nejsme jisti, přednost vyznačíme závorkami. Proměnná `evenNumber` bude obsahovat hodnotu `true`, pokud číslo obsažené v proměnné `evenNumber` modulo `2` se rovná `0` (nule), což platí pro všechna **sudá čísla**.

#	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

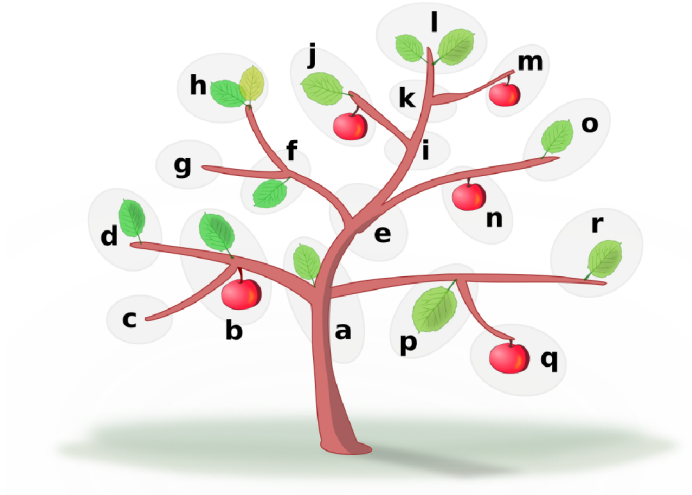
Výše uvedené příklady uvádějí jen vybraný seznam operátorů a jejich možných kombinací, další příklady najdeme v kapitole Programové výrazy a podrobnější výklad tématu třeba zde [22]. Výhodou primitivních typů je nízká paměťová náročnost a podpora `operátorů` (výrazů pro matematické a logické operace).

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.

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*).



Obrázek 3. Schéma stromu

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 kořen 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.

K obecnému pojmenování takových zjednodušených napodobenin můžeme použít slovo *model*. Pokud se model skládá z informací vhodných pro počítačové zpracování, říkáme mu *datový model*.

4.8 Fyzikální objekt

Význam slova *objekt* v běžné řeči je poměrně obecný a závisí na kontextu. Pro jeho lepší pochopení v prostředí *objektově orientovaného jazyka* nám může pomoci jistá podobnost s výrazem *těleso*, který se používá ve fyzice pro popis fyzikálních zákonů. Slovem *těleso* se označuje hmotný předmět, který má své **místo** (či adresu), **velikost** v prostoru a také **čas** vzniku i zániku, přitom není podstatné, zdali pozorovatel tyto údaje zná. Za *těleso* lze považovat libovolný konkrétní předmět (dům, kedlubna, veverka), mezi *tělesa* však nepatří abstraktní pojmy (chůze, spánek, fyzikální jednotka).

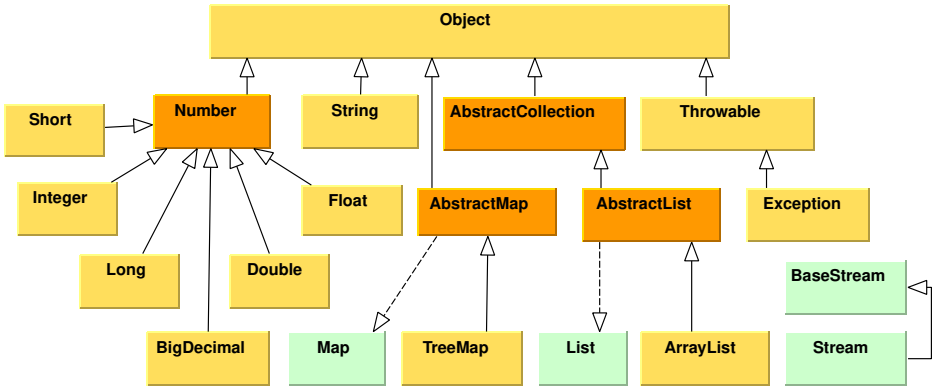
Je užitečné si připomenout, že *tělesa* mají své **vlastnosti** (např. hmotnost, barvu, teplotu) a mohou poskytovat také **služby**, např. automat na jízdenky může (za určitých podmínek) vytisknout cestovní lístek. Pojem *těleso* pokrývá jen podmnožinu objektů reálného světa, ale kvůli některým podobnostem s počítačovými objekty bude užitečné zmínit ho za chvíli.

4.9 Programový objekt

Programový objekt (v našem případě *Java objekt*) je datový **model** uložený v operační paměti počítače, který popisuje reálné či abstraktní objekty reálného světa pomocí *dat* a *algoritmů*. Jeden objekt může reprezentovat konkrétní automobil, vybraný umělecký styl nebo zelenou barvu. Co to má společného s *tělesem*? Počítačový *objekt* má také své **místo** v prostoru (adresu v operační paměti počítače), **velikost** (objekt zabírá přesně vymezenou část paměti) a **čas** svého vzniku i zániku. *Java* objekty vznikají *příkazem* a zanikají při automatickém sběru nepotřebných objektů, jak už bylo zmíněno na začátku knihy. Všechny objekty zaniknou nejpozději při vypnutí počítače, z pohledu definice už není podstatné, že existují techniky pro ukládání objektů na pevný disk i pro jejich případnou obnovu v paměti po opětovném zapnutí počítače. Programové objekty (podobně jako tělesa) mají také své vlastnosti a mohou nabízet služby, jako třeba matematické výpočty či hledání dat.

4.14.6 Shrnutí kapitoly o třídách

Standardní knihovna jazyka *Java* obsahuje řadu užitečných *tříd* a *rozhraní*, které se řadí do tématických balíčků. Společným rodičem každé třídy je třída `Object`. Každá *třída* může (ale nemusí) implementovat jedno či více *rozhraní*. Každé *rozhraní* může (ale nemusí) mít svého předka. Číselné objektové typy (popsané v této knize) mají společného abstraktního předka `Number`, abstraktní třídy (podobně jako *rozhraní*) však nemohou vytvářet objekty přímo, ale potřebují implementaci potomka. Zmíněné číselné objekty také neumožňují změnu své číselné hodnoty, vždy se vytváří nový objekt – podobně jako když třída `String` spojuje další texty.



Obrázek 10. Diagram vybraných tříd

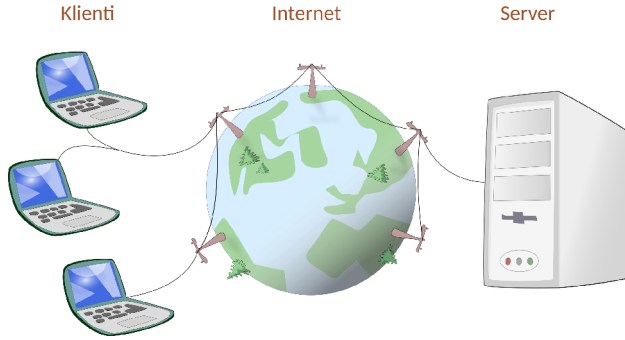
Třída `ArrayList` je jedna z několika implementací rozhraní `List` a třída `TreeMap` je jedna z několika implementací rozhraní `Map`. Výjimkou jsou objekty typu `Throwable` se řídí stejnými pravidly jako ostatní třídy, ale ve spojení s klíčovým slovem `throw` umožňují předčasně ukončit běh programu. Pokud je nikdo nezachytí blokem `try-catch`, povedou ke shození celé aplikace. Objekty typu `Stream` vytváříme zpravidla nějakou tovární metodou.

4.15 Webové technologie a pojmy

Na chvíli se vzdálíme od programovacího jazyka *Java*, všechny nové pojmy pak brzy zhodnotíme v reálných příkladech.

Podívejme se ale nejdřív na zjednodušené fungování webových aplikací a zkusme si představit běžného čtenáře webových stránek, který v internetovém prohlížeči klikne na nějaký odkaz. Internetový prohlížeč je aplikace, která primárně zobrazuje webové stránky v graficky pěkné podobě. Nejznámější produkty jsou: *Chrome*, *Firefox*, *Microsoft Edge*, existuje však celá řada dalších. Internetový **odkaz** si pak lze

představit jako poštovní **adresu** stránky na internetu, která se označuje běžně zkratkou URL (je odvozena z anglického termínu: *Uniform Resource Locator*). Pro zjednodušení můžeme říci, že internetový prohlížeč předá prostřednictvím internetové sítě žádost webovému serveru o získání nějaké webové stránky. Webový server tuto žádost posoudí a vrátí zpět buď stránku očekávanou, nebo i nějakou nestandardní, která může obsahovat třeba zprávu o chybě. Taková komunikace využívá síťovou architekturu, která se označuje pojmem **klient-server**, a je pro ni typické, že **klient** žádá (prostřednictvím počítačové sítě) službu nějakého serveru a čeká na odpověď. Popsanou architekturu zobrazuje následující schéma:



Obrázek 11. Schéma internetové sítě

Doplňme, že na straně serveru může být celá řada alternativních technologií, které zvládnou sestavení webové stránky. Pro potřeby této knihy budeme využívat prostředky programovacího jazyka *Java*.

4.15.1 XML – rozšiřitelný značkovací jazyk

Jazyk XML (z anglického *Extensible Markup Language*) nabízí sadu pravidel umožňujících zápis `stromové struktury dat` v textovém formátu. Jazyk se využívá nejen při tvorbě webových aplikací: pro svoji dobrou čitelnost a podporu mnoha knihoven (napříč technologiemi) našel řadu uplatnění i v dalších oblastech při zpracování dat. Pro popis *uzlů stromu* se používají XML *elementy*, které se zapisují v textu párovými značkami obsahujícími *jméno elementu* ohraničené špičatými závorkami.

Pojem *element* označuje *uzel* stromu včetně všech jeho dětských *uzlů* (přesněji elementů).

S touto znalostí můžeme zkusit přepsat část struktury jabloně (z kapitoly Stromová struktura dat) do formátu XML, výsledek je zde: