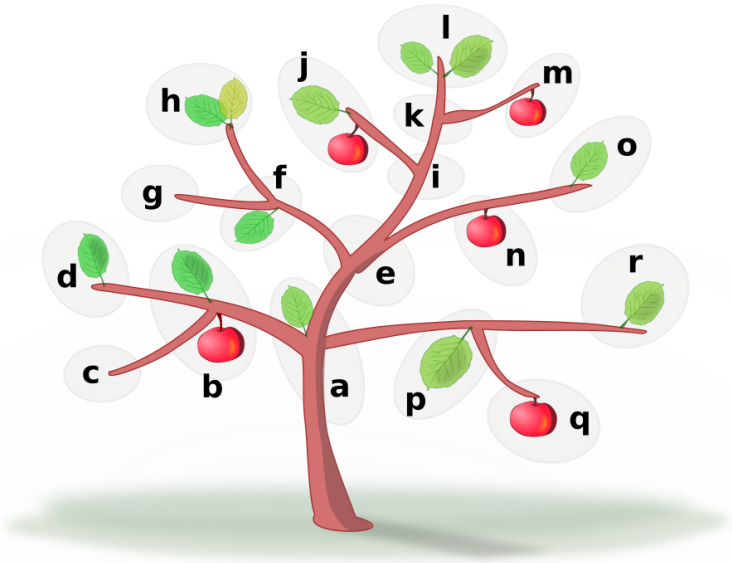
Obsah

#	Kód	Výsl.	Typ	Poznámka
14.	`2 + 3 == 6 - 1`	true	boolean	Porovnání výsledků dvou operací. Matematické operátory mají přednost před porovnáním.
15.	`2 + 3 <= 6 - 2`	false	boolean	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	Negace logického výrazu.

Znak podtržítka v číselném literálu se používá pro grafické oddělení, jeho použití není povinné. 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`. Zde jsou uvedeny jen vybrané operátory. Vlastní preference lze vynutit kulatými závorkami. Primitivní typy jsou výhodné pro svou nízkou paměťovou náročnost a přímou podporu operátorů. Další příklady najdete v kapitole Programové výrazy.

4.7 Stromová struktura dat

Data lze seskupovat do struktur, které zrychlují vyhledávání nebo usnadňují manipulaci. Se **stromovou** strukturou se budeme setkávat často, proto si představíme základní pojmy.



Obrázek 5. 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*). Tento pojem obecně označuje jakékoliv místo, ke kterému se připojují další potomci. Na obrázku jabloně to jsou další větve, plody či listy stromu.

Koncový *uzel*, který již nemá žádné potomky, se nazývá *list* (anglicky *leaf*). Každý uzel má právě jednoho *rodiče*, s výjimkou prvního uzlu, kterému říkáme *kořen* (anglicky *root*). Uzlům i listům můžeme přiřadit názvy. Spojnice mezi dvěma uzly se nazývá *hrana*. Přesnou adresu uzlu zapíšeme jako postupný seznam názvů od kořene, například oddělený tečkou.

```
a.e.f.h.žlutý_list
```

Podobně bychom mohli lokalizovat i červené jablko vpravo nahoře.

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

Stromovou strukturu má také adresa na poštovní obálce (stát → město → ulice). Jiným příkladem může být *hierarchická klasifikace organismů*, dalších příkladů by se našlo určitě více.

4.8 Model reálného světa

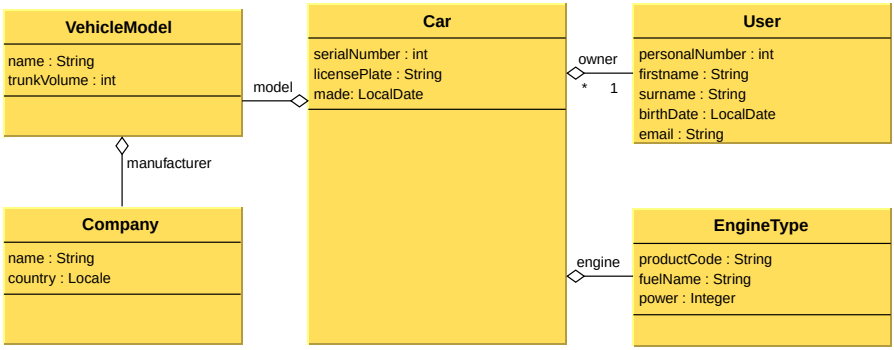
Vysvětlení pojmu *model* uvedeme krátkou definicí:

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

Děti si hrají s modely autíček nebo s panenkami, což jsou v napodobeniny předmětů reálného světa. Často se zachovává tvar či barva, zatímco jiné vlastnosti jsou potlačeny kvůli bezpečnosti nebo ceně. Pokud se model skládá z informací pro počítačové zpracování, říkáme mu *datový model*.

4.9 Fyzikální objekt

Při programování si *objekt* můžeme představit jako digitální model *tělesa*, který napodobuje jeho vlastnosti i chování. Slovem *těleso* se označuje hmotný předmět, který má své **místo** (či adresu), **velikost** v prostoru a také **čas** vzniku i zániku. Za *těleso* lze považovat libovolný konkrétní předmět (např. dům nebo veverku),



Obrázek 7. 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 Vytvoření objektu pro datum.

❷ Konstruktor se dvěma parametry. Atribut `JAPAN` je statický. Objekt lze vytvořit a rovnou předat jako parametr bez ukládání do proměnné.

❸ Propojíme model s autem.

❹ Datum lze vytvořit i pomocí konstanty reprezentující měsíc.

❺ Propojíme uživatele s autem jako vlastníka.

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

    MyPrinter.print(made, manufacturer,
        modelName, trunkVolume, firstname, surname, owner); ❻
    return; ❼
}
```

❶ Zavoláme metodu pro vytvoření auta. `this` znamená, že voláme metodu na stejném objektu, kde se zrovna nacházíme. Slovo `this` lze v při volání metod zpravidla vynechat.

❷ Načtení data výroby do proměnné `made`.

❸ Ukázka řetězení metod: postupně se „prokoušeme“ až k názvu výrobce. `car.getModel()` nám dá model, na něm zavoláme `getManufacturer()` a nakonec `getName()`. Chybný název metody odhalí kompilátor nebo chytrý editor.

❹ Další ukázka řetězení.

❺ Spojení tří textů do jednoho (celé jméno).

❻ Předání dat k tisku.

❼ Příkaz `return` ukončí metodu. Na konci metody s návratovým typem `void` se ten příkaz psát nemusí.

Všimněte si, že v diagramu přibyla nová třída popisující typ energie automobilu (například benzin, nafta, elektřina z baterie). Může nás také zajímat, jak se zachová volání *metody* `getName()` v případě, že metoda `getManufacturer()` **neposkytuje** žádný přiřazený objekt. Volání *metod* na neexistujícím objektu vyvolá chybu při běhu programu, která předčasně ukončí provádění *metody*. Jak už jsme zmínili dříve, hodnota prázdných objektů se v kódu popisuje klíčovým slovem `null`, a takové prázdné objekty jsou výchozí (anglicky `default`) hodnotou každého *objektového atributu* či *pole*. Přesněji bychom mohli říci, že tento případ vede k vyhození *výjimky*. Více informací o *výjimkách* (anglicky *exceptions*) se dozvíme v kapitole Výjimky. Hodnotu `null` lze také zapisovat do proměnných, posílat do *metody*, či použít při porovnání instancí. Přikládám několik příkladů použití.

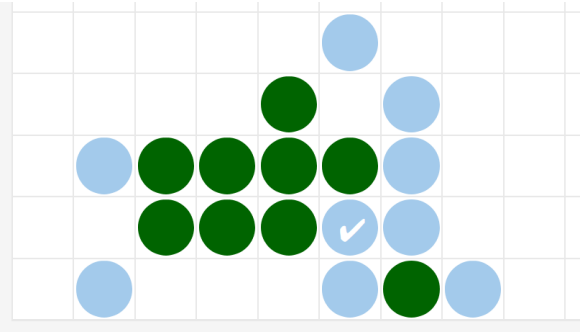
❶ Nejdříve si připravíme číslo, jehož bitová hodnota bude o jedničku větší než maximální hodnota čísla čtyřbitového rozsahu.

❷ Operátor `|` (česky *svislík*, *kolmítko*, *roura*, anglicky *pipe*) provádí *bitový součet* dvou čísel, což znamená, že do výsledku promítne *logický součet bitů* na stejné pozici. Je pravda, že náš výsledek bude o jeden znak delší, ale správnou hodnotu získáme snadno metodou `substring()`. Pro úplnost uveďme také doplňkový operátor `&` pro *bitový součin*, který provádí logický součin *bitů*.

Závěrem proveďme ve zdrojovém kódu vlastní změny: rozsah znakové sady rozšiřme o **jeden bit** tak, aby sada pojala dvojnásobný počet znaků. Dále připravme model na vkládání znakové sady **konstruktorem**, přičemž do konstruktoru vložíme kontrolu velikosti znakové sady, která bude vyhazovat výjimku typu `IllegalArgumentException`.

7.5.4 Piškvorky s defenzivní strategií hry

Hra Piškvorky patří mezi **strategické hry**, jejichž kořeny sahají hluboko do historie. Existuje několik variant, mezi nejznámější patří zřejmě hra *Gomoku* s alternativním názvem *5 in row* (v překladu 5 v řadě). Naše pravidla hry budou následující: hráči kladou střídavě barevné kameny do mřížky s konečnou velikostí, každý hráč má svoji barvu. Vítězem se stává hráč, který jako první postaví nepřerušenou řadu alespoň pěti kamenů své barvy, povolena je i řada šikmá. Hru je možné kdykoli resetovat tlačítkem a zahájit hru novou. V naší implementaci bude protihráčem vždy počítač a výhodu prvního tahu dostane vždy člověk.



Obrázek 36. Náhled hrací desky

Strategie SW protihráče bude pouze jednoduchá obrana. Cílem hry totiž není vytvořit neporazitelný algoritmus, ale především ukázat řešení vhodné pro další studium a později třeba i pro implementaci vlastních nápadů. Čtenáři mohou nad rámec této knihy vyhledat některou knihovnu zaměřenou na hry typu `Gomoku` a zkusit napojit její algoritmy do stávající architektury.

Teď už je snad jasnější, že zavedení výčtových typů přispělo ke stručnějšímu zápisu kódu docela významně. Tím však výhody enumerátorů nekončí, **lze** je používat efektivně také v *příkazech* `switch`.

Příkaz switch

Příkaz `switch` umožňuje větvit program podle *primitivních celočíselných hodnot* (literálů či konstant). Z objektů jsou podporovány ještě typy `String` a `Enum`. Přikládám zkrácenou ukázku použití ze třídy `BoardModel`.

Zdrojový kód 76. Ukázka použití příkazu switch

```
/** Set a stone by an index */
public void setStone(BoardPoint point, StoneEnum stone) {
    switch (stone) { ❶
        case BLACK_STONE: ❷
        case WHITE_STONE:
            fields.set(convertToBitSetIndex(point, stone)); ❸
            lastMove.set(point);
            break; ❹
      default: ❺
            throw new IllegalArgumentException("Illegal stone" + stone);
    }
}
```

❶ *Příkaz* `switch` vyhodnocuje proměnnou, která musí být různá od `null`. Tělo *příkazu* je ohraničené složenými závorkami.

❷ Jednotlivé případy se zachytávají pomocí klíčového slova `case` s konstantou zakončenou dvojtečkou. Výčtový typ je jednoznačně určen v hlavičce *příkazu* a při zachytávání případů se už neuvádí.

❸ Následuje blok *příkazů*, které se provádějí až do nalezení klíčového slova `break`. Zpracování je shodné pro bílý i tmavý kámen. Uložíme pozici kamene do objektu typu `BitSet`, metoda `convertToBitSetIndex()` však nejprve spočítá hodnotu potřebného indexu. Pozici posledního položeného kamene si pak uložíme pro pozdější grafické zvýraznění.

❹ *Příkaz* pro ukončení bloku *příkazů*. V případě jeho absence by zpracování pokračovalo průchodem následujících případů v příkazu `switch`.

❺ Hodnoty, které nejsou zachyceny žádným případem `case`, lze zachytit (volitelně) klíčovým slovem `default`, přičemž tento blok nemusí být nutně na poslední pozici. Zde se vyhazuje výjimka, protože zadanému bodu má smysl přiřadit kámen pouze bílé nebo černé barvy.

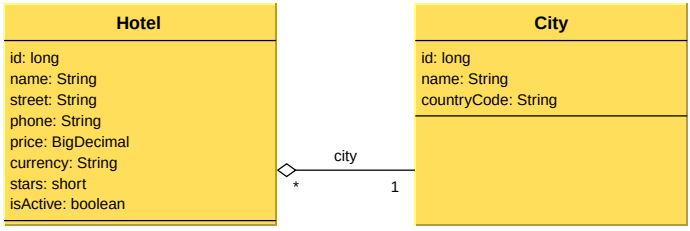
constraint). Pro rychlé vyhledávání může mít jeden či více sloupců definovaný *index*, což je pomocná datová struktura, kterou si databáze udržuje ve své režii. Unikátní *index* funguje zároveň jako *constraint* (omezení unikátnosti). Data se ukládají do *řádků* (anglicky *record* nebo *row*). Jeden sloupec databázové tabulky (nebo i více) může poskytovat jedinečný identifikátor řádku. Pokud má takový sloupec unikátní *index*, mluvíme o *primárním klíči*. Aby databáze mohla kontrolovat hodnotu cizího klíče (zda odpovídá *primárnímu klíči* v **cizí** tabulce), je třeba tento *vztah* deklarovat již při vytváření sloupce.

Je dobré vědět, že každý index zabírá místo na disku a jeho aktualizace mírně zpomaluje zápis.

Dvě *tabulky* mezi sebou mohou mít *vztah* (anglicky *relation*). Pokud má záznam v tabulce vazbu na více řádků cizí tabulky, mluvíme o vazbě typu **1:N**. Použití databáze si ukážeme na příkladu evidence **hotelů**. Zvažme uspořádání, kde jedno město může mít více hotelů (vazba **1:N**), ale jeden hotel může mít vazbu právě na jedno město (vazba **N:1**). V relačních databázích se vazba realizuje přidáním pomocného sloupce, který obsahuje hodnotu *primárního klíče* (zpravidla) jiné tabulky. Takovému sloupci se pak říká *cizí klíč*. Setkat se můžeme také s vazbou typu **M:N**. Například, jeden host může mít více rezervací v různých hotelech a jeden hotel může mít více hostů. V relační databázi se taková vazba řeší vložením pomocné vazební tabulky, která obsahuje cizí klíče na obě hlavní tabulky. Požadovaný vztah se tak rozloží na dvě vazby typu **N:1**.

8.2 Relace a entity

Zmínili jsme dvě databázové *tabulky* pro hotely a města s vazbou N:1. Protože s hotely chceme pracovat v prostředí Javy, připravíme si dvě entity, do kterých můžeme databázové záznamy kopírovat.



Obrázek 41. Entity hotelu

Popis položek diagramu:

- `Hotel` – objekt této třídy bude obsahovat data jednoho řádku tabulky `hotel`.

- `City` – objekt této třídy bude obsahovat data z jednoho řádku tabulky `city`.
- `id` – primární klíč mapovaný na sloupec `id`.
- `name` – název hotelu (či města).
- `city` – tento atribut bude v databázi realizován sloupcem `city_id`, který označíme jako *cizí klíč* do tabulky `city`.
- `isActive` – sloupec typu boolean umožňující logicky vyjmout řádek z vyhledávání bez jeho smazání.
- `countryCode` – kód země, který by v reálné aplikaci pravděpodobně odkazoval do samostatného číselníku zemí.

8.3 Jazyk SQL aneb jak mluvit s databází

Požadavky na dotazování a manipulaci dat se popisují jazykem *SQL* (anglicky *Structured Query Language*). Jazyk *SQL* je standardem, který byl naposledy aktualizován v roce 2023 pod označením *ISO/IEC 9075:2023*. V této kapitole si ukážeme jen základní možnosti jazyka, na další studium lze navázat například zde [38].

Dodavatelé databází implementují standard, ale často přidávají vlastní dialekty, které nejsou přenositelné.

Mezi základní operace pro práci s daty patří **vložení**, **čtení**, **aktualizace** a **mazání**. Soubor těchto čtyř operací se označuje zkratkou *CRUD* (z anglických sloves: *Create*, *Read*, *Update*, *Delete*). V jazyce *SQL* se tyto operace realizují příkazy `INSERT`, `SELECT`, `UPDATE` a `DELETE`. Abychom však mohli pracovat s daty, musíme mít nejdříve definovanou databázovou strukturu. Názvy databázových struktur zpravidla nerozlišují malá a velká písmena, a pokud se názvy skládají z více slov, oddělují se běžně podtržítkem. *SQL* rozlišuje velikost písmen pouze u textových hodnot v uvozovkách (v některých případech). Pro přehlednost budou klíčová slova jazyka *SQL* psána (v této knize) velkými písmeny a ostatní názvy (tabulky, sloupce, indexy) budou psány písmeny malými. Jednořádkové komentáře v kódu se uvádějí zpravidla dvěma znaky pomlček `--`.

8.3.1 Tvorba databázové tabulky

Začněme příkazem pro založení tabulky hotelů. Seznam sloupců je uzavřen v kulatých závorkách a oddělovačem je čárka. Mezery v *SQL* příkazech mohou obsahovat i zalomení řádků, což budeme pro přehlednost využívat. Předpokládejme, že tabulka měst s názvem `city` již existuje.

❶ Protože zde pracujeme přímo s objektem knihovny *JDBC*, je třeba propagovat (či zachytávat) kontrolovanou výjimku.

❷ Návratový objekt si připravíme v proměnné `result`. Do objektu pak postupně přepíšeme hodnoty z objektu `ResultSet`, který obsahuje metody pro čtení různých datových typů. Na sloupec se můžeme odkázat jeho databázovým názvem (což je přehlednější), nebo pořadovým číslem v *SQL* dotazu (což vede k rychlejšímu běhu). Obecně se doporučuje preferovat čitelnost kódu a výkon optimalizovat až v případě prokázaných problémů.

❸ Metoda se používá pro dva mírně odlišné příkazy `SELECT`. V jednom je k dispozici navíc **název města**. Abychom nemuseli psát a udržovat dvě různé implementace mapování, tak přiřazení města podmíníme druhým pomocným parametrem s názvem toho volitelného sloupce. Pokud název databázového sloupce není definován, zápis se neprovede.

Specifikace *JDBC* poskytuje také API pro získání dat o struktuře databáze. Říkáme tomu *metadata*. Právě takové informace využívá i metoda `hasTables()`. Zjednodušená implementace kontroluje pouze přítomnost databázové tabulky `hotel`. Pokud tabulka chybí, budeme předpokládat, že chybí i druhá tabulka `city`. Práce s metadaty však patří mezi pokročilejší techniky, a tak případné zájemce jen odkážu na dokumentaci *JavaDoc*. Implementaci najdeme v příkladech.

8.9 Vkládání Java objektů do databáze

Připomeňme, že vkládání dat do databáze provádíme příkazem `INSERT` a kód najdeme ve třídách typu `DAO`. Podívejme se na komentovanou ukázku.

Zdrojový kód 92. Java kód pro vložení hotelu.

```
public long insert(Hotel entity) {
  String sql = """
    INSERT INTO hotel ❶
    (  name,  city_id, street,  phone,  price,  stars ) VALUES
    ( :name, :cityId, :street, :phone, :price, :stars )
    """;
  try (SqlParamBuilder builder = builder()) { ❷
    builder.sql(sql)
      .bind("name", entity.getName()) ❸
      .bind("cityId", entity.getCity().getId())
      .bind("street", entity.getStreet())
      .bind("phone", entity.getPhone())
      .bind("price", entity.getPrice())
      .bind("stars", entity.getStars())
      .executeInsert(); ❹
    return builder.generatedKeys(resultSet -> resultSet.getLong(1))
                  .findFirst().get(); ❺
  }
}
```

❶ Příkaz se od nativního *SQL* liší jen značkami parametrů.

❷ Využití metody pro získání builderu zpřehlední kód a usnadní případnou změnu sestavení na ostatních místech.

❸ Formu přiřazení hodnot už známe z příkazu `SELECT`.

❹ Metoda `execute()` by nám data také korektně vložila. Pokud však potřebujeme získat hodnotu databázového identifikátoru, je třeba zavolat metodu `executeInsert()`.

❺ Metoda `generatedKeys()` pak vrátí seznam přiřazených identifikátorů ve formátu `Stream<ResultSet>`. Při vkládání jednoho hotelu můžeme očekávat jeden identifikátor. Podle dokumentace *JDBC* ho najdeme na pozici `1`. V případě prázdného Streamu by metoda `get()` vyhodila výjimku. Pro případ obecného použití tento kód přesuneme do samostatné metody abstraktního předka. Takovou metodu využije i třída `CityDao`.

Ostatní metody obou *DAO* objektů jsou psány ve stejném duchu, kód najdete v přiložených příkladech.

8.10 Generická abstraktní třída

Aplikace pro vyhledávání hotelů obsahuje pro každou databázovou tabulku jednu DAO třídu. Reálná aplikace může pracovat s desítkami takových tabulek, a proto je vhodné přesunout obecné služby do společného předka. Dalším důvodem je sjednocení API podobných metod tak, aby měly stejný název i logiku. Příkladem může být vyhledávání databázového záznamu podle jeho primárního klíče nebo vkládání nového hotelu do databáze. V takovém případě však budeme potřebovat v abstraktní třídě pracovat s datovým typem, který zatím ještě neznáme. To je dobrá příležitost připomenout si *generické datové typy*, které tento problém elegantně řeší. Implementaci si vysvětlíme na zjednodušené ukázce abstraktní DAO třídy.

Zdrojový kód 93. Generická abstraktní DAO.

```
/** Common DAO object
* @param <E> Entity type */
public abstract class AbstractDao<E> { ❶
    public abstract Optional<E> findById(long id); ❷
    public abstract long insert(E entity); ❸
}
```

❶ Generický typ třídy je reprezentován textem uvedeným za názvem třídy ve špičatých závorkách. Třída může mít i více generických typů, jejichž názvy se pak oddělují čárkou. Použitý znak `E` je odvozený od slova *Entita*. Význam generického typu je vhodné zdokumentovat v *JavaDoc*.