Problematika perzistentního (trvalého) ukládání dat ve vybraném programovacím jazyce.

*"Data s dobou života překračující běh aplikačního programu i vypnutí počítače. Data mohou být uložena v jednoduchém plochém **souboru**, nebo mohou být uložena v nějakém typu **databáze**."*

Další zdroje vhodné pro prostudování:
I/O operace I. – Standardní IO API
I/O operace II. – Pokročilejší využití IO API
JDBC – Java Database Connectivity (eng.)
Relační databáze – Relační databáze
ORM – Objektové-relační mapování (eng.)

Zpracované otázky z minulých ročníků (viz. Podklady.zip)

Podle přístupu k datům můžeme persistentní data rozdělit do kategorií:

- **částečně perzistentní:** datové úložiště obsahuje **jen** aktuální verzi dat.
- **perzistentní:** datové úložiště obsahuje aktuální verzi dat a verzi dat před poslední změnou (v případě, že během transakce dojde k chybě, je použita funkce *rollback*, která vrátí DB do původního stavu, před začátkem transakce).
- **plně perzistentní:** datové úložiště obsahuje aktuální verzi dat + všechny verze dat.

Zvolený jazyk: Java (možnost práce s daty je ovlivněna vybraným jazykem a úložištěm dat)

Možnosti ukládání dat

- **Standardní IO API** (standardní vstupní/výstupní operace) uložené v balíku *java.io* (i další třídy jako **File** pro práci se soubory/adresáři či **StreamTokenizer**/**StringTokenizer** = rozdělení obsahu na úseky určené zadanými oddělovači).
  Každá z abstraktních tříd má specifického potomka (abstraktní třídu), která je předkem filtrů/dekorátorů = převádějí primitivní I/O stream na „pokročilejší" stream, např.:
  - *DataInputStream*/*DataOutputStream* (pro primitivní datové typy)
  - *BufferedInputStream* /*BufferedOutputStream* (primitivní stream na bufferovaný stream)

  - **Soubory** (soubor s vlastní strukturou) = posloupnost dat uložená na externí paměťová média (disk, disketa, …). Používají se také pro komunikaci s periferními zařízeními.
    > Výjimky **IOException** a **FileNotFoundEcxeption**
    - **Textové** – čitelná podoba dat, posloupnost znaků uspořádaných do řádků.
      - čtení po znacích > abstraktní třída *FileReader*
        `FileReader soubor = new FileReader(„soubor.txt");` (čtení po znacích soubor.read();)
        `BufferedReader br = new BufferedReader( soubor );` čtení po řádcích br.readLine();
      - výstup po znacích > abstraktní třída *FileWriter*
        `FileWriter soubor = new FileWriter(„soubor.txt");` (zápis po znacích soubor.write();)
        `PrintWriter pw = new PrintWriter( soubor );` (po řádcích pw.println();)
    - **Binární** – číslicová podoba dat, posloupnost *bajtů*.
      - čtení po bytech > abstraktní třída *FileInputStream*
        `FileInputStream soubor = new FileInputStream(„soubor.dat")`
      - výstup po bytech > abstraktní třída *FileOutputStream*
        `FileOutputStream soubor = new FileOutputStream(„soubor.dat")`
  - **Streamy** (**proudy dat**) – implementují (jako objekty) mimo jiné přístup k datům uloženým v souborech, obecné použití (komunikace po síti), sekvenční přístup
    - **Vstupní a výstupní** (binární a znakové[1])
      - čtení po bytech > abstraktní třída *InputStream*
      - výstup po bytech > abstraktní třída *OutputStream*

[1] využití lokálních znakových sad a kódování

čtení a zápis 1 bytu > **read**(), **write**(B) (readBoolean(); readChar();)
čtení a zápis pole bytů > **read**(poleB), **write**(poleB)
čtení a zápis úseku z pole bytů > **read**(poleb,od,kolik), **write**(poleb,od,kolik)
zavření > **close**()

- čtení po znacích > abstraktní třída *Reader*
>Třída *Reader* by se měla používat vždy, když je potřeba číst text. Garantuje správnou obsluhu znakových sad a převod textu do vnitřního kódování Javy (což je *Unicode*).
- výstup po znacích > abstraktní třída *Writer*
- **ObjectInputStream** - umožní číst celé objekty, kontejnery atd. (nutná *serializace*)
- **ObjectOutputStream** oos = new ObjectOutputStream(soubor); (zápis objektů oos.writeObject(obj); (nutná *serializace*)

- **Databáze**
  Pro přístup se používá JDBC rozhraní (použití ovladače pro překlad požadavků do nativního volání daného úložiště – databáze, Excel soubory, Access atp.).
  JDBC specifikace rozpoznává čtyři typy JDBC ovladačů:

  1. **JDBC-ODBC bridge –** ovladač přistupuje k databázi přes nativní ovladač ODBC, který **musí** být na daném stroji nainstalován.
  2. **Native-API driver –** ovladač přistupuje k databázi přes služby nativního ovladače dané databáze (využívá jeho metody, které jsou vytvořeny v C, C++), **musí** být na daném stroji nainstalován. Jistá podoba s typem 1.
  3. **Network-Protocol driver –** založen čistě na Javě a JDBC, který konvertuje svoji komunikaci do síťového protokolu a spojuje se s centrálním serverem (Network Server), který poskytuje připojení k databázi (obvykle s „poolem“ připojení). Tento server konvertuje síťový protokol, kterým komunikuje s klienty, do databázově specifického protokolu, jemuž již databáze rozumí. Takový model je vysoce efektivní, a to jak s ohledem na možnost „poolingu“ připojení a tím i zrychlení dotazování a práce s databází, tak i možnost připojení k sadě heterogenních databázových systémů.
  4. **Native-Protocol driver –** Tento ovladač komunikuje s databázovým serverem přímo přes síťový protokol.

JDBC API

- používá pro práci s DB několik základních tříd a rozhraní, jsou to:

- **DriverManager:** Třída, která má na starost nahrávání a registraci dostupných ovladačů. Je to mezivrstva mezi JDBC ovladačem a programovým kódem.
- **Connection:** Toto rozhraní reprezentuje připojení k DB. Je to centrální rozhraní pro správu databázového spojení. Umožňuje přípravu a vykonání db dotazů, transakční zpracování.
- **Statement:** je rozhraní, které je zodpovědné za posílání SQL příkazů databázi. Rozhraní implementuje tyto metody:
  - **executeQuery(sql):** provadí SQL dotaz, který zpravidla vrací *ResultSet* což je seznam databázových vět.
  - **ExecuteUpdate(sql):** používá se v případě, že chceme modifikovat data v DB (INSERT, UPDATE, DELETE).
- **ResultSet:** Rozhraní představuje výsledky databázového dotazu(tabulku). Zpravidla je to několik datových vět. Metody tohoto rozhraní můžeme rozdělit do těchto kategorií:
  - **Navigační metody:** Tyto metody umožňují pohyb v množině výsledků pomocí **funkcí** *first(), next(), last(), previous().*
  - **Gettry:** *ResultSet* rozhraní poskytuje metody (getBoolean, getString, atd.), které vrací hodnoty jednotlivých sloupců v aktuální řádce.
  - **Modifikační metody:** Tyto metody umožňují modifikovat (měnit, vkládat, rušit ..) data v ResultSet a následně provést modifikaci v DB. Jsou to např. insertRow(), updateString, atd.

Ukázka:

```
Class.forName("org.hsqldb.jdbcDriver"); 1: Zavedení ovladače
String url = "jdbc:hsqldb:data/tutorial"; 2: Vytvoření spojení s DB
conn = DriverManager.getConnection(url, "sa", ""); 3 : Přihlášení k DB
```

```
st = conn.createStatement(); 4: Vytvoření instance Statement pro dotazování
String sqlQuery = "SELECT uid, name, duration from EVENTS"; 5 : Vytváření dotazu.
ResultSet rs = stmt.executeQuery(sqlQuery); 6: Zpracování ResultSet.
rs.first();
while (rs.next()) {
String name = rs.getString(2);
Timestamp hireDate = rs.getTimestamp(5);
System.out.println("Name: " + name + " Hire Date: " + hireDate);
}
```

Nevýhody JDBC

- **Problém relační model + objektový přístup:** Rozhraní *ResultSet* vrací data jako tabulku, ne jako objekty (OOP v Javě). Vhodnější řešení by bylo vrácení řádků dat jako objekty, které reprezentují jednotlivé entity v tabulkách (člověk, faktura). Protože *ResultSet* toto neumožňuje, programátor je velmi často nucen si takové třídy vytvořit a zde dochází k obrovské ztrátě času.
- **Vyjímky:** Při každé operaci je potřeba odchytat mnoho vyjímek což značně znepřehledňuje kód.

Objektově relační mapování

Objektově relační mapování slouží k tomu, aby bylo možné snadno používat relační databáze v prostředí objektově orientovaných programovacích jazyků - Javě. Vzhledem k tomu, že objektově orientovaný návrh dat není jednoznačně převoditelný na relační databáze a opačně, používají se různé formy mapování. Mapování má za účel načítat data z relační databáze a naplnit jimi příslušné datové položky objektů, případně naopak datové položky objektů ukládat do databáze. Snahou ORM je co nejlepším využití obou zmíněných technologií – tj. objekty by měly reprezentovat objekty reálného světa, jak to požadují principy OOP, na straně databáze bychom zase měli využít všech možností relačních databází – indexy, pohledy, primární klíče, atd. V současnosti je snad nejpoužívanějším nástrojem pro ORM produkt od firmy JBOSS Hibernate.

Objektové databáze

Objektové databáze umožňují skladování dat s libovolnou strukturou. Programátor či kdokoliv jiný se při použití objektové databáze nemusí vůbec starat o mapování objektových struktur do dvourozměrné tabulky. Každý persistentní objekt v objektové databázi má svoji vlastní identitu, je tedy jednoznačně odlišitelný od libovolného jiného objektu nezávisle na hodnotách svých vlastností – odpadají tedy problémy s tvorbou primárních klíčů. Kvalitní objektová databáze podporuje všechny vlastnosti nutné pro práci s objekty – *zapouzdření*, *jednoznačnou identifikaci*, *reference mezi objekty* a další (*dědičnost*, *polymorfismus* atp.).
Např.: db4o, Ozone, ObjectDB.
Existují i hybridní objektové databáze, které implementují pouze některé z výše zmíněných vlastností (**N**otes **S**torage **F**ile u Lotus Notes ☺).