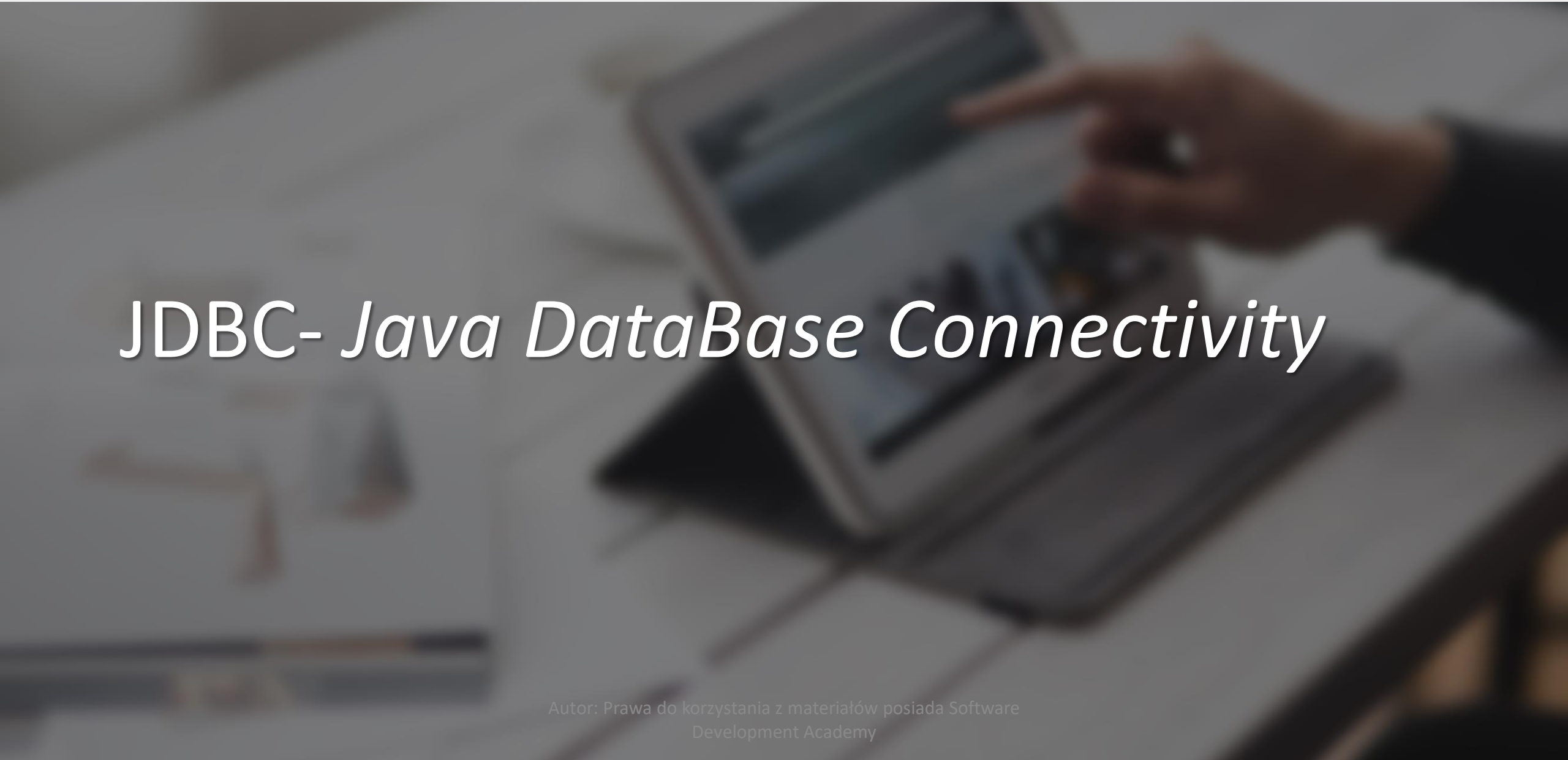
Bazy danych - programowanie (JDBC, Hibernate)

Robert Engelbrecht

SDA

Plan prezentacji:

1. JDBC
2. JPA (Java Persistance API)
3. Hibernate



JDBC- *Java DataBase Connectivity*

JDBC

JDBC (ang. *Java DataBase Connectivity* - łącze do baz danych w języku Java) – interfejs programowania opracowany w 1996 r. przez Sun Microsystems, umożliwiający niezależnym od platformy aplikacjom napisanym w języku Java porozumiewanie się z bazami danych za pomocą języka SQL. Interfejs ten jest odpowiednikiem standardu ODBC opracowanego przez SQL Access Group.

Środowisko Java Platform, Standard Edition zawiera API JDBC, natomiast użytkownik musi uzyskać specjalny sterownik JDBC do swojej bazy danych. Możliwe jest łączenie się z ODBC przez specjalne sterowniki, tłumaczące odwołania JDBC na komendy ODBC.

https://pl.wikipedia.org/wiki/Java_DataBase_Connectivity

JDBC – jak połączyć się z bazą danych

- Ładujemy sterownik:

  ```
  Class.forName("com.mysql.cj.jdbc.Driver");
  ```

  (Od wersji 4 nie trzeba ładować sterownika metodą Class.forName, sterownik jest wykrywany i ładowany automatycznie)

- Tworzymy poprawny adres URL:

  ```
  String url= "jdbc:mysql://localhost:3306/ksiegarnia";
  ```

- Tworzymy połączenie do bazy danych

  ```
  Connection connection= DriverManager.getConnection(url,"sdatest","Start123!");
  ```

- Zamykamy połączenie

  ```
  connection.close();
  ```

Wykonywanie poleceń SQL

JDBC udostępnia trzy interfejsy, które można wykorzystać do budowy zapytań:

- Statement –pomocny przy większości zapytań, nie można przekazać parametrów
- PreparedStatement–użyteczny gdy chcemy wiele razy wykonać zapytanie, akceptuje parametry
- CallableStatement–należy użyć gdy chcemy skorzystać z procedur bazodanowych, można przekazać parametry

Statement

```
Statement stmt= null;
try{
stmt= conn.createStatement( );
. . .
}
catch(SQLExceptione) {
. . .
}
finally{
stmt.close();
}
```

PreparedStatement

Pozwala sparametryzować nasze zapytanie, co za tym idzie wielokrotnie wykorzystać to samo zapytanie. Aby dodać parametry, używamy „?”:

SELECT * FROM ksiazka WHERE tytul like "%"?"%";

Parametry dla konkretnego PreparedStatement preStmt dodajemy poprzez:

preStmt.getXXX([nr parametru],wartość)

Pamiętamy o tym, że numerujemy parametry od 1.

Następnie wykonujemy jedno z poleceń:

- executeQuery() -wykonuje zapytanie i zwraca obiekt ResultSet
- execute() -wykonuje zapytanie i zwraca true jeśli zapytanie zwróciło ResultSet i false gdy nie ma wyników
- executeUpdate() –wykonuje instrukcje DML (INSERT, UPDATE, DELETE) i zwraca liczbę przetworzonych rekordów lub 0

PreparedStatement

```
PreparedStatement preStmt= null;
try{
preStmt= conn.prepareStatement( );
preStmt.setXXX(1,”wda”);
preStmt.executeQuery();
. . .
}
catch(SQLExceptione) {
. . .
}
finally{
preStmt.close();
}
```

PreparedStatement

Zawsze dla sparametryzowanych zapytań SQL używamy PreparedStatement. Nie używamy nigdy konkatenacji Stringów (nie używamy +), aby wstawić parametr do zapytania SQL. Unikniemy sql injection, czyli wstrzyknięcia złośliwego kodu przy zapytaniach SQL

ResultSet

Wynik zapytania SELECT uzyskujemy w obiekcie ResultSet.

Do następnego rekordu przechodzimy używając metody next().

Dane z poszczególnych rekordów pobieramy metodą getXXX() w zależności od oczekiwanego typu danych, np. obiekt String pobieramy metodą getString(). Metody getXXX() jako parametr przyjmują id kolumny lub nazwę kolumny.

CRUD

CRUD – od ang. create, read, update and delete (pol. utwórz, odczytaj, aktualizuj i usuń) cztery podstawowe operacje aplikacji korzystających z baz danych. Odpowiednikami w SQL są:

- Create –INSERT
- Read –SELECT
- Update –UPDATE
- Delete -DELETE

Try-with-resources

```
try(Connection conn …){
…
}catch{
…
}
```

Konstrukcja pozwalająca zamykać automatycznie zainicjalizowane zmienne w bloku try. Zmienne, które mogą być zamykane muszą implementować interfejs java.lang.AutoCloseable. Interfejs ten posiada jedną metodę close(). Blok try objęty jest nawiasami () i tam inicjalizujemy zmienne np. Connection, Statement, PrepareStatement.

Należy pamiętać, o kolejności inicjalizacji zmiennych. Będą one zamykane w odwrotnej kolejności.

Transakcje w jdbc

Transakcje obsługujemy metodami obiektu Connection:

- commit() -do zatwierdzania transakcji
- rollback() –do cofnięcia/wycofania transakcji

Defaultowo ustawiony jest autocommit, co oznacza że wszystkie wywołania execute() są zatwierdzane automatycznie. Żeby to wyłączyć musimy użyć:

```
connection.setAutoCommit(false);
```

JPA (Java Persistance API)

ORM

ORM (ang. *Object-Relational Mapping)* mapowanie obiektowo-relacyjne

sposób odwzorowania obiektowej architektury systemu informatycznego na bazę danych (lub inny element systemu) o relacyjnym charakterze.

https://pl.wikipedia.org/wiki/Mapowanie_obiektowo-relacyjne

JPA

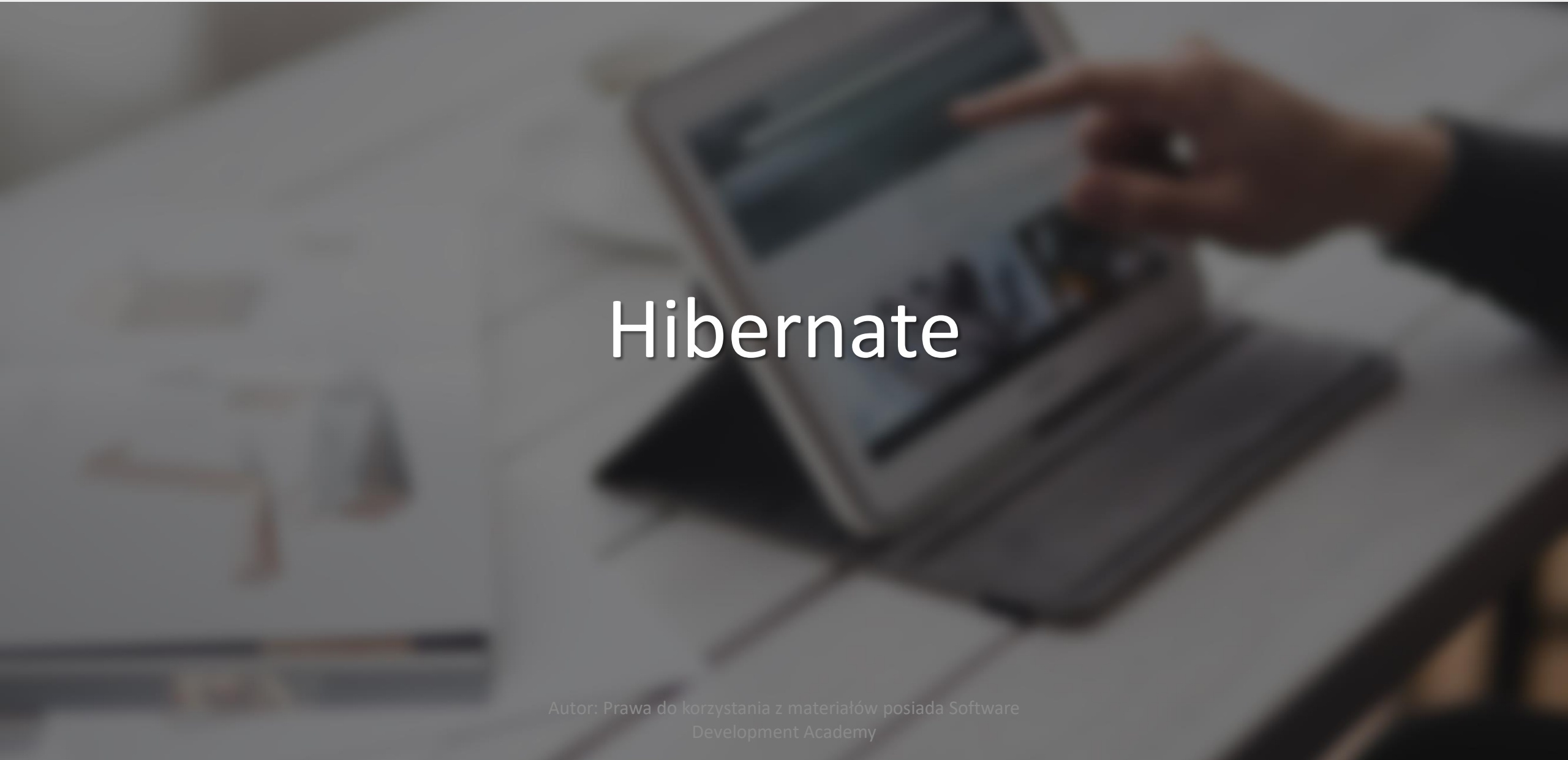
JavaPersistence API jest standardem ORM dla języka Java. Z punktu widzenia programisty jest to możliwość operowania na obiektach - zwanych **encjami** - oraz zapisywania wyników operacji do relacyjnej bazy danych za pomocą obiektu **EntityManager**. Sposób w jaki obiekty i ich połączenia przekładają się na elementy bazy danych są definiowane za pomocą adnotacji lub dokumentów XML. Poza standardowym zestawem operacji udostępnianych przez obiekt EntityManager standard JPA definiuje język zapytań **JPA Query Language** podobny do SQL.

https://pl.wikipedia.org/wiki/Java_Persistence_API

JPA

Najpopularniejszymi implementacjami JPA są:

- Hibernate
- EclipseLink
- SpringData



Hibernate

Czym jest Hibernate

- Jeden z najpopularniejszych frameworkow ORM w Javie
- Implementuje JPA
- Tam pod spodem też jest JDBC
- Znosi z programisty pisanie wielu instrukcji SQL
- Działa tak samo, niezależnie od relacyjnej bazy danych

EntityMangaer, Session

EntityManager to interfejs, który pozwala na operacje bazodanowe (zapisywanie, odczytywanie, modyfikowanie itd.). **EntityManger** pobieramy korzystając z interface **EntityManagerFactory.** Działanie **EntityMangera** opisuje dokładnie JPA.

Jako że Hibernate implementuje JPA, musi również implementować **EntityManagera**. Takim odpowiednikiem **EntityManagera** jest obiekt **Session. Session** pobieramy z **SessionFactory**. **SessionFactory** tworzony jest na podstawie dostarczonej konfiguracji (informacje do połączenia z bazą danych, konfiguracj hibernate)

Session

Wybrane metody Session:

- *find()* – pobiera encję o zadanym id
- *persist()* – zapisuje **nową** encję
- *delete()* – usuwa zadaną encję
- *merge()* – ponownie dołącza **zapisaną** wcześniej encję odłączoną od kontekstu utrwalania (zwraca zarządzaną kopię przekazanej encji)
- *flush()* – wymusza wykonanie instrukcji SQL na bazie danych
- *clear()* – czyści kontekst utrwalania

Encje

Klasa która jest odwzorowaniem tabeli bazy danych w obiektowym świecie Javy. Klasę taką oznaczamy adnotacją @Entity.

Encja musi:

- być oznaczona adnotacją @Entity
- być jednoznacznie identyfikowalny pomiędzy innymi podobnymi sobie obiektami, posiadać klucz/ identyfikator (uzyskujemy to dzięki adnotacji @Id na kolumnie z id)
- konstruktor bezargumentowy

Adnotacje do opisu encji

@Table –oznacza którą tabele reprezentuje obiekt

@Id –wskazuje kolumnę która jest unikalnym kluczem

@Column –oznacza do której kolumny dopasować pole

@GeneratedValue –określa w jaki sposób jest generowany id

@Temporal – mapowanie pola z datą/czasem

@Enumerated – mapowanie typu enum

@Transient – pole pomijane przy mapowaniu

Metody pobierające encje

Session/ EntityManager posiada szereg metod umożliwiających zdefiniowanie różnych zapytań pobierających encje:

- createQuery() – zapytanie JPQL
- createQuery() – gdy wyspecyfikujemy typ, możemy używać silnie typowanych zapytań budując kryteria w Javie (DSL)
- createNativeQuery() – zapytanie SQL
- createNamedQuery() – zapytanie zdefiniowane za pomocą adnotacji @NamedQuery

Typy relacji

Mamy następujące typy relacji:

- OneToOne –jeden do jednego
- OneToMany/ManyToOne –jeden do wielu/ wiele do jednego
- ManyToMany –wiele do wielu

O relacja mówimy również:

- jednokierunkowe (tylko właściiel relacji wie o dowiązanej encji)
- dwukierunkowe (obie strony- encje, wiedzą o sobie)

Lazy/ Eager

W Hibernate/ JPA rozróżniamy dwa tryby ładowania encji:

- Leazy tzw. leniwe ładowanie, dane pobierane dopiero przy pierwszym użyciu
- Eager tzw. chciwe ładowanie, dane ładowane od razu

W Hibernate

- Kolekcje ładowane są domyślie Leazy
- Wszystkie inne obiekty Eager

Aby ustawić tryb ładowania zmieniamy właściwość fetch dla relacji lub pola. Możliwe wartości to FetchType.Lazy lub FetchType.Eager.

HQL/JPQL

HQL/JPQL język tworzenia zapytań w Hibernate/JPA podobny do SQL, ale zamiast na tabelach operuje na encjach. HQL jest rozszerzeniem JPQL i daje większe możliwości. W większości funkcjonalności HQL i JPQL są takie same. Przykładowe użycie HQL/JPQL:

String hql = "from Student S where S.id > 10 order by S.lastName asc, S.firstName asc";

Query query = session.createQuery(hql);

List results = query.list();

Dziedziczenie

W specyfikacji JPA zostały przedstawione trzy strategie realizacji dziedziczenia po stronie bazy danych:

- table per class hierarchy
- table per subclass
- table per concrete class

Transakcje

Analogicznie jak w JDBC możemy zrobić commit(), rolback(), ustawić Savepoint()

Przykładowa transakcja:

```
session.beginTransaction();

…

Działania na bazie danych

…

session.getTransaction().commit();
```

@MappedSuperclass

Często tabele w bazie danych posiadają grupy takich samych kolumn np. dataDodania, uzytkownikModyfikujacy, dataOstatniejModyfikacji. Zamiast umieszczać w każdej encji te same pola, można je wydzielić do osobnej klasy. Wydzielona klasa musi zostać oznaczona adnotacją @MappedSuperclass, a każda encja w której chcemy wykorzystać zdefiniowane pola, musi po niej dziedziczyć.

@Embedded/ @Embedable

Niekiedy tabele w bazie danych mają wiele kolumn. Odwzorowując je w encje, czasami chcemy podzielić na kilka mniejszych obiektów. Przykładowo tabela posiada dane użytkownik:

Id, imie, nazwisko, ulica, nrDomu, nrMieszkania, kodPocztowy, miejscowość

Encja takiej tabeli może posiadać pola odpowiadające kolumnom, lub posiadać pola:

Id, imię, nazwisko, adres

Gdzie adres to klasa AdresUsera z polami:

ulica, nrDomu, nrMieszkania, kodPocztowy, miejscowość

Używamy do tego adnotacji @Embedded (dla pola encji w naszym wypadkuadres) i @Embedable (dla podklasy w naszym przypadku klasa AdresUsera)