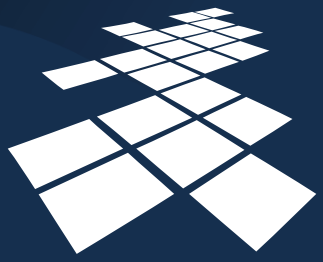
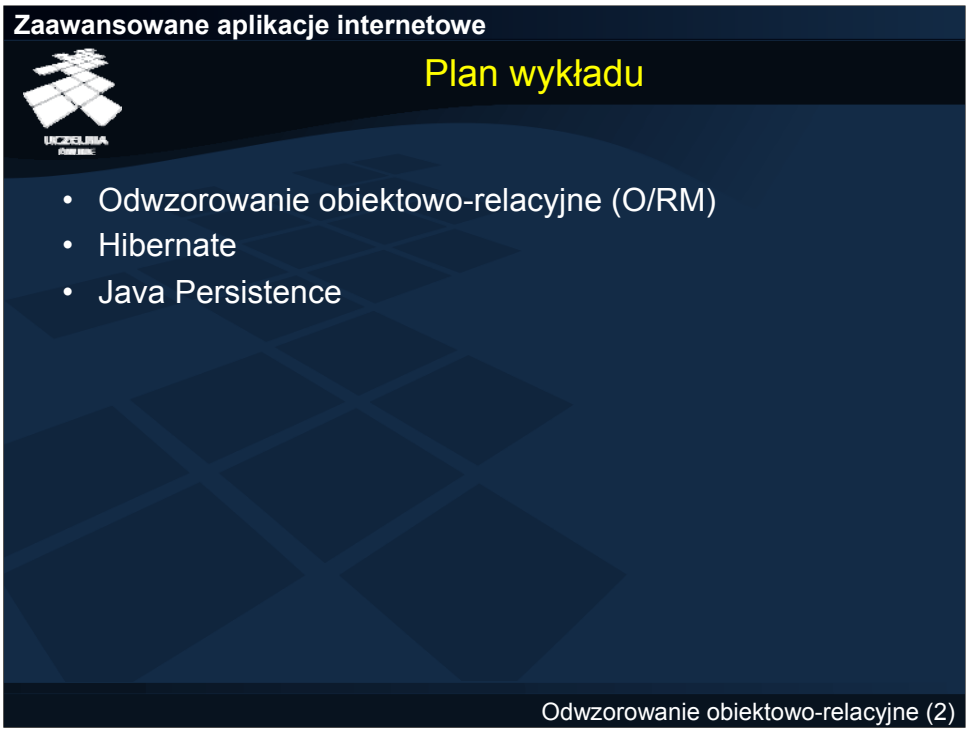
Odwzorowanie obiektowo-relacyjne

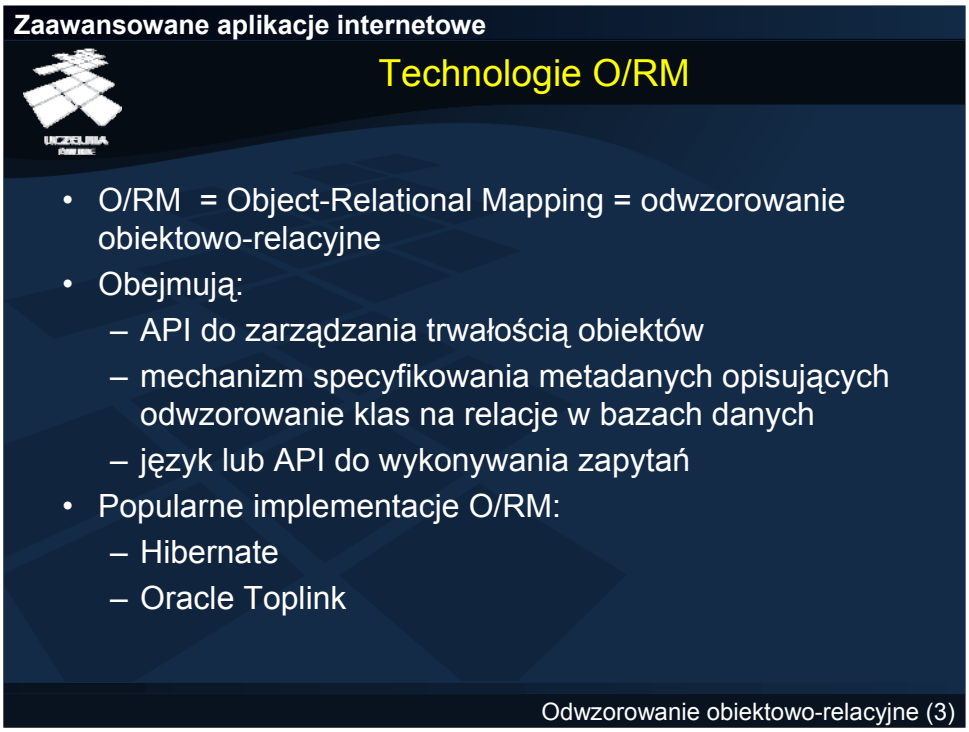
Wykład prowadzi:
Marek Wojciechowski



UCZELNIA
ONLINE



Celem wykładu jest przedstawienie zaawansowanych mechanizmów dostępu do baz danych w aplikacjach Java EE. Wykład rozpocznie się od krótkiego wprowadzenia do technologii odwzorowania obiektowo-relacyjnego (O/RM). Następnie omówiona będzie najpopularniejsza z technologii O/RM – Hibernate. Druga część wykładu poświęcona będzie nowemu standardowi dostępu do baz danych z aplikacji Java - Java Persistence.



Implementacja aplikacji Java pracujących na relacyjnej bazie danych na poziomie interfejsu JDBC jest czasochłonna i uciążliwa. Problem stanowi niski poziom abstrakcji interfejsu JDBC i różnice w organizacji danych między obiektowym językiem Java, a relacyjnymi bazami danych. Lansowana przez specyfikację Java EE do wersji 1.4 jako rozwiązanie tego problemu technologia encyjnych EJB okazała się niepraktyczna i nieefektywna. Jako alternatywę, różne środowiska zaproponowały technologie automatyzujące odwzorowanie obiektów na poziomie programu Java w struktury relacyjne. Technologie te są określane jako technologie odwzorowania obiektowo-relacyjnego (Object-Relational Mapping – w skrócie O/RM). Można z nich korzystać również w celu uzyskania obiektowej reprezentacji danych dla istniejącego schematu relacyjnej bazy danych.

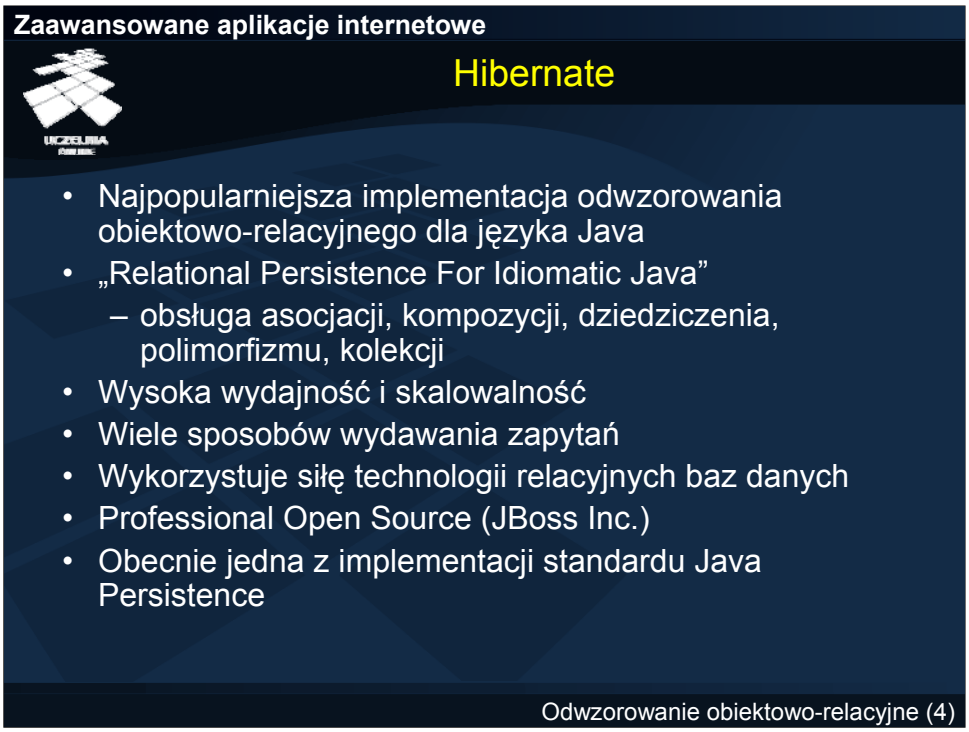
Elementy technologii O/RM to:

1. API do zarządzania trwałością obiektów;

2. Mechanizm specyfikowania metadanych opisujących odwzorowanie klas na relacje w bazach danych;

3. Język lub API do wykonywania zapytań.

Najpopularniejsze implementacje technologii odwzorowania obiektowo-relacyjnego dla aplikacji Java to Hibernate (rozwiązanie Open Source firmy JBoss) i Oracle Toplink (rozwiązanie firmowe firmy Oracle). Mniejszą popularność zyskała technologia JDO (firmy Sun).



Hibernate to najpopularniejsza implementacja odwzorowania obiektowo-relacyjnego dla języka Java. Założeniem jego twórców było zaoferowanie rozwiązania określanego po angielsku jako „Relational Persistence For Idiomatic Java”, co oznacza zapewnienie trwałości obiektów ze wsparciem dla wszystkich mechanizmów obiektowych języka Java, takich jak: obsługa asocjacji, kompozycji, dziedziczenia, polimorfizmu i kolekcji.

Hibernate cechuje wysoka wydajność i skalowalność oraz wiele możliwości wydawania zapytań do bazy danych. Hibernate wykorzystuje siłę technologii relacyjnych baz danych. Oferując warstwę abstrakcji pozwalającą na tworzenie przenaszalnych aplikacji, jednocześnie potrafi wykorzystać zalety poszczególnych konkretnych systemów zarządzania bazą danych.

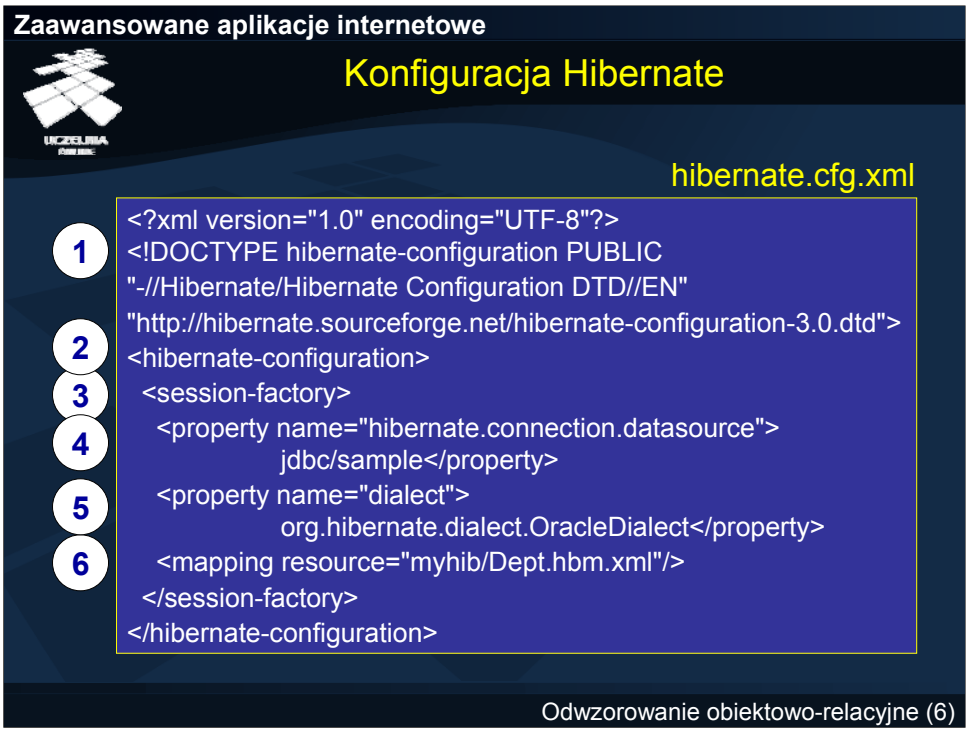
Hibernate jest rozwiązaniem kategorii Professional Open Source. Z jednej strony jego źródła są dostępne, a z drugiej jest on rozwijany przez firmę JBoss Inc., będącą znaczącym graczem na rynku serwerów aplikacji.

Twórcy Hibernate mieli znaczący wpływ na powstanie standardu Java Persistence, który będzie omówiony w drugiej części wykładu. Obecnie Hibernate oprócz swojego specyficznego interfejsu obsługuje również Java Persistence API i jest jedną z dostępnych implementacji dostawców usług trwałości dla standardu Java Persistence.

Slajd przedstawia uproszczoną architekturę Hibernate. Hibernate stanowi warstwę abstrakcji pośredniczącą w komunikacji aplikacji z bazą danych, obsługując trwałość obiektów aplikacji.

Opcje konfiguracyjne Hibernate, takie jak np. parametry połączenia z bazą danych zawiera plik konfiguracyjny Hibernate, który może mieć postać pliku .properties (o nazwie hibernate.properties) lub, co jest wygodniejsze, postać pliku XML (o nazwie hibernate.cfg.xml).

Obsługując trwałość obiektów, Hibernate wykorzystuje informacje o odwzorowaniu obiektowo-relacyjnym modelu danych aplikacji zawarte w plikach XML.



Slajd pokazuje przykład XML-owego pliku konfiguracyjnego Hibernate. Znaczenie poszczególnych elementów pliku konfiguracyjnego jest następujące:

1. Plik hibernate.cfg.xml jest plikiem XML. Jego struktura jest opisana we wskazanym pliku DTD.

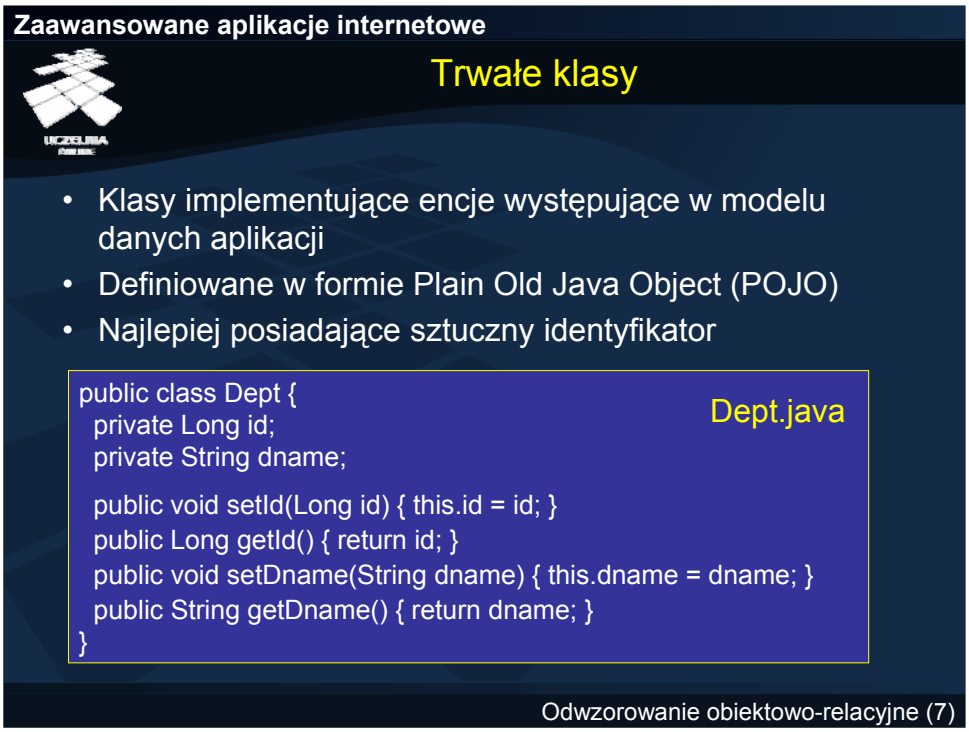
2. Konfiguracja Hibernate jest zawarta w elemencie głównym <hibernate-configuration>.

3. Parametry konfiguracyjne dla konkretnej bazy danych są zawarte w elemencie <session-factory>. Parametry konfiguracyjne połączenia z bazą danych są zawarte w elementach <property>, a pliki z opisem odwzorowania klas Java na tabele w bazie danych wskazane w elementach <mapping>.

4. Dla aplikacji Java EE konfiguracja połączenia z bazą danych typowo sprowadza się do podania nazwy JNDI źródła danych reprezentującego bazę danych. Istnieją również parametry konfiguracyjne umożliwiające podanie nazwy użytkownika, hasła i parametrów połączenia JDBC.

5. Ważnym parametrem jest wskazanie dialektu SQL dla wykorzystywanej bazy danych (w przykładzie dialekt bazy danych Oracle). Ustawienie tego parametru powoduje przyjęcie domyślnych wartości wielu innych parametrów, odpowiednich dla danej bazy danych.

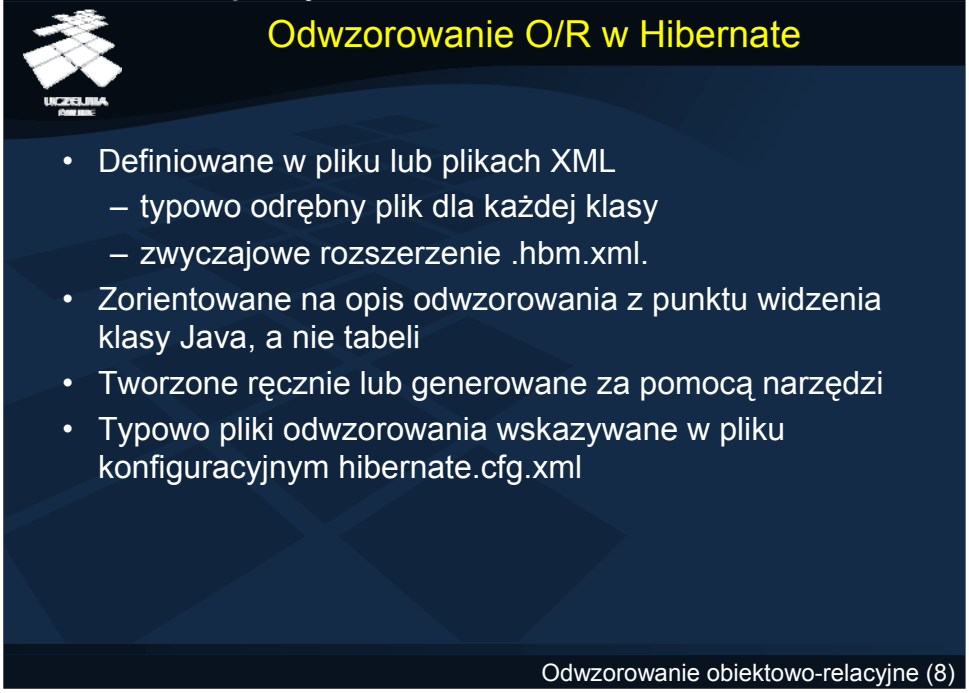
6. Elementy <mapping> wskazują pliki XML zawierające opis sposobu odwzorowania poszczególnych klas Java na tabele w relacyjnej bazie danych. Typowo dla każdej klasy tworzony jest odrębny plik z odwzorowaniem, posiadający rozszerzenie .hbm.xml.



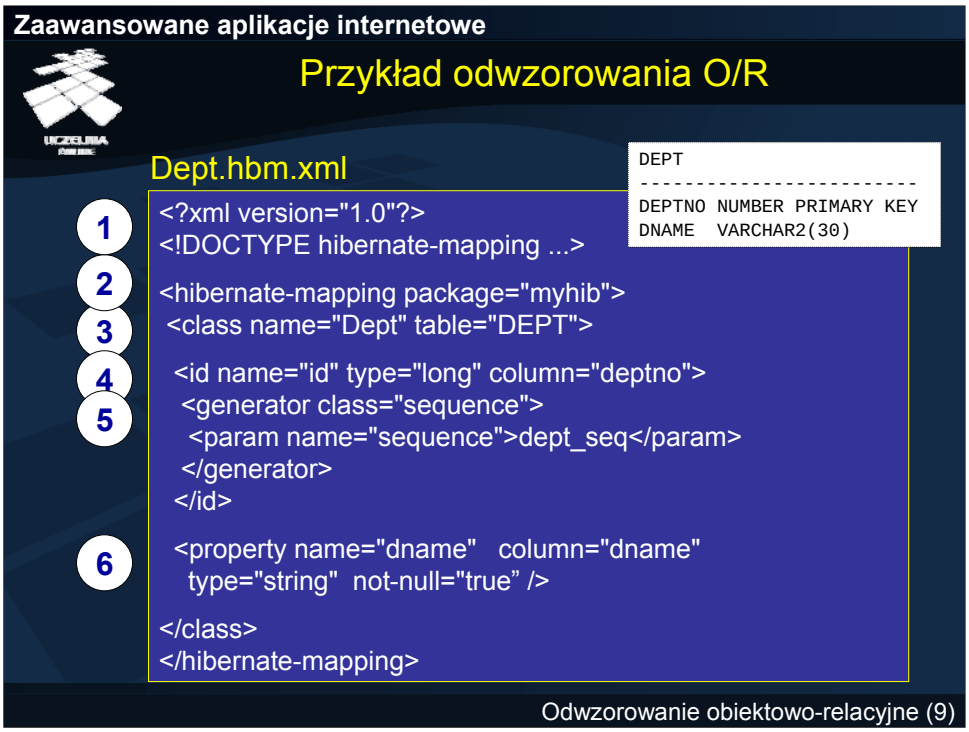
Trwałe klasy to klasy implementujące encje występujące w modelu danych aplikacji np. Departament, Klient, Faktura. Nie wszystkie instancje trwałej klasy muszą być trwałe.

Hibernate najlepiej działa z klasami spełniającymi reguły Plain Old Java Object (POJO). Klasy POJO dla Hibernate muszą posiadać metody set/get dla trwałych pól i bezargumentowy konstruktor. Mocno zalecane jest aby klasa posiadała wyróżniony identyfikator, najlepiej sztuczny w formie dodatkowego pola w klasie, typu nieprostego.

U dołu slajdu pokazany jest kod przykładowej klasy trwałej do reprezentacji informacji o departamentach. Klasa ma postać POJO, zawiera metody set/get dla pól i bezargumentowy domyślny konstruktor. Jedno z pól („id”) pełni funkcję sztucznego identyfikatora encji.



Odwzorowanie obiektowo-relacyjne w Hibernate jest definiowane w pliku lub plikach XML. Typowo odwzorowanie dla każdej trwałej klasy opisane jest w odrębnym pliku z rozszerzeniem .hbm.xml. Język do opisu odwzorowania jest zorientowany na opis odwzorowania z punktu widzenia klasy Java, a nie tabeli. Dokumenty opisujące odwzorowanie można tworzyć ręcznie lub korzystając z narzędzi. Istnieją generatory tworzące plik odwzorowania na podstawie klasy POJO, a także generujące klasy POJO i pliki odwzorowania dla istniejących tabel w bazie danych. Dokumenty opisujące odwzorowanie typowo są wskazywane w pliku konfiguracyjnym hibernate.cfg.xml, ale mogą też być wskazane programowo w aplikacji.



Slajd pokazuje przykładowy plik z opisem odwzorowania klasy Dept przedstawionej na jednym z poprzednich slajdów na tabelę w bazie danych Oracle o schemacie pokazanym w prawym górnym rogu slajdu. Znaczenie poszczególnych elementów pliku jest następujące:

1. Plik odwzorowania jest plikiem XML o strukturze określonej przez wskazany DTD.

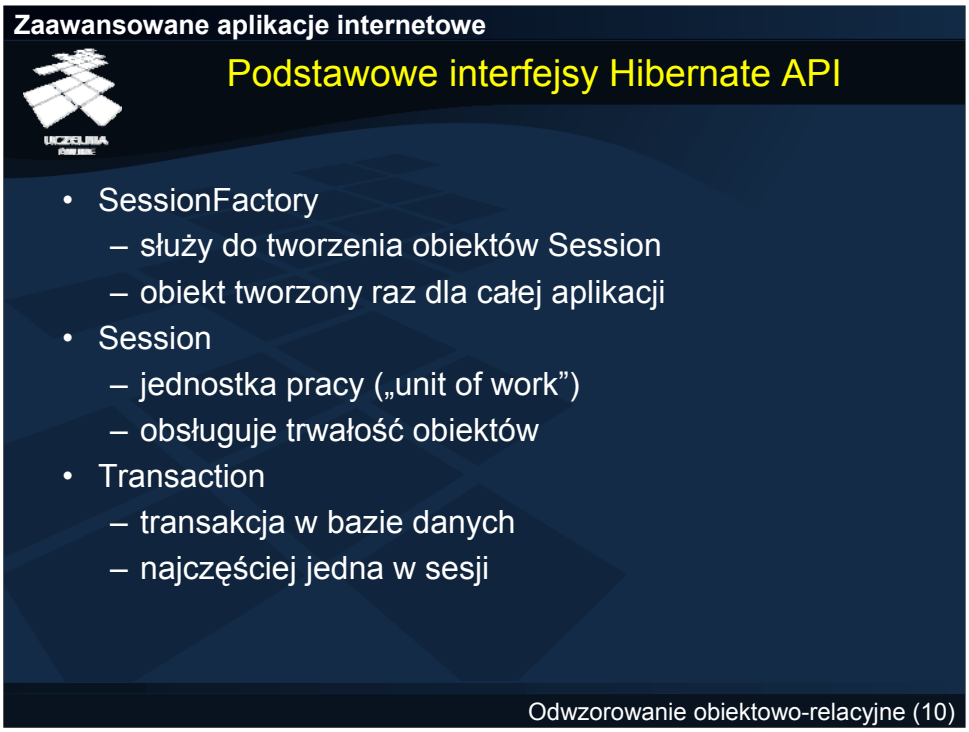
2. Odwzorowanie zawarte jest w elemencie głównym <hibernate-mapping>. Atrybut „package” podaje nazwę pakietu dla klas opisanych w pliku. Dzięki temu nie trzeba ich nazw w dalszych sekcjach pliku poprzedzać nazwą pakietu.

3. Odwzorowanie dla klasy Dept na tabelę DEPT.

4. Odwzorowanie identyfikatora „id” na kolumnę „deptno”. Do odwzorowania identyfikatora służy element <id>, dla pozostałych właściwości wykorzystywany jest element <property>. Atrybut „name” zawiera nazwę pola/właściwości w klasie, a atrybut „column” nazwę kolumny w tabeli bazy danych. Atrybut „type” przyjmuje wartość jednego z typów danych Hibernate, opisujących typ danych i sposób ich konwersji między językami Java i SQL.

5. Jako część odwzorowania identyfikatora należy podać sposób generacji jego wartości zagnieżdżonym elementem <generator>. W naszym przykładzie do generacji jest użyta sekwencja w bazie danych Oracle o nazwie „dept_seq”.

6. Odwzorowanie właściwości niebędącej identyfikatorem na kolumnę tabeli. Ponieważ nazwa właściwości w klasie i kolumny w tabeli są takie same, atrybut „column” można było pominąć.

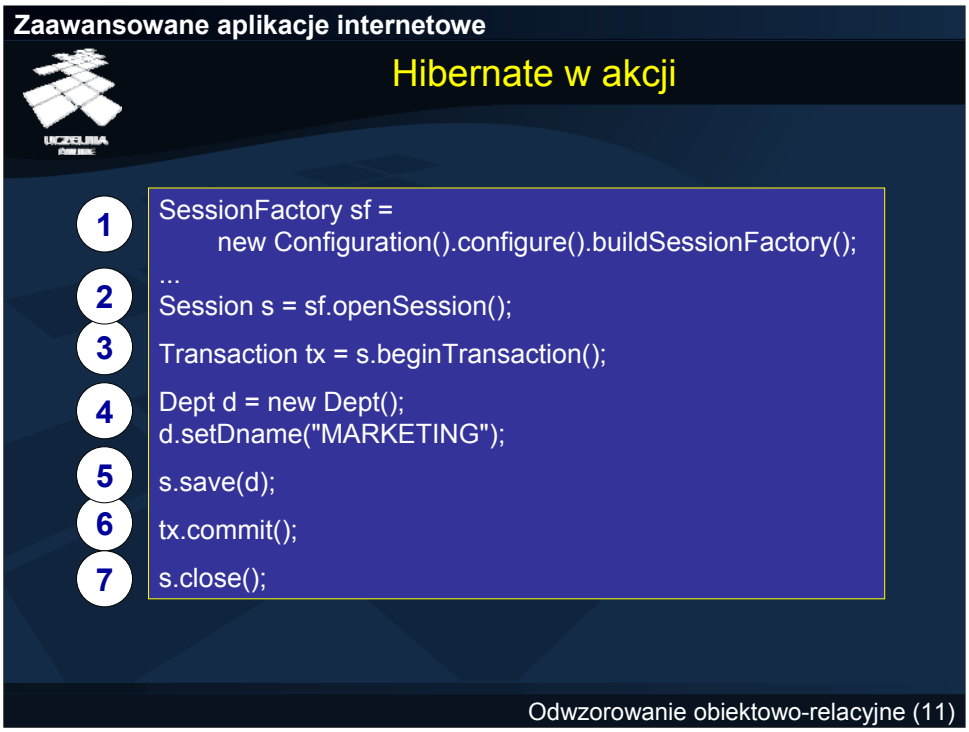


Trzy najważniejsze interfejsy interfejsu programistycznego Hibernate to SessionFactory, Session i Transaction.

Obiekt SessionFactory służy do tworzenia obiektów Session, poprzez które następnie realizowana jest komunikacja z bazą danych. Z założenia obiekt SessionFactory powinien być tworzony raz, na początku pracy aplikacji, gdyż jego tworzenie jest kosztowne, a może on być bezpiecznie współdzielony przez wiele wątków. Typowym rozwiązaniem jest tworzenie obiektu SessionFactory w klasie pomocniczej implementującej wzorzec projektowy singleton.

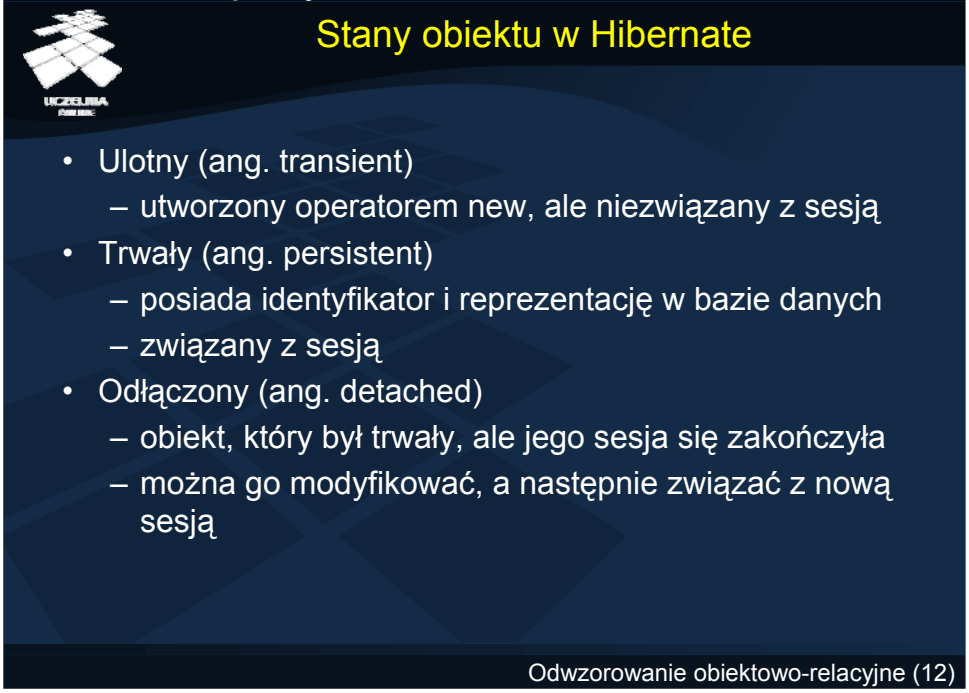
Obiekt Session reprezentuje jednostkę pracy („unit of work”). Obiekt Session jest tworzony dla jednego procesu, typowo na czas obsługi pojedynczego żądania w aplikacji WWW. Nie ma on nic wspólnego z sesjami HTTP. Metody obiektu Session służą do obsługi trwałości obiektów w aplikacji.

Obiekt Transaction reprezentuje transakcję w bazie danych, separując aplikację od konkretnej implementacji transakcji (JDBC/JTA). Typowo w sesji realizowana jest jedna transakcja, ale może być również wiele kolejnych transakcji w jednej sesji.



Przykładowy fragment kodu na slajdzie ilustruje sposób zapisu obiektu do bazy danych z wykorzystaniem Hibernate. Znaczenie poszczególnych instrukcji jest następujące:

1. Utworzenie obiektu SessionFactory (raz w aplikacji, typowo w klasie pomocniczej).
2. Otwarcie sesji – utworzenie obiektu Session poprzez obiekt SessionFactory.
3. Rozpoczęcie transakcji.
4. Utworzenie obiektu klasy Dept i ustawienie jego właściwości. Obiekt pozostaje ulotny do momentu związania go z sesją.
5. Uczynienie wskazanego obiektu trwałym. Operacja nie powoduje natychmiastowego wykonania poleceń JDBC w bazie danych. Zostaną one wykonane w momencie zatwierdzania transakcji lub przed wykonaniem zapytania do bazy danych. Istnieje możliwość wymuszenia zapisu danych sesji do bazy danych poprzez wywołanie metody flush() na rzecz obiektu Session.
6. Zatwierdzenie transakcji metodą commit() obiektu Transaction (do wycofania transakcji służy metoda rollback()).
7. Zamknięcie sesji.

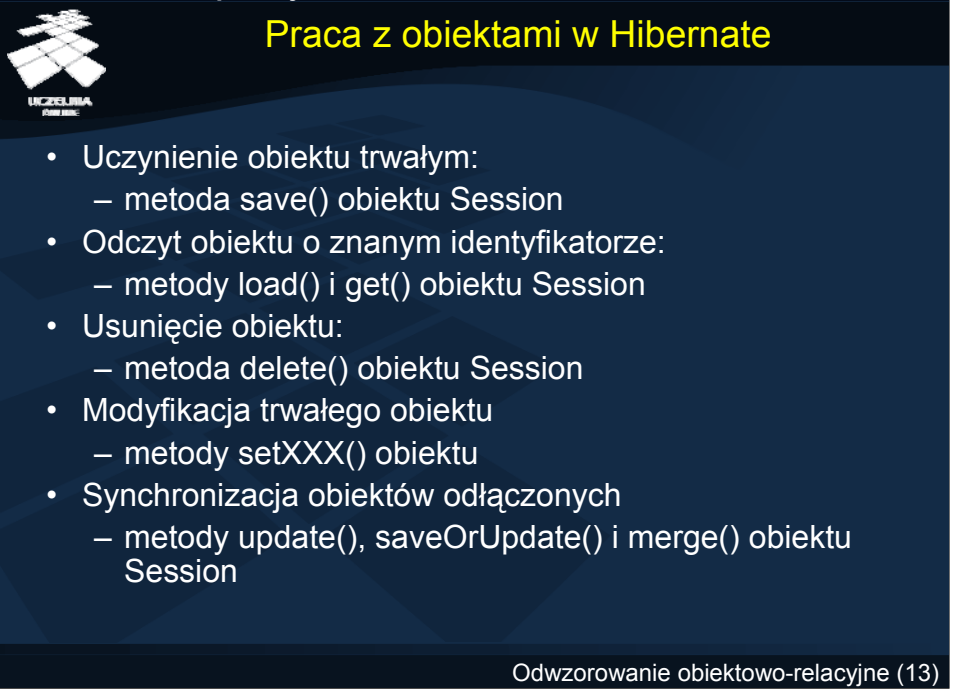


Obiekty aplikacji mogą z punktu widzenia Hibernate być w jednym z trzech stanów: ulotny, trwały lub odłączony.

Obiekt ulotny (ang. transient) to obiekt utworzony operatorem new, ale niezwiązany jeszcze z sesją.

Obiekt trwały (ang. persistent) to obiekt posiadający identyfikator i reprezentację w bazie danych, związany z sesją. Hibernate wykrywa zmiany dokonane na trwałych obiektach i synchronizuje ich stan z bazą danych.

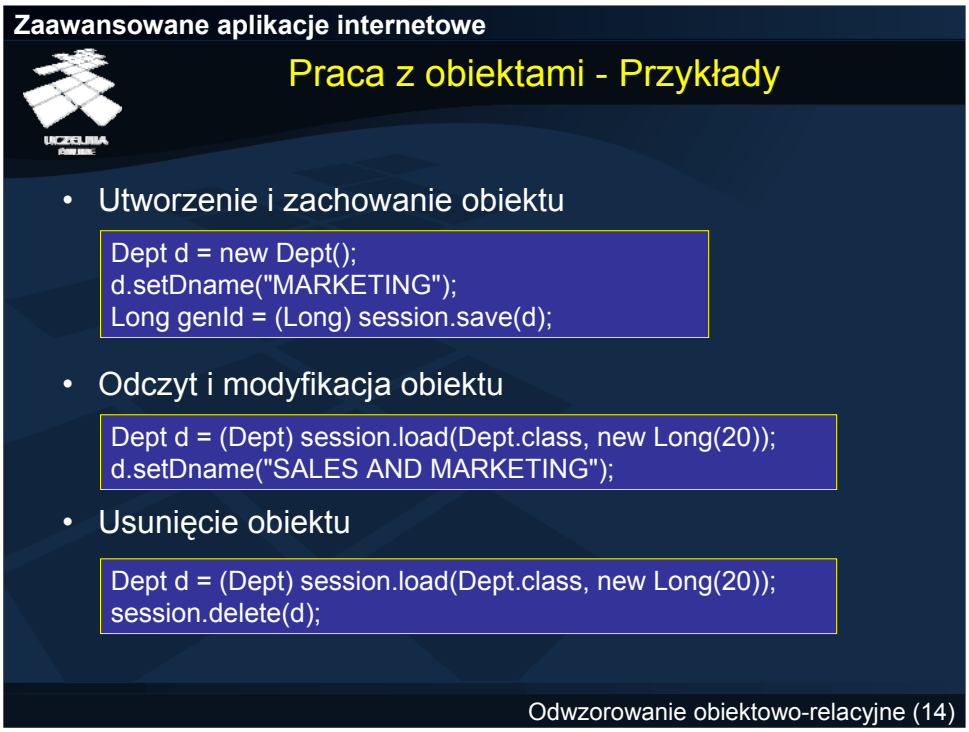
Obiekt odłączony (ang. detached) to obiekt, który był trwały, ale jego sesja się zakończyła. Obiekty odłączone można modyfikować, a następnie związać z nową sesją. Funkcjonalność ta jest przydatna gdy obiekt po odczycie z bazy danych jest przekazywany do warstwy interfejsu użytkownika i tam modyfikowany.



Obsługa trwałości obiektów w aplikacjach opartych o Hibernate polega głównie na wywoływaniu odpowiednich metod na rzecz obiektu Session. Do uczynienia obiektu trwałym służy metoda save() obiektu Session. Odczyt obiektu o znanym identyfikatorze umożliwiają bardzo podobne do siebie metody load() i get() obiektu Session. Do usuwania reprezentacji obiektu z bazy danych służy metoda delete() obiektu Session.

Modyfikacje trwałego obiektu dokonywane są metodami udostępnianymi przez sam obiekt bez pośrednictwa obiektu Session. Hibernate śledzi zmiany na obiektach trwałych i przeniesie je do bazy danych w momencie zatwierdzania transakcji lub wcześniej gdy zajdzie taka potrzeba w związku z wykonywaniem zapytań lub po wywołaniu w kodzie metody flush() na rzecz obiektu Session().

Do obsługi obiektów odłączonych obiekt Session udostępnia trzy metody umożliwiające różne sposoby synchronizacji odłączonego obiektu z bazą danych po rozpoczęciu nowej sesji: update(), saveOrUpdate() i merge().

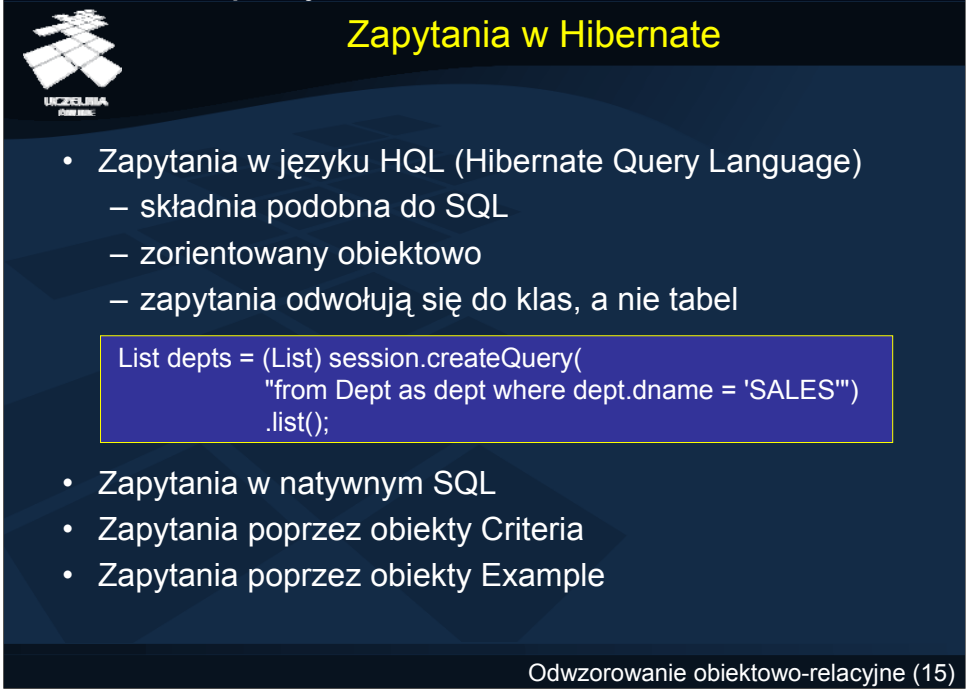


Przykłady na slajdzie ilustrują pracę z obiektami w Hibernate. Wszystkie przedstawione fragmenty kodu wykorzystują obiekt Session dostępny poprzez zmienną o nazwie „session”.

Pierwszy przykład pokazuje utworzenie i uczynienie obiektu trwałym. Do momentu przekazania referencji na obiekt trwałej klasy jako parametru metody save() obiektu Session obiekt jest ulotny. Metoda save() zwraca identyfikator obiektu.

Drugi przykład ilustruje odczyt obiektu poprzez jego identyfikator metodą load() obiektu Session, a następnie modyfikację stanu obiektu jedną z metod klasy Dept.

Trzeci przykład pokazuje operację usunięcia obiektu metodą delete() obiektu Session. Przed usunięciem obiektu należy uzyskać referencję do usuwanego obiektu.



Podstawowym sposobem odczytu danych z bazy danych w Hibernate są zapytania formułowane w języku HQL (Hibernate Query Language). Język ten pod względem składni bardzo przypomina SQL. Podstawowa struktura zapytania to SELECT-FROM-WHERE-GROUP BY-HAVING-ORDER BY, z tym że klauzula SELECT jest opcjonalna, gdy mają być odczytane całe obiekty. HQL w odróżnieniu od SQL jest zorientowany obiektowo i operuje na klasach języka Java, a nie tabelach relacyjnej bazy danych. HQL obsługuje dziedziczenie, polimorfizm i asocjacje między klasami.

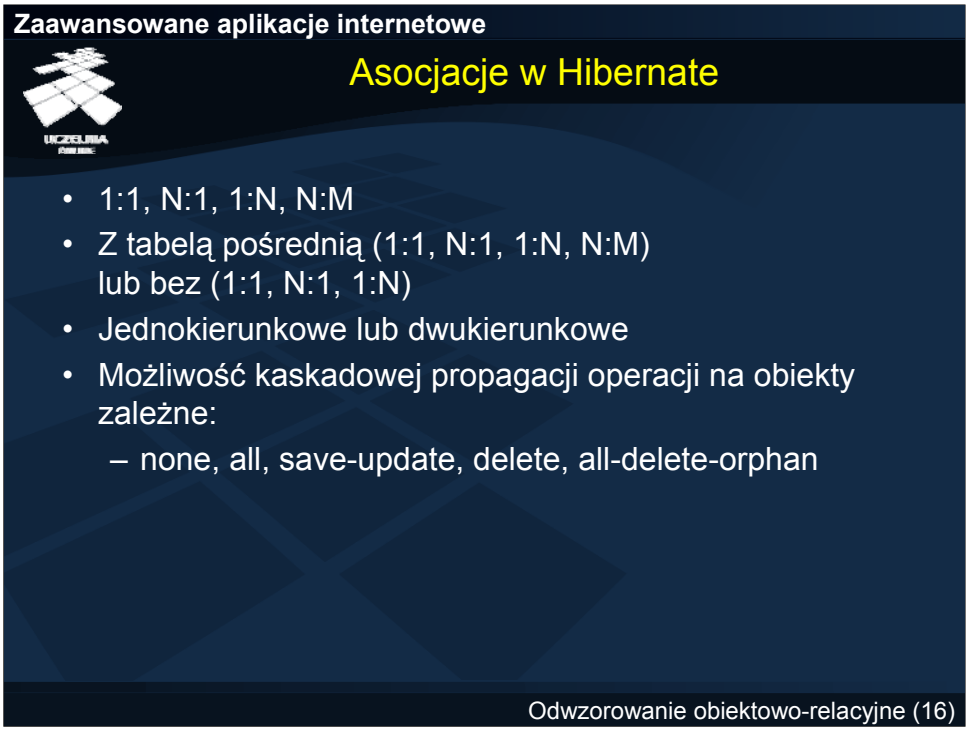
Na slajdzie pokazano przykład realizacji prostego zapytania w HQL. Zapytanie odczytuje wszystkie departamenty o nazwie „SALES”. W składni zapytania pominięta została klauzula SELECT, gdyż zwrócone mają być całe obiekty klasy Dept. Obiekt zapytania jest tworzony metodą createQuery() obiektu Session. Wyniki zapytania zostały zwrócone jako kolekcja typu List metodą list() obiektu Query.

Inne możliwości realizacji zapytań do bazy danych w Hibernate to:

• Zapytania w natywnym SQL, dające możliwość wykorzystania specyficznych dla danego systemu konstrukcji składniowych np. CONNECT w Oracle.

• Zapytania poprzez obiekty Criteria, umożliwiające budowę zapytań poprzez obiektowe API.

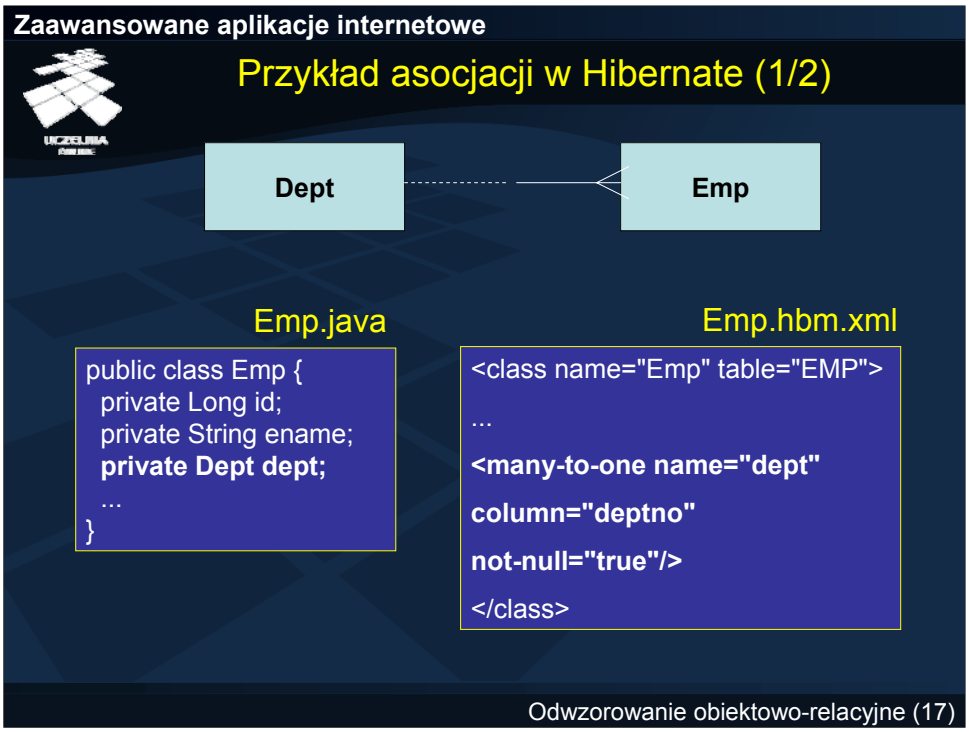
• Zapytania poprzez obiekty Example, umożliwiające wyszukiwanie danych w oparciu o przykładową instancję (mechanizm QBE – Query By Example).



Hibernate wspiera następujące typy asocjacji między obiektami klas trwałych: 1:1, N:1, 1:N i N:M. Asocjacje N:M zawsze wykorzystują tabelę pośrednią w bazie danych do reprezentacji powiązań. Asocjacje 1:1, N:1 i 1:N typowo są implementowane bez tabeli pośredniej, ale Hibernate umożliwia wykorzystanie tabeli pośredniej również dla nich.

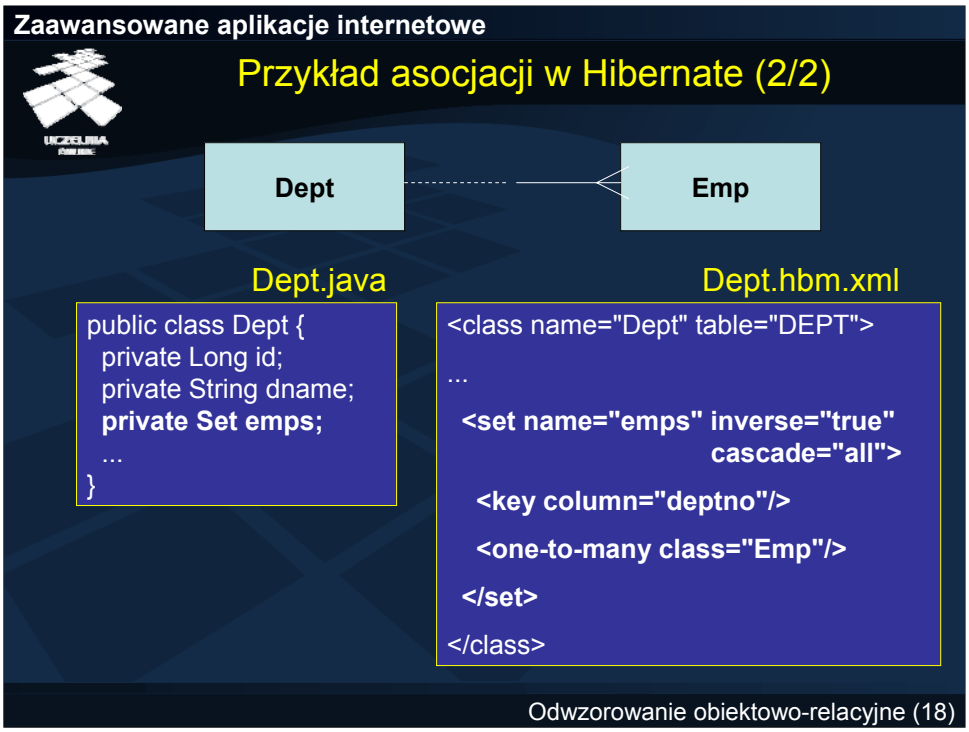
Asocjacje mogą być jednokierunkowe lub dwukierunkowe. Kierunkowość determinuje możliwość nawigacji między powiązanymi instancjami klas trwałych. Asocjacja dwukierunkowa jest definiowana jako para asocjacji jednokierunkowych, ze wskazaniem dla jednej z nich, że definiuje ona drugi kierunek nawigacji dla dwukierunkowej asocjacji.

Dla asocjacji istnieje możliwość zlecenia kaskadowej propagacji operacji na obiekty zależne. Domyślnie kaskadowa propagacja nie zachodzi. Kaskada jest często definiowana dla związków 1:N mających charakter związku kompozycji. Dostępne wartości atrybutu „cascade” opisującego asocjację to none (brak propagacji), all (propagacja wszystkich operacji), save-update (propagacja utrwalania i uaktualniania), delete (propagacja usuwania) i all-delete-orphan (propagacja wszystkich operacji i dodatkowo usuwanie obiektów zależnych, nie posiadających powiązania z obiektem nadrzędnym).

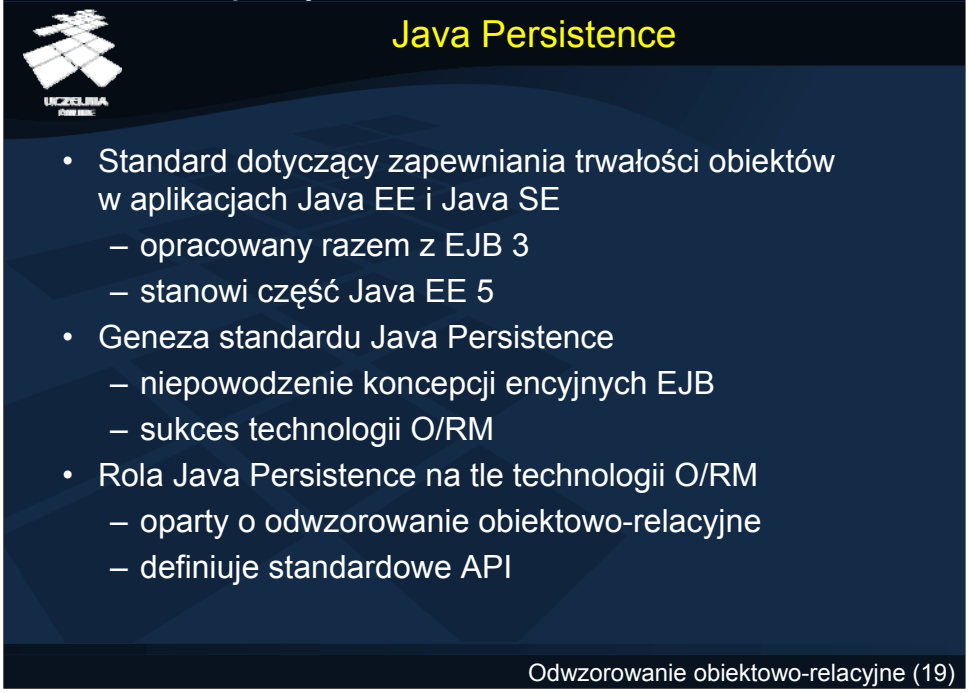


Ten i następny slajd pokazują przykład definicji dwukierunkowej asocjacji 1:N miedzy klasami trwałymi Dept (departament) i Emp (pracownik). Asocjacja jest dwukierunkowa, gdyż ma umożliwiać nawigację do danych departamentu dla danego pracownika jak i do zbioru przypisanych do departamentu pracowników dla danego departamentu.

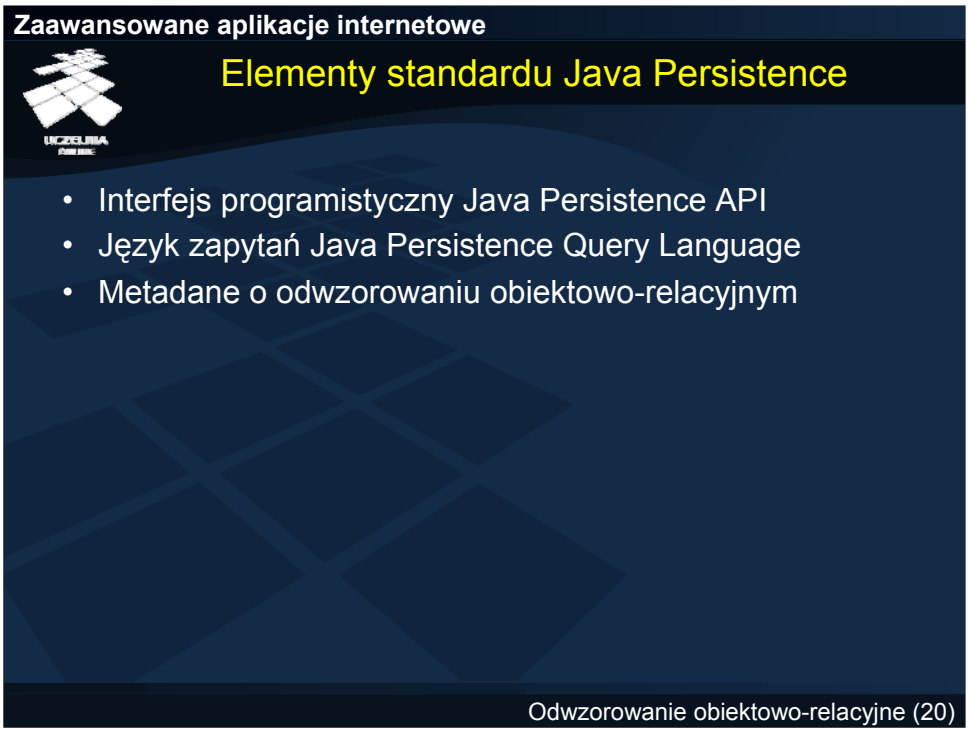
Na slajdzie pokazano definicję klasy Emp (z pominięciem metod set/get) i plik odwzorowania klasy Emp na tabelę EMP w bazie danych. W kodzie klasy asocjację reprezentuje właściwość „dept” typu Dept. Odwzorowanie dla właściwości „dept” w pliku Emp.hbm.xml ma postać elementu <many-to-one>.



Niniejszy slajd pokazuje definicję drugiego kierunku przykładowej asocjacji – od strony klasy Dept. W kodzie klasy Dept asocjację reprezentuje właściwość „emps” typu Set. Hibernate obsługuje różne typy kolekcji, ale Set jest wykorzystywany najczęściej. Odwzorowanie dla właściwości „emps” w pliku Dept.hbm.xml ma postać elementu <set>. Jego atrybut „inverse” wskazuje, że asocjacja od strony klasy Dept jest drugim kierunkiem asocjacji dwukierunkowej. Atrybut „cascade” ma wartość „all”, co oznacza, że wszystkie operacje wykonane na instancji klasy Dept będą propagowane na związane z nią instancje klasy Emp. Zagnieżdżony w elemencie <set> element <key> wskazuje nazwę kolumny klucza obcego w tabeli, na którą odwzorowana jest powiązana klasa, a element <one-to-many> wskazuje nazwę tej klasy.



Java Persistence to nowy, opracowany razem z EJB 3.0 standard zapewniania trwałości obiektów w aplikacjach Java EE i Java SE, stanowiący część specyfikacji Java EE od wersji 5.0. Został on opracowany razem z EJB 3.0 w odpowiedzi na niepowodzenie lansowanej do tej pory koncepcji encyjnych EJB i niewątpliwy sukces technologii odwzorowania obiektowo-relacyjnego takich jak Hibernate czy Oracle Toplink. Technologie te, mimo że oparte o te same idee, różnią się jeśli chodzi o API. Standard Java Persistence jest oparty o odwzorowanie obiektowo-relacyjne i definiuje standardowe API do obsługi trwałości obiektów.

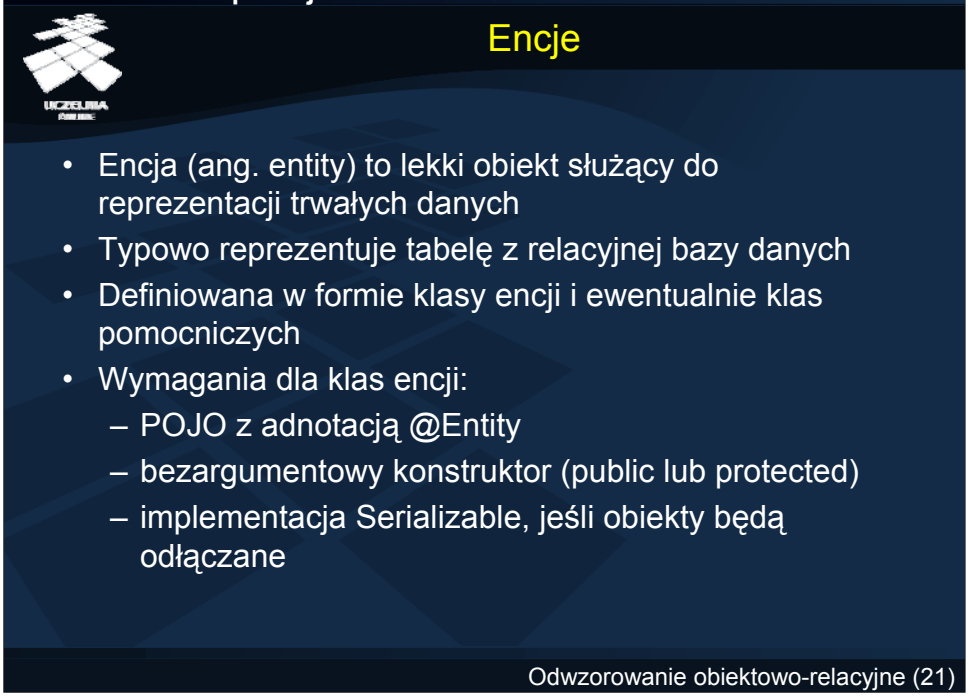


Elementy standardu Java Persistence to:

1. Interfejs programistyczny Java Persistence API, obejmujący interfejs do zarządcy trwałości EntityManager;

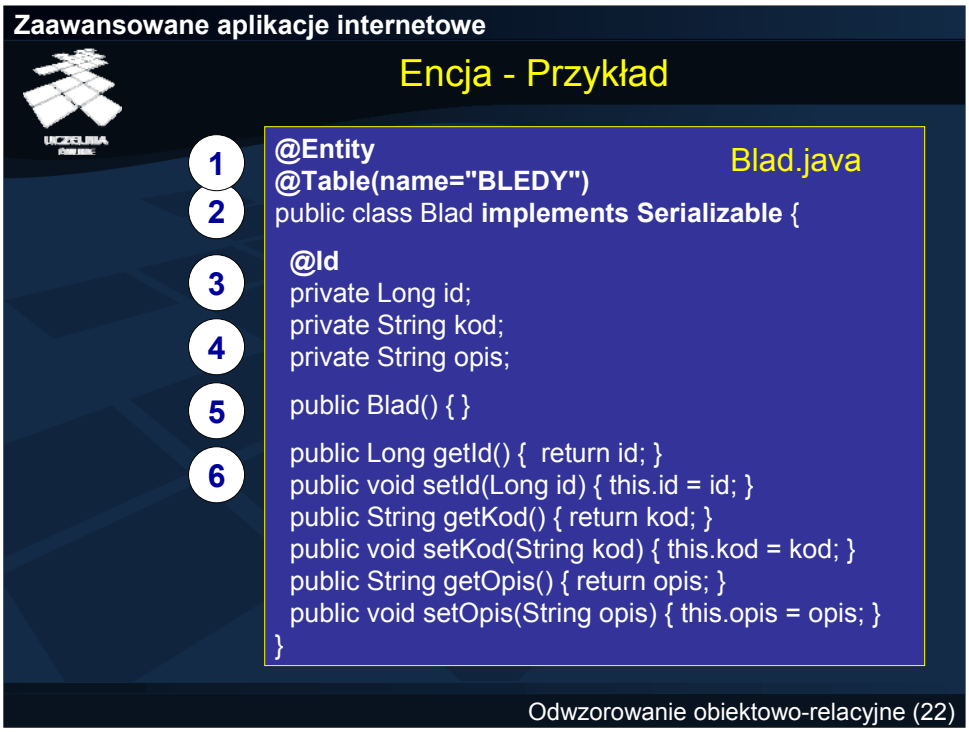
2. Język zapytań Java Persistence Query Language (JPQL), o składni przypominającej SQL, umożliwiający tworzenie przenaszalnych zapytań.

3. Metadane o odwzorowaniu obiektowo-relacyjnym, najczęściej umieszczone w kodzie w formie adnotacji, z możliwością dodatkowej konfiguracji w środowisku produkcyjnym poprzez XML-owe pliki konfiguracyjne.



Encja (ang. entity) to lekki obiekt służący do reprezentacji trwałych danych. Typowo encja reprezentuje tabelę z relacyjnej bazy danych, ale istnieje również możliwość odwzorowania encji na kilka tabel. Encja definiowana jest w formie klasy encji. Niekiedy klasa encji jest uzupełniana o klasy pomocnicze np. klasę definiującą strukturę złożonego klucza głównego.

Klasa encji to zwykła klasa POJO (Plain Old Java Object), spełniająca reguły JavaBeans tj. dostęp do pól klasy tylko przez metody klasy setXXX()/getXXX() i bezargumentowy publiczny lub zabezpieczony konstruktor. Klasa encji nie może być final. Klasa encji nie musi dziedziczyć z żadnej konkretnej klasy ani implementować konkretnego interfejsu. W praktyce klasy encji są tworzone jako implementujące interfejs Serializable, gdyż jest to wymagane gdy obiekty klasy mają być odłączane od kontekstu trwałości np. gdy są parametrami metod zdalnego interfejsu EJB.



Slajd przedstawia przykładową klasę encji do reprezentacji informacji o błędach. Znaczenie wyróżnionych fragmentów kodu klasy jest następujące:

1. Aby klasa była klasą encji musi być oznaczona adnotacją @Entity. Domyślnie klasa jest odwzorowywana na tabelę o nazwie takiej samej jak nazwa klasy. Adnotacja @Table zmienia to odwzorowanie, wskazując jawnie nazwę tabeli. Jest to szczególnie przydatne gdy schemat bazy danych już istnieje.

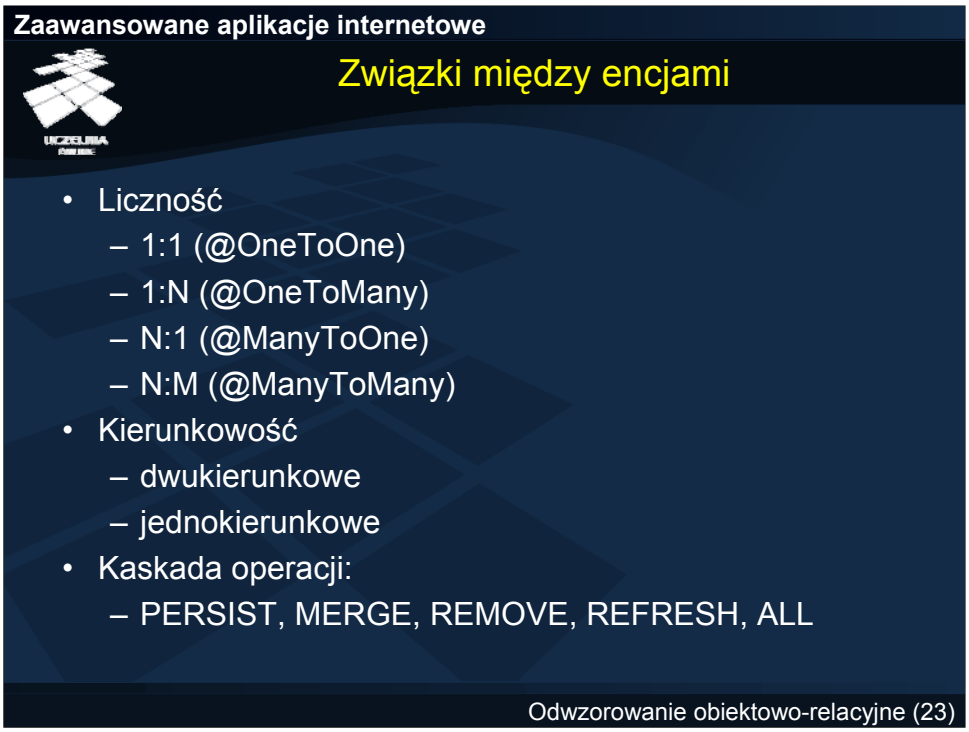
2. Klasa implementuje interfejs Serializable, co jest typowe dla klas encji, choć nie zawsze konieczne.

3. Pole „id” zostało wskazane adnotacją @Id jako klucz główny dla encji. Każda encja musi posiadać klucz główny. Gdy, tak jak w przykładzie, obejmuje on jedno pole standardowego typu języka Java, nie jest konieczne definiowanie klasy pomocniczej.

4. Pozostałe pola w klasie, w tym wypadku reprezentujące kod i opis błędu.

5. Bezargumentowy konstruktor o odpowiedniej widzialności, wymagany w klasie encji.

6. Metody setXXX()/getXXX() do ustawiania i odczytu poszczególnych właściwości encji.



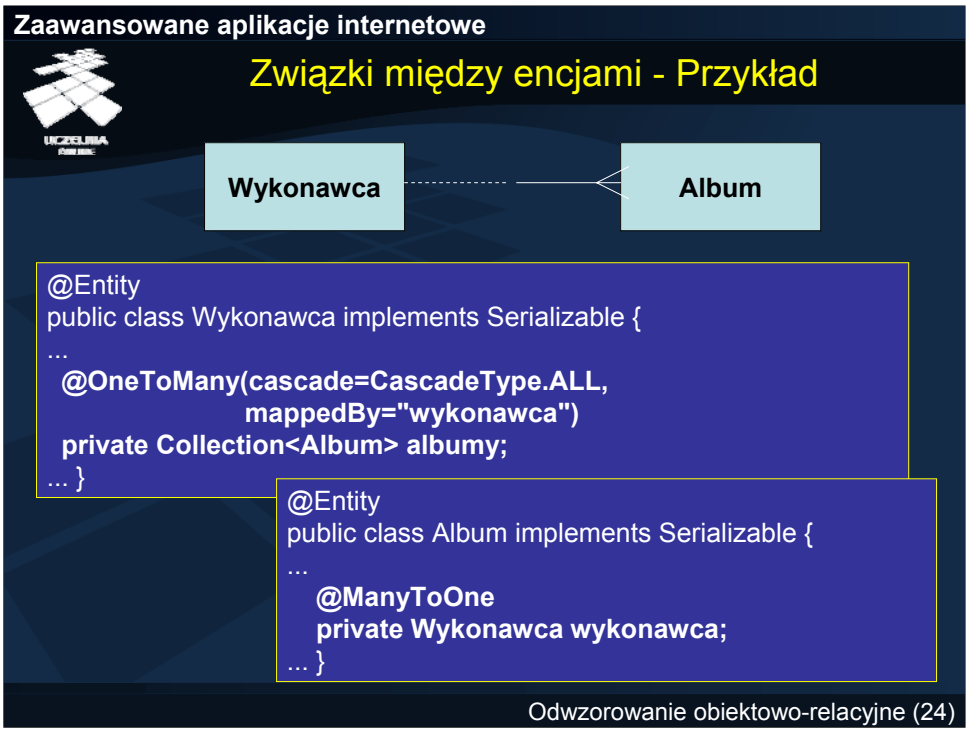
Typowo model biznesowy aplikacji obejmuje wiele encji, między którymi występują powiązania. Powiązania te na poziomie relacyjnej bazy danych są reprezentowane przez klucze obce.

Związek między dwoma encjami charakteryzuje liczba powiązanych wystąpień encji dla każdego końca związku, kierunkowość nawigacji między wystąpieniami encji poprzez związek i kaskadowa propagacja operacji na wystąpienia powiązanych encji.

Java Persistence umożliwia definiowanie związków 1:1 (one-to-one), 1:N (one-to-many), N:1 (many-to-one) i N:M (many-to-many). Związek jest reprezentowany w klasie encji przez pole lub właściwość oznaczone odpowiadającą liczności związku adnotacją.

Związek między dwoma encjami może być dwukierunkowy lub jednokierunkowy. Kierunkowość determinuje możliwość nawigacji między powiązanymi instancjami encji. W modelu relacyjnym do nawigacji między wierszami dwóch powiązanych relacji wystarczy obecność klucza obcego w schemacie jednej z nich. Na poziomie klas języka Java, w celu umożliwienia nawigacji w obie strony między instancjami dwóch klas, należy w obu klasach zawrzeć pole lub właściwość wskazującą instancję (lub kolekcję instancji) drugiej klasy. Związek dwukierunkowy jest definiowany jako dwa związki jednokierunkowe.

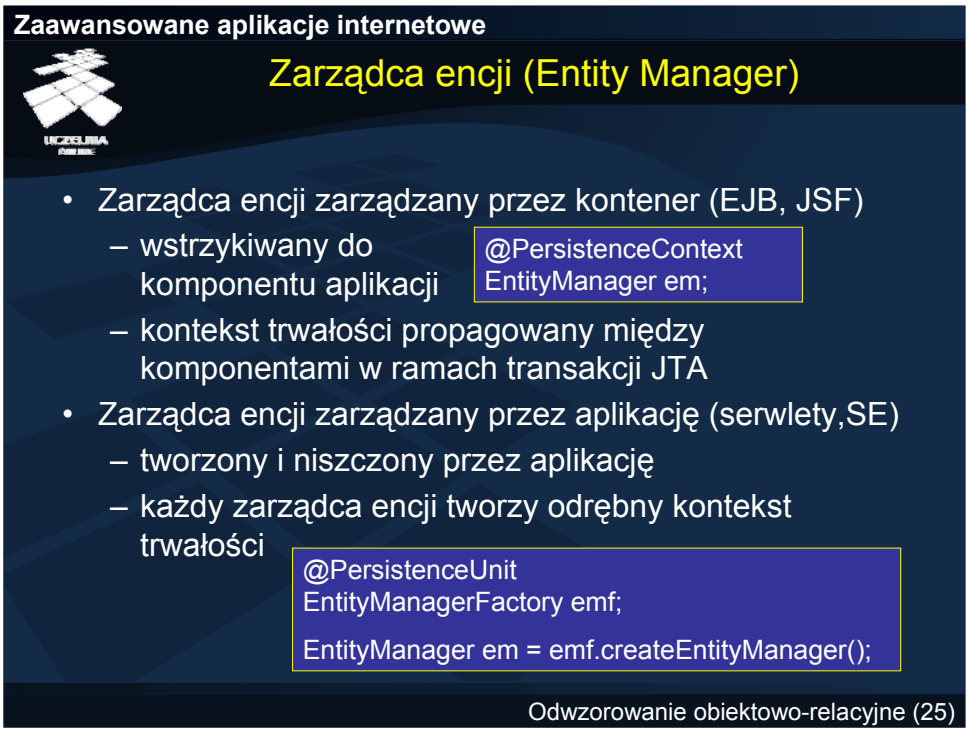
Oprócz liczności i kierunkowości, definicja związku może zawierać informację o kaskadowym propagowaniu operacji utrwalania, modyfikacji, usuwania i odświeżania z wystąpień encji nadrzędnej do wystąpień encji podrzędnej. Kaskadowość jest charakterystyczna dla związków 1:N, które mają charakter związku kompozycji.



Slajd pokazuje sposób definicji dwukierunkowego związku typu 1:N na przykładzie encji Wykonawca i Album. Związek jest dwukierunkowy, aby możliwe w aplikacji było zarówno odczytanie wszystkich albumów danego wykonawcy jak i wykonawcy dla danego albumu. Z wykonawcą związanych jest wiele albumów, stąd pole reprezentujące albumy wykonawcy jest typu kolekcji i jest oznaczone adnotacją @OneToMany. Album posiada jednego wykonawcę, więc pole reprezentujące związek po stronie encji Album jest typu Wykonawca i posiada adnotację @ManyToOne. Dla związku dwukierunkowego konieczne dla jednej strony związku jest wskazanie pola (lub właściwości) w drugiej encji reprezentującego drugi kierunek związku za pomocą elementu „mappedBy” adnotacji @OneToOne, @OneToMany lub @ManyToMany. W naszym przykładzie po stronie encji Wykonawca element „mappedBy” adnotacji @OneToMany wskazuje, że związek ten definiuje drugi kierunek nawigacji dla związku @ManyToOne reprezentowanego przez pole „wykonawca” w encji Album.

Związek @OneToMany po stronie encji Wykonawca posiada wartość CascadeType.ALL elementu „cascade”. Oznacza to, że wszystkie operacje na instancji encji Wykonawca mają powodować wykonanie takiej samej operacji dla powiązanych z nią instancji encji Album. Domyślnie kaskadowa propagacja operacji nie zachodzi. Inne stałe CascadeType, wskazujące poszczególne operacje to MERGE, PERSIST, REFRESH, REMOVE.

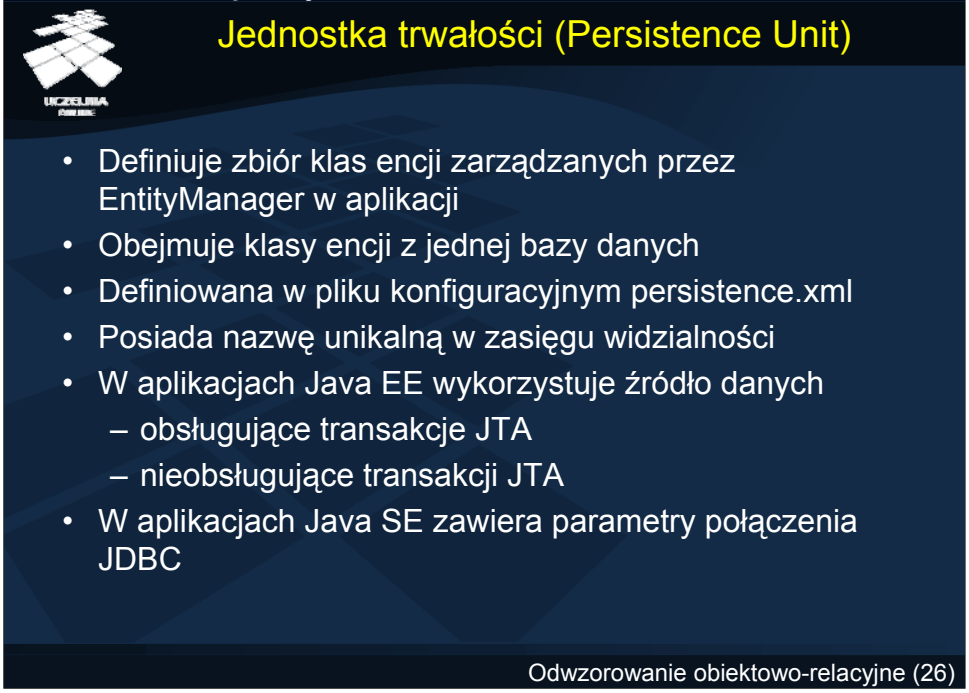
W przypadku związków dwukierunkowych należy pamiętać, że w celu powiązania instancji dwóch encji ze sobą należy dokonać powiązania dla obu stron związku. W naszym przykładzie, aby związać album z wykonawcą należałoby przypisać referencję do wykonawcy w obiekcie Album i dodać referencję do albumu do kolekcji albumów w obiekcie Wykonawca.



Zarządcy encji mogą być zarządzani przez kontener, co jest dostępne dla komponentów EJB i komponentów managed bean w JSF, lub zarządzani przez aplikację, co jest wykorzystywane w serwletach i aplikacjach Java SE.

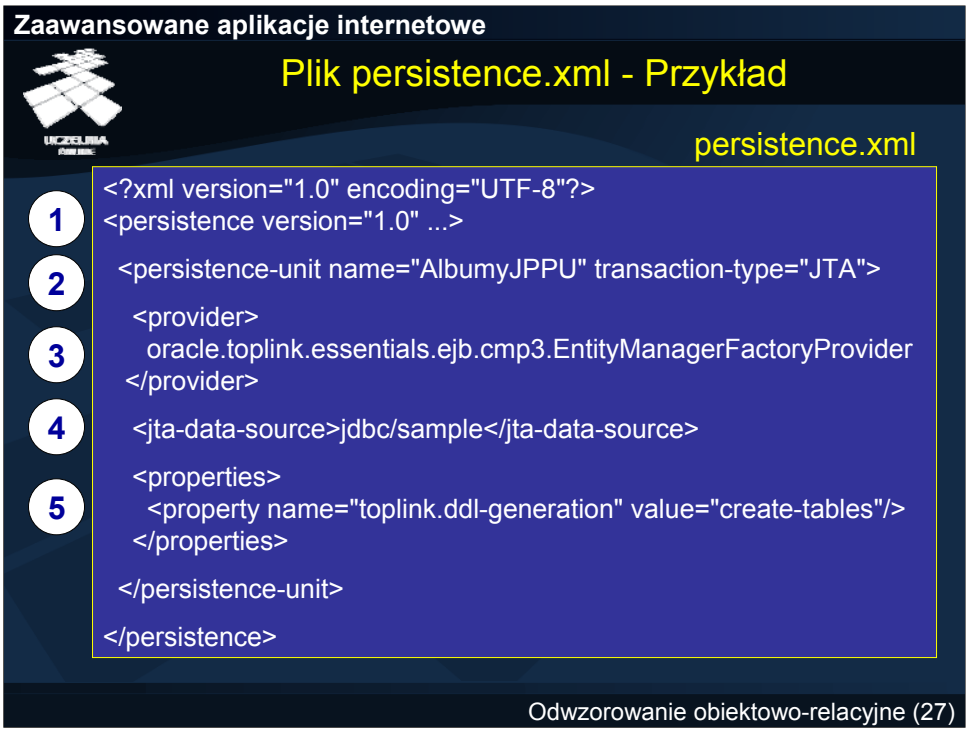
Zarządca encji zarządzany przez kontener jest wstrzykiwany do komponentu aplikacji adnotacją @PersistenceContext. W tym wypadku poszczególne komponenty aplikacji mają dostęp do tego samego kontekstu trwałości w ramach realizacji pojedynczej transakcji JTA.

Zarządca encji zarządzany przez aplikację jest tworzony i niszczony przez aplikację za pośrednictwem obiektu EntityManagerFactory wstrzykiwanego do komponentu adnotacją @PersistenceUnit lub tworzonego statyczną metodą createEntityManagerFactory klasy Persistence. W tym wypadku każdy utworzony zarządca encji tworzy odrębny kontekst trwałości.



Jednostka trwałości (Persistence Unit) definiuje zbiór klas encji zarządzanych przez EntityManager w aplikacji. Zbiór klas encji w ramach jednej jednostki trwałości reprezentuje dane z jednej bazy danych. Jednostka trwałości jest definiowana w pliku konfiguracyjnym persistence.xml. Jeden plik persistence.xml może zawierać definicje kilku jednostek trwałości. Każda jednostka trwałości musi posiadać nazwę unikalną w zasięgu widzialności jednostki trwałości. Nazwa ta wykorzystywana jest np. w adnotacji wstrzykującej obiekt EntityManagerFactory w przypadku gdy w pliku persistence.xml zdefiniowano więcej niż jedną jednostkę trwałości. Zasięg jednostki trwałości wyznacza tzw. korzeń jednostki trwałości czyli katalog zawierający podkatalog META-INF z plikiem persistence.xml.

W aplikacjach Java EE jednostka trwałości wykorzystuje źródło danych (obsługujące lub nieobsługujące transakcji JTA). W aplikacjach Java SE z jednostką trwałości związane są parametry połączenia JDBC.



Slajd pokazuje przykład pliku persistence.xml dla aplikacji Java EE. Znaczenie wyróżnionych elementów pliku jest następujące:

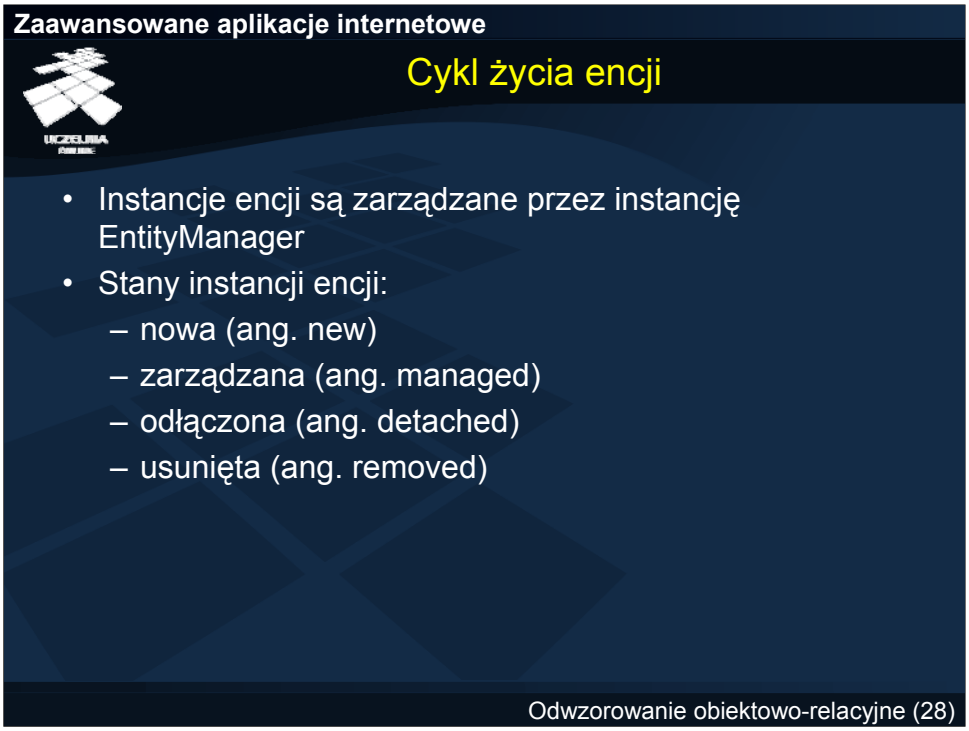
1. Treść pliku zawarta jest w elemencie <persistence> zawierającym deklaracje przestrzeni nazw pominięte na slajdzie.

2. Definicja jednostki trwałości ze wskazaniem jej nazwy i typu transakcji, które mają być w jej ramach realizowane.

3. Wskazanie konkretnego dostawcy usług trwałości. W tym wypadku jest to Oracle Toplink – referencyjna implementacja standardu Java Persistence (najpoważniejszą alternatywą jest Hibernate).

4. Nazwa JNDI źródła danych dla definiowanej jednostki trwałości.

5. Parametry konfiguracyjne dla wykorzystywanego dostawcy usług trwałości. W naszym przykładzie jeden parametr oznaczający, że Oracle Toplink ma utworzyć tabele w bazie danych jeśli nie będą istniały w momencie uruchomienia aplikacji.



Zarządzanie instancjami encji w kodzie aplikacji jest realizowane poprzez instancję zarządcy encji (EntityManager).

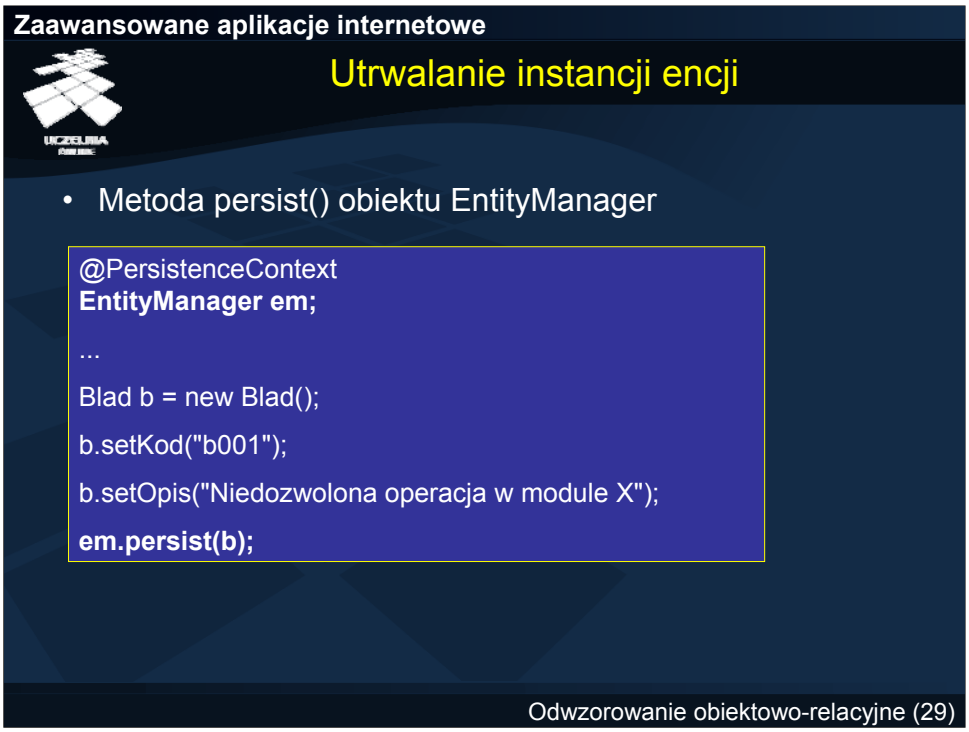
Instancja encji może znajdować się w jednym z czterech stanów:

• nowa (ang. new) – nieposiadająca trwałej tożsamości i niezwiązana jeszcze z kontekstem trwałości;

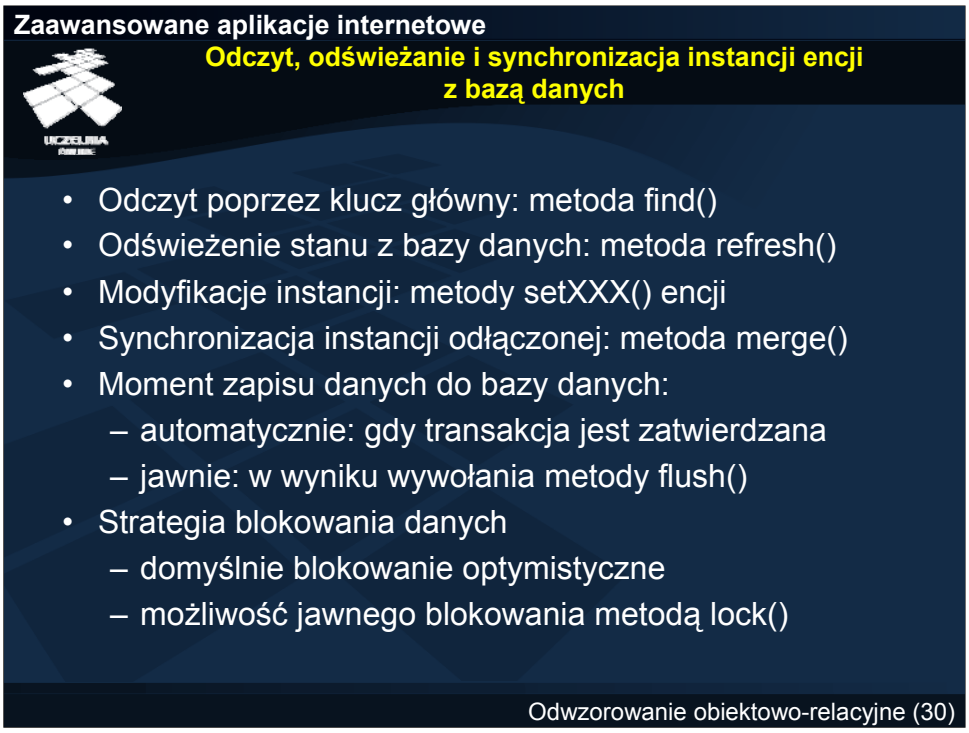
• zarządzana (ang. managed) - posiadająca trwałą tożsamość i związana z kontekstem trwałości;

• odłączona (ang. detached) - posiadająca trwałą tożsamość, a niezwiązana w danym momencie z kontekstem trwałości;

• usunięta (ang. removed) - posiadająca trwałą tożsamość, związana z kontekstem trwałości i zaszeregowana do usunięcia z bazy danych.



Do „utrwalania” instancji encji, czyli ich zachowania w bazie danych, służy metoda persist() obiektu EntityManager. W przykładzie na slajdzie obiekt EntityManager został wstrzyknięty adnotacją @PersistenceContext, co jest stosowane w komponentach EJB. Instancja encji jest tworzona jak normalny obiekt w programie Java, a następnie przekazywana jako parametr metody persist() obiektu EntityManager. Wywołanie tej metody spowoduje zapisanie danych instancji encji w bazie danych w momencie zatwierdzania transakcji. Jeśli utrwalana instancja encji jest powiązana związkiem z inną instancją encji i związek posiada własność kaskady PERSIST lub ALL, również związana instancja encji zostanie automatycznie zapisana w bazie danych.



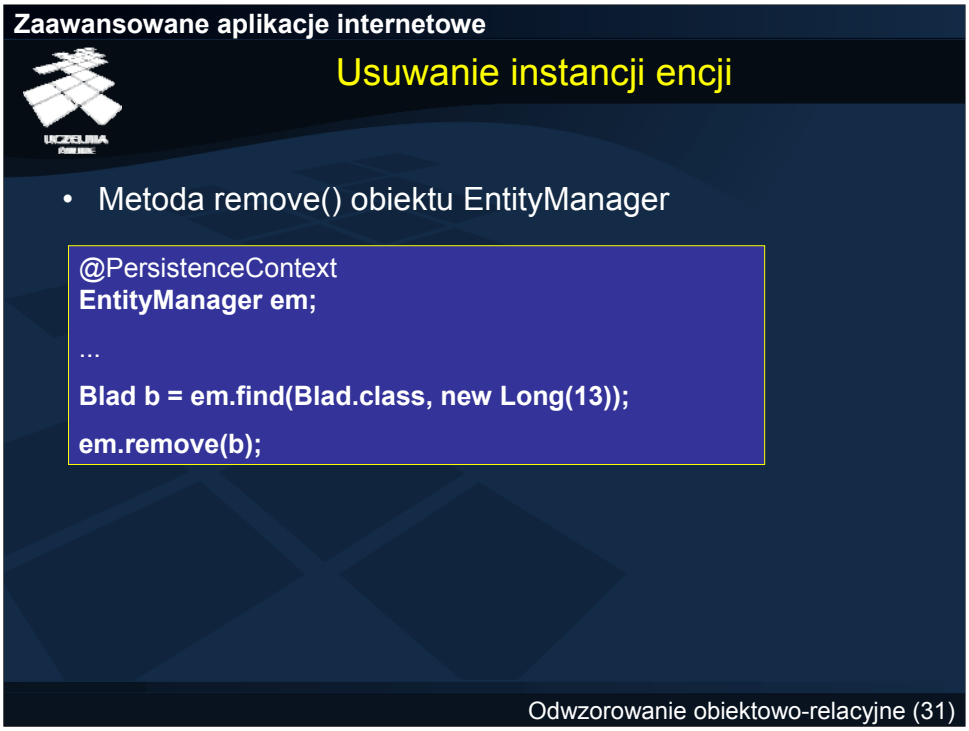
Do odczytu instancji encji poprzez klucz główny służy metoda find() obiektu EntityManager. Inna możliwość odczytu danych z bazy danych to realizacja zapytań, co będzie omówione na kolejnych slajdach.

Po odczytaniu instancji encji z bazy danych można odświeżyć jej stan poprzez ponowny odczyt z bazy danych metodą refresh() obiektu EntityManager. Operacja refresh() powoduje anulowanie zmian dokonanych na instancji w pamięci. Jeśli odświeżana instancja encji jest powiązana związkiem z inną instancją encji i związek posiada własność kaskady REFRESH lub ALL, również związana instancja encji zostanie automatycznie odświeżona poprzez odczyt z bazy danych.

Obiekt EntityManager nie posiada metod do modyfikacji instancji encji. Modyfikacje te są dokonywane metodami setXXX() encji. Jeśli instancja encji była modyfikowana w stanie odłączonym, np. po przekazaniu jej do innej warstwy aplikacji, do synchronizacji jej stanu z bazą danych służy metoda merge() obiektu EntityManager. Jeśli synchronizowana instancja encji jest powiązana związkiem z inną instancją encji i związek posiada własność kaskady MERGE lub ALL, również związana instancja encji zostanie automatycznie zsynchronizowana z bazą danych.

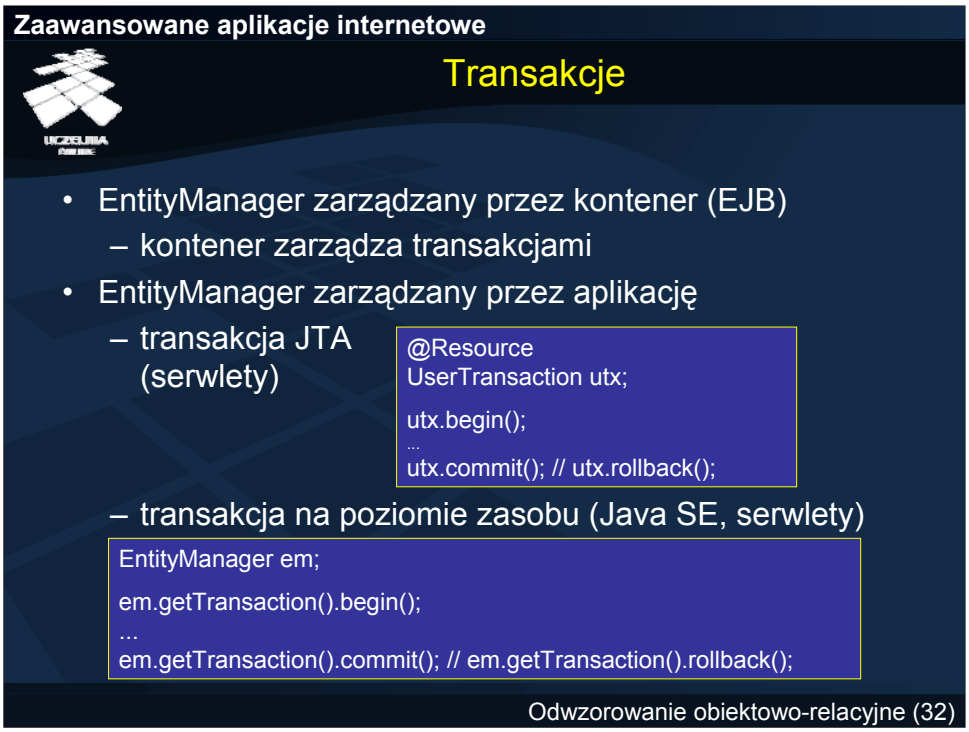
Domyślnie dokonane zmiany są zapisywane w bazie danych gdy transakcja jest zatwierdzana. Gdy aplikacja tego wymaga, gdyż np. dalszy kod polega na fizycznej obecności zmodyfikowanych danych w bazie danych, można wymusić wcześniejszy zapis danych do bazy danych metodą flush() obiektu EntityManager. Jeśli zapisywana instancja encji jest powiązana związkiem z inną instancją encji i związek posiada własność kaskady PERSIST lub ALL, również związana instancja encji zostanie automatycznie zapisana w bazie danych.

Java Persistence domyślnie stosuje blokowanie optymistyczne. Istnieje możliwość jawnego zakładania blokad do odczytu i zapisu metodą lock() obiektu EntityManager.



Do usuwania instancji encji z bazy danych służy metoda remove() obiektu EntityManager. W celu usunięcia instancji encji należy wcześniej uzyskać do niej referencję. Po wywołaniu metody remove() instancja encji pozostaje dostępna w pamięci jako obiekt nietrwały. Jeśli usuwana instancja encji jest powiązana związkiem z inną instancją encji i związek posiada własność kaskady REMOVE lub ALL, również związana instancja encji zostanie automatycznie usunięta z bazy danych.

Przykład na slajdzie pokazuje wyszukanie instancji encji poprzez klucz główny metodą find(), a następnie jej usunięcie metodą remove(). Parametrami metody find() są obiekt reprezentujący klasę encji i obiekt z wartością klucza głównego. Klucz główny przykładowej klasy encji Blad jest typu Long.



W komponentach EJB wykorzystujących zarządcę encji zarządzanego przez kontener, transakcjami zarządza kontener.

W przypadku aplikacji WWW opartych o serwlety i JSP, wykorzystujących zarządcę encji zarządzanego przez aplikację, możliwe jest wykorzystanie transakcji JTA lub transakcji obsługiwanych poprzez EntityManager, określanych jako transakcje na poziomie zasobu. Aplikacje Java SE wykorzystują transakcje na poziomie zasobu.

Granice transakcji na poziomie zasobu wyznaczane są metodami begin(), commit() i rollback() obiektu EntityTransaction uzyskanego poprzez wywołanie metody getTransaction() na rzecz obiektu EntityManager.

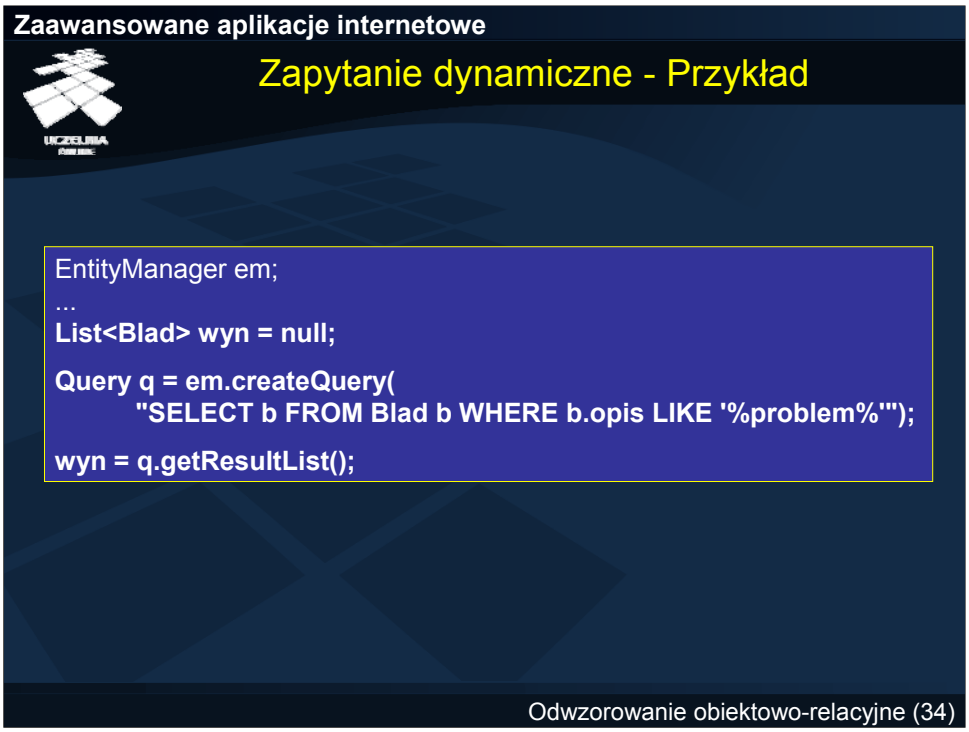
Zapytania do bazy danych

- Rodzaje zapytań (metody obiektu EntityManager)
  - dynamiczne w JPQL - createQuery()
  - dynamiczne natywne - createNativeQuery()
  - nazwane (JPQL lub natywne) - createNamedQuery()
- Parametryzacja zapytań
  - nazwane (np. :kodBledu)
  - pozycyjne (np. ?1)
- Wykonanie zapytania (metody obiektu Query)
  - getResultList(), getSingleResult()
  - executeUpdate()

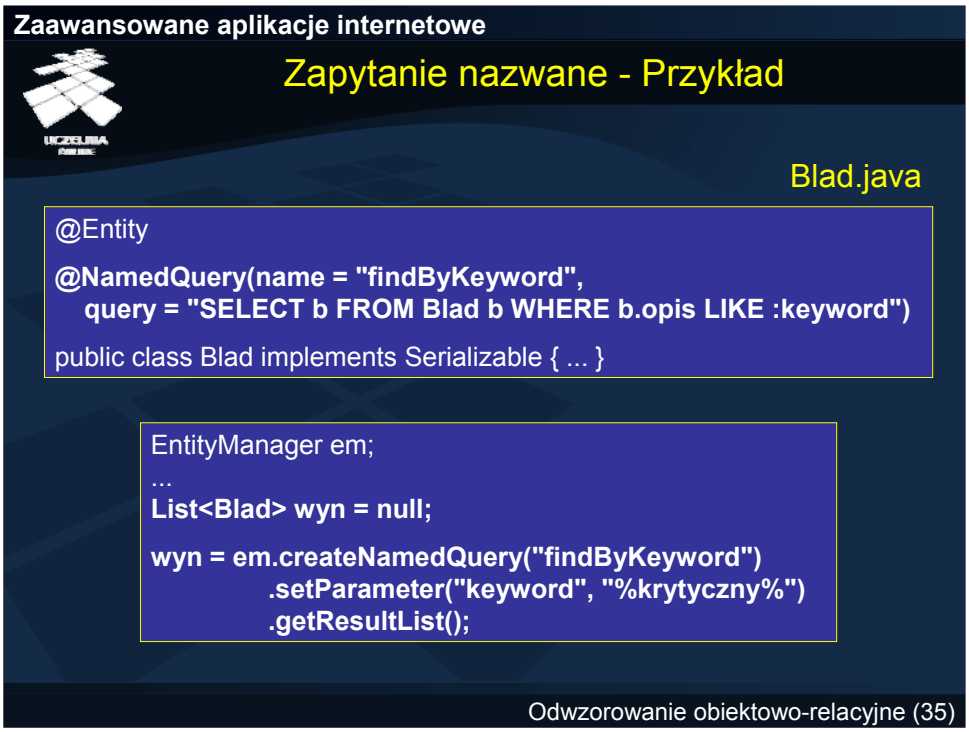
Zapytania do bazy danych w standardzie Java Persistence są reprezentowane przez obiekty Query tworzone metodami obiektu EntityManager. Standard przewiduje trzy rodzaje zapytań: dynamiczne w języku JPQL (Java Persistence Query Language), dynamiczne natywne i nazwane (w JPQL lub natywne). Zapytania nazwane mają taką przewagę nad dynamicznymi, że mogą być prekompilowane i lepiej optymalizowane, a przez to efektywniejsze.

Zapytania mogą zawierać parametry. Zalecane jest wykorzystywanie parametrów nazwanych, ale dostępna jest też notacja pozycyjna.

Wykonanie zapytania sprowadza się do wywołania jednej z metod utworzonego obiektu Query (po wcześniejszym ustawieniu wartości parametrów, jeśli zapytanie jest sparametryzowane). Metoda getResultList() zwraca wyniki zapytania jako kolekcję typu List. Metoda getSingleResult() służy do odczytu pojedynczego wyniku zapytania w postaci obiektu Object. Metoda executeUpdate() służy do wykonania polecenia DELETE lub UPDATE w bazie danych i jest przeznaczona do masowych operacji modyfikowania i usuwania wierszy w tabelach bazy danych.



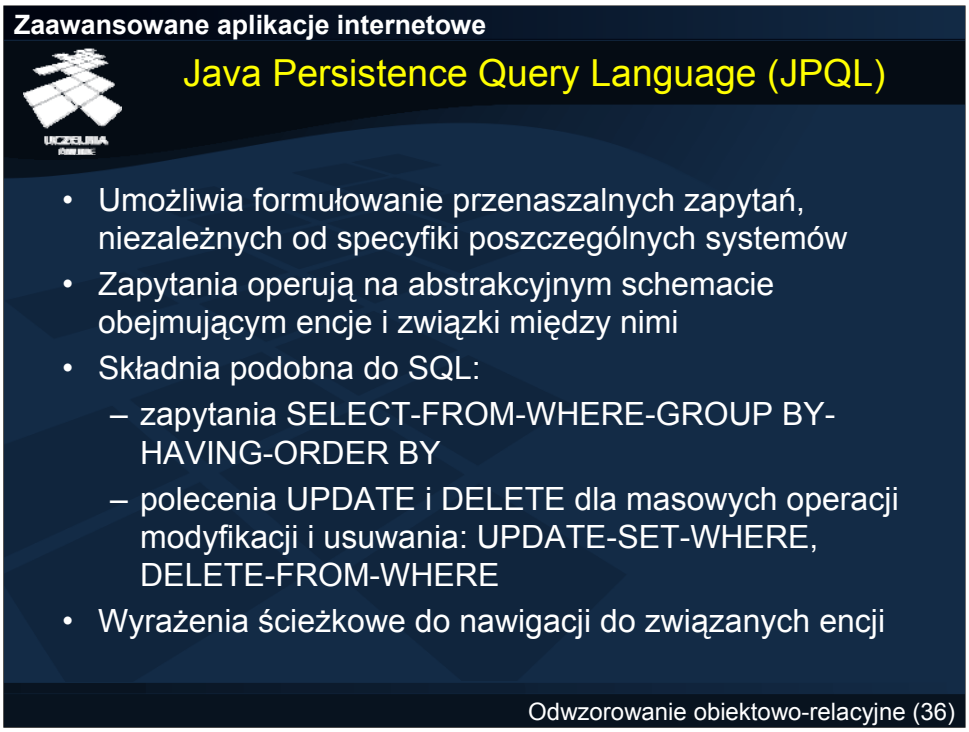
Slajd pokazuje przykład wykonania dynamicznego zapytania sformułowanego w języku JPQL. Zapytanie ma zwrócić wszystkie obiekty Blad, których atrybut „opis” zawiera słowo kluczowe „problem”. Zapytanie podane jako parametr metody createQuery() obiektu EntityManager zostało sformułowane w przenaszalnym języku JPQL, którego składnia została oparta o SQL. Dla prostych zapytań, takich jak w naszym przykładzie, często jedyną różnicą między sformułowaniem zapytania w JPQL i SQL jest to, że w JPQL w klauzuli FROM podaje się nazwy encji, a nie nazwy tabel w bazie danych. Zapytanie może zwrócić więcej niż jedną instancję encji spełniającą kryteria selekcji, więc wynik zapytania jest pobrany metodą getResultList() i przypisany do kolekcji typu List<Blad>.



Slajd pokazuje przykład definicji i wykonania sparametryzowanego nazwanego zapytania sformułowanego w języku JPQL. Zapytanie umożliwia wyszukanie wszystkich obiektów Blad, których atrybut „opis” zawiera słowo kluczowe podane jako nazwany parametr „:keyword”.

Zapytania nazwane są definiowane za pomocą adnotacji @NamedQuery. Gdy klasa encji zawiera definicje kilku zapytań nazwanych, są one zgrupowane w adnotacji @NamedQueries. Fragment kodu u góry slajdu pokazuje definicję w klasie Blad nazwanego zapytania o nazwie „findByKeyword” i treści zawierającej nazwany parametr „:keyword”.

Fragment kodu u dołu slajdu pokazuje sposób utworzenia obiektu nazwanego zapytania, a następnie ustawienia wartości parametru i wykonania zapytania. W przykładzie te trzy operacje zostały zawarte w jednej złożonej instrukcji, ale mogłyby również zostać rozbite na trzy odrębne instrukcje. Zapytanie nazwane tworzone jest metodą createNamedQuery() obiektu EntityManager, której parametrem jest nazwa zapytania nazwanego, podana w adnotacji @NamedQuery w klasie encji. Do ustawienia wartości parametru sparametryzowanego zapytania (tworzonego jako nazwane lub dynamiczne) służy metoda setParameter() obiektu Query.



Java Persistence Query Language (JPQL) umożliwia formułowanie przenaszalnych zapytań, niezależnych od specyfiki poszczególnych systemów zarządzania bazą danych i różnych dialektów języka SQL.

Zapytania w JPQL operują na abstrakcyjnym schemacie obejmującym encje i związki między nimi. Przed wykonaniem treść zapytania jest tłumaczona na odpowiedni dla wykorzystywanego systemu zarządzania bazą danych dialekt SQL i formułowana w kontekście tabel w bazie danych.

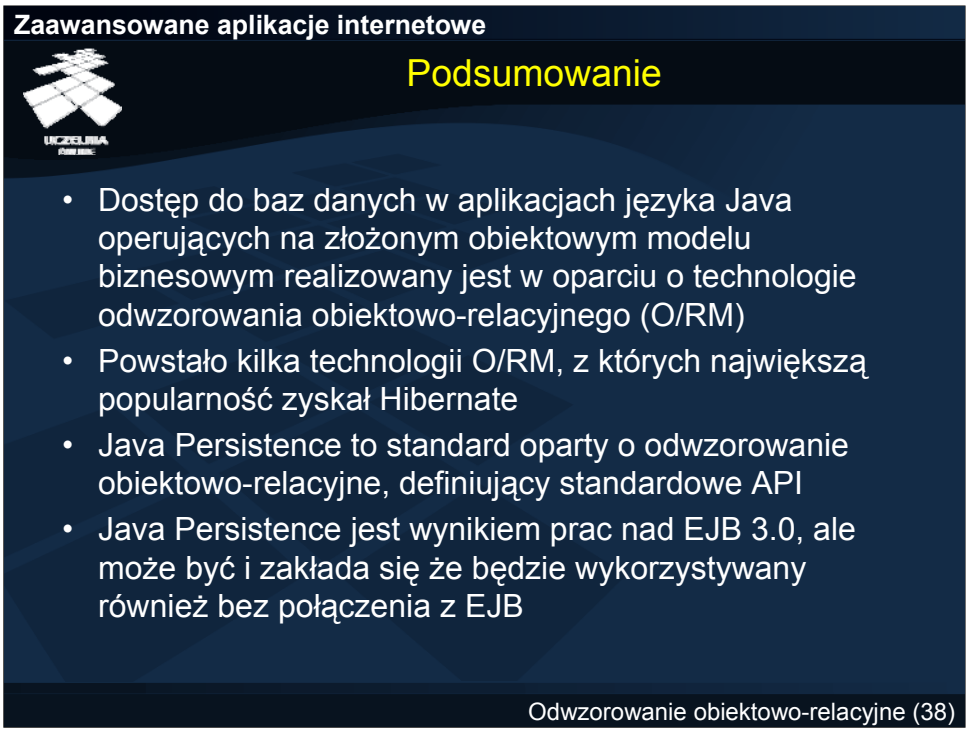
Składnia JPQL jest bardzo podobna do składni SQL. Ogólna składnia zapytań obejmuje znane z SQL klauzule SELECT-FROM-WHERE-GROUP BY-HAVING-ORDER BY. JPQL zawiera również polecenia UPDATE i DELETE dla masowych operacji modyfikacji i usuwania o ogólnej składni UPDATE-SET-WHERE i DELETE-FROM-WHERE.

Podstawową różnicą między JPQL a SQL jest obecność w języku JPQL konstrukcji do odczytu powiązanych kolekcji instancji encji i wyrażeń ścieżkowych do nawigacji po powiązanych instancjach encji.

Slajd pokazuje przykładowe zapytania w JPQL wykorzystujące konstrukcje składniowe nieposiadające swoich odpowiedników w SQL.

U góry pokazany został przykład zapytania realizującego nawigację do kolekcji powiązanych instancji encji. Źródłem danych dla zapytania jest encja Wykonawca i związane z każdym obiektem Wykonawca obiekty Album. Słowo kluczowe IN podkreśla, że albumy to kolekcja powiązanych instancji encji, ale to samo zapytanie można zapisać również z wykorzystaniem operatora połączenia JOIN. Wynikiem zapytania są wykonawcy posiadający jakieś albumy. Operator DISTINCT eliminuje duplikaty jakie mogą się pojawić w wyniku operacji połączenia.

Przykład u dołu ilustruje wykorzystanie wyrażeń ścieżkowych do nawigacji po związkach między encjami. Wyrażenia ścieżkowe mogą pojawić się we wszystkich klauzulach polecenia SELECT, a także w poleceniach UPDATE i DELETE. W przykładowym zapytaniu wyrażenie ścieżkowe zostało użyte w klauzuli WHERE do wybrania albumów wykonawcy o podanej nazwie.



Dostęp do baz danych w aplikacjach języka Java operujących na złożonym obiektowym modelu biznesowym najczęściej realizowany jest w oparciu o technologie odwzorowania obiektowo-relacyjnego (O/RM). Powstało kilka różnych technologii O/RM, z których największą popularność zyskał Hibernate.

Java Persistence to standard oparty o odwzorowanie obiektowo-relacyjne, definiujący standardowy interfejs programistyczny, standardowy mechanizm specyfikowania metadanych opisujących odwzorowanie i język do formułowania przenaszalnych zapytań do baz danych – JPQL. Java Persistence jest wynikiem prac nad EJB 3.0, ale może być i zakłada się, że będzie wykorzystywany również bez połączenia z EJB, zarówno w aplikacjach Java EE jak i Java SE.

Materiały dodatkowe

- Hibernate, http://www.hibernate.org/
- The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/

•Hibernate, http://www.hibernate.org/
•The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/