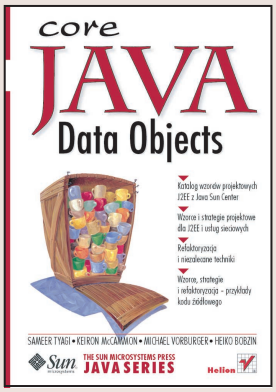
Java Data Objects



Autorzy: Sameer Tyagi, Keiron McCammon, Michael Vorburger, Heiko Bobzin
Tłumaczenie: Jaromir Senczyk
ISBN: 83-7361-392-7
Tytuł oryginału: Core Java Data Objects
Format: B5, stron: 456
Przykłady na ftp: 107 kB

Książka „Java Data Objects”:

- Demonstruje praktyczne techniki stosowane przez zawodowych programistów
- Zawiera poprawny, gruntownie przetestowany kod źródłowy programów oraz przykłady zaczerpnięte z praktyki
- Skoncentrowana jest na nowoczesnych technologiach, które muszą poznać programiści
- Zawiera rady profesjonalistów, które pozwolą czytelnikowi tworzyć najlepsze programy

Java Data Objects (JDO) przyspiesza tworzenie aplikacji w Javie dostarczając obiektowego mechanizmu trwałości i standardowych interfejsów umożliwiających korzystanie z baz danych. Książka ta jest wszechstronnym przewodnikiem po zagadnieniach trwałości JDO, przeznaczony dla zaawansowanego programisty.

Korzystając z realistycznych przykładów kodu autorzy przedstawiają sposoby tworzenia, pobierania, aktualizacji i usuwania obiektów trwałych, cykl życia obiektów i przejścia pomiędzy stanami, klasy i interfejsy JDO, zapytania, architekturę, problemy bezpieczeństwa i wiele innych zagadnień. Prezentują sposoby integracji JDO z EJB™, JTA, JCA i innymi technologiami J2EE™, omawiają też najlepsze sposoby wykorzystania JDO przez samodzielne aplikacje oraz komponenty J2EE™.

Jeśli chcesz poświęcić więcej czasu na rozwiązywanie zagadnień biznesowych, a mniej tracić na zajmowanie się problemem trwałości, to potrzebujesz właśnie JDO i jednej dobrej książki, która pomoże Ci efektywnie użyć JDO: „Java Data Objects”.

- Omówienie specyfikacji JDO i podstawowych zagadnień związanych z trwałością obiektów
- Programowanie z użyciem JDO; najważniejsze klasy i obiekty
- Cykl życia obiektów
- Wyszukiwanie danych w JDO
- Przykładowa architektura i jej konstrukcja z użyciem JDO
- JDO a J2EE: JCA, EJB, transakcje, bezpieczeństwo
- Porównanie JDO z JDBC
- Przyszłość JDO i rozwój tej specyfikacji
- Studium przypadku

Uzupełnieniem są liczne dodatki omawiające między innymi: stany JDO, metadane, język JDOQL w notacji BNF i dostępne implementacje JDO.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

4

Cykl życia obiektów

W rozdziale tym omówimy cykl życia obiektów oraz przedstawimy powody, dla których znajomość zmian stanu obiektów JDO może być istotna dla programisty aplikacji.

Dlaczego znajomość cyklu życia obiektu i zmian stanu jest tak ważna? Wyobraźmy sobie, jak działa maszyna wirtualna JVM, gdy aplikacja wywołuje operator `new`. Wiemy, że przydzielany jest odpowiedni obszar pamięci i zwracana jest jego referencja, ale w rzeczywistości działanie maszyny wirtualnej jest bardziej skomplikowane. Najpierw może się okazać, że maszyna JVM musi załadować kod bajtowy z pewnego zasobu i przetłumaczyć go na instrukcje macierzystego procesora. Wykonywana jest statyczna inicjacja, ładowane są klasy bazowe i wywoływane konstruktory. Jeszcze bardziej skomplikowany może być przebieg usuwania obiektu, w który zaangażowany będzie mechanizm odzyskiwania nieużytków, mechanizm słabych referencji, kolejki i zwrotne wywołania metod `finalize`. Dlatego też maszyna wirtualna JVM musi dysponować dobrze zdefiniowanym modelem cyklu życia obiektów. Podobnym modelem cyklu dysponuje również implementacja JDO dla obiektów trwałych — rozpoczynają one życie jako zwykłe, ulotne obiekty języka Java, następnie uzyskują trwałość, mogą zostać usunięte bądź z powrotem przejść do stanu ulotnego i tak dalej. Znajomość cyklu życia obiektów trwałych w ogóle nie jest konieczna, zwłaszcza szczegółowa. Jeśli jednak aplikacja używa wywołań zwrotnych, to zalecane jest myślenie w kategoriach efektów ubocznych związanych ze stanami obiektów trwałych i ich zmianami.

Najpierw omówione zostaną obowiązkowe i opcjonalne stany obiektów trwałych, metody umożliwiające określenie stanu obiektu oraz metody, których skutkiem jest zmiana stanu. Zmiany stanu obiektów trwałych zilustrowane zostaną fragmentami kodu źródłowego w języku Java wyświetlającymi dodatkowo informacje o stanie tych obiektów. Cześć poświęcona wywołaniom zwrotnym wyjaśni sposób i warunki ich użycia. Na końcu przedstawione zostaną te stany i operacje, które są w JDO opcjonalne.

4.1. Cykl życia obiektu trwałego

JDO definiuje siedem stanów obowiązkowych i trzy stany opcjonalne. Stany `transient`, `persistent-new`, `persistent-new-deleted`, `hollow`, `persistent-clean`, `persistent-dirty` i `persistent-`

deleted są obowiązkowe. Stany transient-dirty, transient-clean i persistent-nontransactional są opcjonalne i zostaną omówione w dalszej części rozdziału.

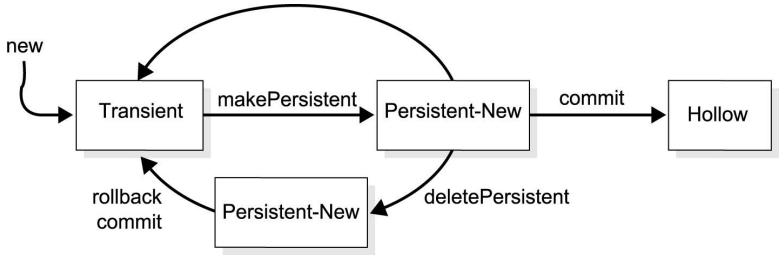
Niektóre ze zmian stanu obiektu są bezpośrednio wywoływane przez aplikację — na przykład na skutek utrwalenia obiektu. Również rozpoczęcie lub zakończenie transakcji może wywołać zmianę stanu obiektu, czyli zachodzi ona na skutek wywołania przez aplikację jednej z metod begin(), commit() lub rollback() klasy Transaction. Zauważmy, że w środowiskach zarządzanych zmiany stanu mogą być również wywoływane przez zewnętrzny kontroler transakcji. Instancja języka Java rozpoczyna cykl życia JDO na skutek wywołania metody makePersistent() lub załadowania istniejącego obiektu trwałego z bazy danych.

Rozdział rozpoczniemy od omówienia dwóch typowych sekwencji utrwalania obiektów i odtwarzania ich na podstawie informacji zapisanych w bazie danych. Zilustrujemy je krótkim przykładem kodu oraz pokażemy sposób użycia klasy JDOHelper w celu uzyskania informacji o zmianach stanów zachodzących podczas wykonywania wspomnianych sekwencji.

4.1.1. Utrwalanie obiektu

Operacje prowadzące do utrwalenia obiektu mogą przebiegać w sposób przedstawiony na diagramie stanów (rysunek 4.1).

Rysunek 4.1. *Utrwalanie instancji ulotnych*



4.1.1.1. Stan ulotny (transient)

Instancje klasy zdolnej do trwałości zachowują się po ich utworzeniu za pomocą operatora new w taki sam sposób jak instancje każdej innej klasy języka Java. Modyfikacje pól tych instancji nie są śledzone, co w praktyce oznacza taką samą efektywność dostępu jak w przypadku instancji klas, które nie są zdolne do trwałości. Na instancje ulotne nie mają wpływu wywołania metod związanych z transakcjami czyli begin(), commit() i rollback() z wyjątkiem sytuacji, w których instancje te są osiągalne za pośrednictwem obiektów trwałych.

4.1.1.2. Stan persistent-new

Wywołanie metody PersistenceManager.makePersistent() przypisuje instancji nowy identyfikator, a jej stan zmienia się z transient na persistent-new. Obiekt w stanie persistent-new zachowuje się tak samo jak obiekt w stanie transient, ale zapisywany jest w bazie danych podczas zatwierdzania transakcji lub gdy stan obiektu musi zostać zsynchronizowany ze stanem w wewnętrznych buforach JDO. Obiekt znajdujący się w stanie persistent-new

jest już kontrolowany przez implementację JDO. Obiekty mogą przejść do stanu persistent-new również wtedy, gdy osiągalne są za pośrednictwem innych obiektów trwałych. Proces ten przebiega tak długo, aż wszystkie instancje osiągalne z danego obiektu znajdą się w stanie transient lub persistent-new. Pomiędzy wywołaniem metody makePersistent() a zatwierdzeniem transakcji referencje mogą ulec zmianie, a osiągalne instancje stać się niedostępne. Takie tymczasowo trwałe obiekty znajdują się również w stanie persistent-new. Jednak ich stan może ulec zmianie podczas zatwierdzania transakcji, gdy okaże się, że tymczasowo trwałe obiekty nie są już osiągalne.

4.1.1.3. Stan pusty (hollow)

Po pomyślnym zakończeniu transakcji stan obiektu zmienia się z persistent-new na hollow. Obiekt w stanie pustym jest referencją, która może reprezentować pewne dane znajdujące się w bazie danych, ale które nie zostały jeszcze załadowane do pamięci lub są nieaktualne. Zwykle inni klienci bazy danych lub inne aplikacje JDO posiadają swobodny dostęp oraz możliwość modyfikacji danych w bazie, których referencją jest pusty obiekt. Gdy aplikacja będzie próbowała użyć obiektu w stanie pustym, to obowiązkiem implementacji jest ponowne załadowanie danych do pamięci. Scenariusz ten zostanie szczegółowo wyjaśniony nieco dalej. Po zatwierdzeniu transakcji zawartość obiektów znajdujących się w stanie pustym zostaje usunięta w celu zwolnienia pamięci. Jeśli wśród tych danych znajdowały się referencje obiektów, to mechanizmowi odzyskiwania nieużytków nie wolno zwolnić grafu obiektów.

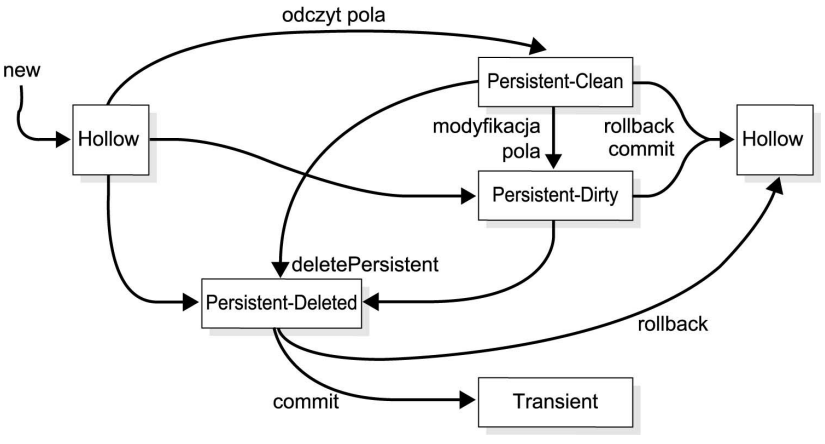
Stan pusty nie może zostać stwierdzony przez aplikację. Nie jest dostępna odpowiednia w tym celu metoda klasy JDOHelper, a próba dostępu do pól takiego obiektu może spowodować zmianę jego stanu i nie jest wtedy możliwe określenie poprzedniego stanu obiektu.

4.1.1.4. Stan persistent-new-deleted

Gdy obiekt znajdujący się w stanie persistent-new zostaje usunięty przez wywołanie metody PersistenceManager.deletePersistent(), to jego stan zmienia się na persistent-new-deleted. Przywrócenie takiego obiektu nie jest możliwe i stan persistent-new-deleted jest ostateczny do momentu zakończenia transakcji. Próba dostępu do pól obiektu znajdującego się w stanie persistent-new-deleted spowoduje zgłoszenie wyjątku JDOUserException. Nie dotyczy to jedynie pól wchodzących w skład klucza głównego. Po zakończeniu transakcji (poprzez wywołanie metody commit() lub rollback()) usunięta instancja przechodzi do stanu transient.

4.1.2. Odtwarzanie obiektów z bazy danych

Przejdziemy teraz do omówienia zmian stanów instancji, które zostają początkowo załadowane z bazy danych. Diagram zmian stanów dla takich instancji przedstawiony został na rysunku 4.2



Rysunek 4.2. *Odtwarzanie instancji z bazy danych*

4.1.2.1. Stan pusty (hollow)

Jak już wspomnieliśmy wcześniej, obiekty utworzone w celu reprezentacji referencji danych zapisanych w bazie danych, które nie zawierają jeszcze danych, znajdują się również w stanie pustym. Próba dostępu do ich pól powoduje załadowanie danych z bazy. Obiekty znajdujące się w stanie pustym tworzone są za pomocą wywołań metod takich jak `Extent.iterator().next()`, `PersistenceManager.getObjectById()` lub wykonania zapytania. Zwykle trwałe pola obiektów z stanie pustym inicjowane są za pomocą wartości domyślnych (0 lub `null`), ale ponieważ obiekty takie tworzone są za pomocą konstruktorów nieposiadających argumentów, to konstruktory te mogą nadawać tymże polom również inne wartości początkowe. Na tym może właśnie polegać jedyna różnica pomiędzy obiektami w stanie pustym, które brały wcześniej udział w transakcji (omówionymi w poprzednim punkcie), a obiektami w stanie pustym utworzonymi przez implementację JDO.

4.1.2.2. Stan persistent-clean

Po nadaniu polom obiektu wartości pobranych z bazy danych obiekt taki przechodzi ze stanu `hollow` do stanu `persistent-clean`, ale tylko wtedy, gdy jego dane nie zostały zmodyfikowane przez bieżącą transakcję. Stan te nosi taką nazwę (ang. *clean* — czysty), ponieważ dane obiektu w pamięci są spójne z odpowiadającymi im danymi w bazie. Chociaż pobraniem danych obiektu zajmuje się implementacja JDO, to referencje do innych obiektów rozwiązywane są przez instancję klasy `PersistenceManager`. Referencje te mogą wskazywać obiekty, które znajdują się już w pamięci. W przeciwnym razie `PersistenceManager` tworzy nowe obiekty w stanie pustym. Zasoby obiektów znajdujących się w stanie `persistent-clean` mogą zostać zwolnione przez mechanizm odzyskiwania nieużytków, jeśli w kodzie aplikacji nie istnieją już referencje do tych obiektów. Jeśli obiekty zostały rzeczywiście zwolnione przez ten mechanizm, to obowiązkiem klasy `PersistenceManager` jest buforowanie ich referencji i utworzenie nowych obiektów w stanie pustym, gdy okaże się, że są one potrzebne.

4.1.2.3. Stan persistent-dirty

Obiekt przechodzi ze stanu `hollow` lub `persistent-clean` do stanu `persistent-dirty`, gdy zmodyfikowane zostaje co najmniej jedno jego pole. Stan ten nosi taką nazwę (ang. *dirty* — brudny), ponieważ dane obiektu w pamięci przestają być spójne z danymi znajdującymi się w bazie. Nawet jeśli uzyskana w wyniku modyfikacji wartość pola jest taka sama jak poprzednia, to obiekt i tak przechodzi do stanu `persistent-dirty`. Co więcej, instancja nie powraca do stanu `hollow` lub `persistent-clean`, gdy przywrócone zostaną poprzednie wartości pól. Gdy obiektowi znajdującemu się w stanie `persistent-clean` lub `persistent-dirty` zostanie przypisana ulotna instancja, to obiekt taki nie przechodzi do stanu `persistent-new`.

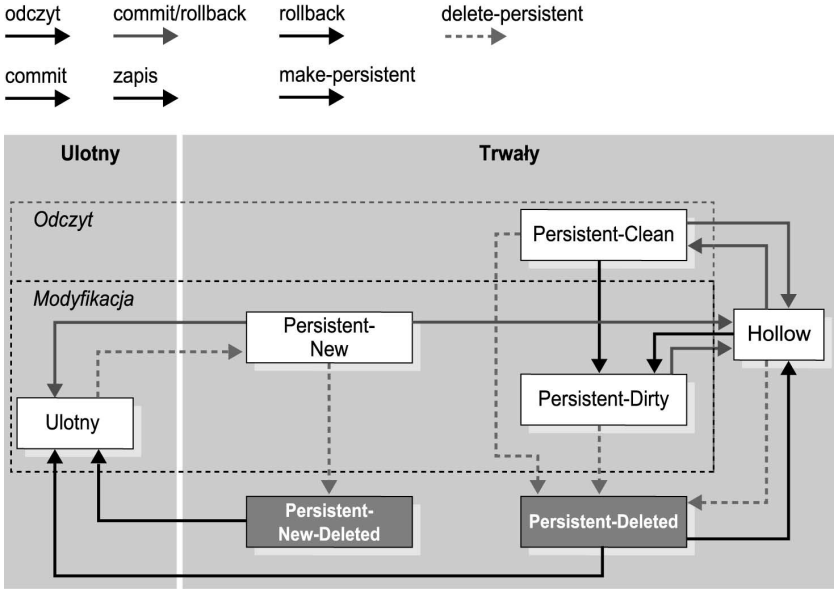
4.1.2.4. Stan persistent-deleted

Na skutek wywołania metody `PersistentManager.deletePersistent()` stan obiektu trwałego zmienia się na `persistent-deleted`. Opuszczenie tego stanu czyli przywrócenie usuniętej instancji trwałej możliwe jest jedynie poprzez przerwanie transakcji. Pomyślne zatwierdzenie transakcji powoduje przejście obiektu ze stanu `persistent-deleted` do stanu `transient`. Obiekt nie może zostać ponownie utrwalony z tą samą tożsamością tak długo jak znajduje się w stanie `persistent-deleted`.

4.1.3. Uproszczony cykl życia obiektu

Diagram zamieszczony na rysunku 4.3 przedstawia uproszczony cykl życia obiektu. Obszar obwiedziony prostokątem *Odczyt* zawiera stany, w których odczyt (dostęp) obiektu nie powoduje zmiany ich stanu. Prostokąt *Zapis* zawiera stany, w których modyfikacja obiektu nie powoduje zmiany ich stanu.

Rysunek 4.3. *Cykl życia obiektu — stany obowiązkowe*



4.2. Informacja o stanie obiektu

Wymienionych wcześniej siedem stanów: `transient`, `persistent-new`, `persistent-new-deleted`, `hollow`, `persistent-clean`, `persistent-dirty` i `persistent-deleted` jest obowiązkowych dla wszystkich implementacji JDO. Jednak implementacje JDO nie udostępniają żadnego interfejsu programowego, który umożliwiałby uzyskanie informacji o stanach, w których znajdują się obiekty JDO. Dokładny stan obiektu znany jest jedynie implementacji JDO, a niektóre stany mogą nawet nie być implementowane przez pewnych producentów. Stany obiektów JDO są zdefiniowane bowiem w sposób behawioralny. Mimo to istnieją pewne metody, które pozwalają sprawdzić atrybuty trwałości dla danego obiektu i na ich podstawie wnioskować o stanie obiektu.

Metody te należą do klasy `JDOHelper`:

```
boolean isDeleted(Object o)
boolean isDirty(Object o)
boolean isNew(Object o)
boolean isPersistent(Object o)
boolean isTransactional(Object o)
```

Nie istnieje natomiast metoda, która pozwalałaby ustalić, czy obiekt znajduje się w stanie pustym. Sposób ładowania zawartości takiego obiektu zależy bowiem od konkretnej implementacji. W dalszej części rozdziału przedstawimy jednak inny sposób, który pozwala uzyskać informację o tym, czy obiekt znajduje się w stanie pustym. Przedstawiona poniżej klasa wyświetla znaczniki instancji, posługując się klasą `JDOHelper`:

```
public class Example01
{
  public static void printInfo(String msg, Object obj)
  {
    StringBuffer buf = new StringBuffer(msg);
    if (JDOHelper.isPersistent(obj))
      buf.append(" persistent");
   if (JDOHelper.isNew(obj))
      buf.append(" new");
   if (JDOHelper.isDirty(obj))
      buf.append(" dirty");
   if (JDOHelper.isDeleted(obj))
      buf.append(" deleted");
   if (JDOHelper.isTransactional(obj))
      buf.append(" transactional");
   System.out.println(buf.toString());
  }
}
```

Poniższy fragment kodu ilustruje omówione dotąd stany i metody:

```
public static void main(String args[])
{
  PersistenceManager pm = pmf.getPersistenceManager();
  pm.currentTransaction().begin();

  Book book = new Book();
```

```
    printInfo("nowa instancja klasy Book ", book);

    pm.makePersistent(book);
    printInfo("trwała instancja klasy Book ", book);

    pm.deletePersistent(book);
    printInfo("usunięta instancja klasy Book ", book);

    pm.currentTransaction().rollback();
    printInfo("odwołanie transakcji ", book);

    pm.currentTransaction().begin();
    pm.makePersistent(book);
    pm.currentTransaction().commit()
    printInfo("zatwierdzenie transakcji ", book);

    pm.currentTransaction().begin();
    String title = book.title;
    printInfo("odczyt pola ", book);


    book.title = "nowa " + book.title;
    printInfo("modyfikacja pola ", book);

    pm.deletePersistent(book);
    printInfo("usunięta instancja klasy Book", book);


    pm.currentTransaction().commit();
    printInfo("zatwierdzenie transakcji ", book);
  }
```

Wykonanie powyższego kodu spowoduje wyświetlenie następującego tekstu:

```
nowa instancja klasy Book: dirty
trwała instancja klasy Book: persistent new dirty
usunięta instancja klasy Book: persistent new dirty deleted
odwołanie transakcji: dirty
zatwierdzenie transakcji: persistent
odczyt pola: persistent
modyfikacja pola: persistent dirty
usunięta instancja klasy Book: persistent dirty deleted
zatwierdzenie transakcji: dirty
```

4.3. Operacje powodujące zmianę stanu

W poprzednim podrozdziale przestawiliśmy podstawowe stany obiektów oraz operacje takie jak odczyt pola lub usunięcie obiektu trwałego z bazy, które w oczywisty sposób zmieniają stan obiektów. W tym podrozdziale przyjrzymy się bliżej metodom zmieniającym stan obiektów i wyjaśnimy ich działanie bardziej szczegółowo. Niektóre z tych metod zmieniają stan obiektów na skutek bezpośredniego ich wywołania, inne wpływają pośrednio na stan wielu obiektów.

4.3.1. PersistenceManager.makePersistent

Metoda `makePersistent` klasy `PersistenceManager` powoduje przejście ulotnych obiektów do stanu `persistent-new`. Podczas wykonania tej metody przeglądany jest graf obiektu w celu znalezienia innych obiektów znajdujących się jeszcze w stanie `transient`. Również te obiekty mogą niejawnie przejść do stanu `persistent-new`.

4.3.2. PersistenceManager.deletePersistent

Wywołanie metody `deletePersistent` powoduje przejście obiektu ze stanu `persistent-clean`, `persistent-dirty` lub `hollow` do stanu `persistent-deleted`, natomiast obiekty znajdujące się dotychczas w stanie `persistent-new` przechodzą do stanu `persistent-new-deleted`. Operacja ta nie ma wpływu na inne osiągalne obiekty.

4.3.3. PersistenceManager.makeTransient

Obiekt może przejść do stanu `transient` pod warunkiem, że jego obecnym stanem jest `persistent-clean` lub `hollow`. Na skutek wywołania metody `makeTransient` obiekt taki zostaje wyjęty spod kontroli instancji klasy `PersistenceManager` i traci swoją tożsamość. Metoda ta nie wywiera wpływu na inne obiekty osiągalne za pośrednictwem obiektu, który był jej parametrem wywołania.

4.3.4. Transaction.commit

Instancje należące do bieżącej transakcji są przetwarzane przez JDO podczas jej zatwierdzania, instancje znajdujące się w stanie `hollow` nie zmieniają swojego stanu, natomiast instancje znajdujące się w stanie `new`, `clean` lub `dirty` przechodzą do stanu `hollow`. Wszystkie inne instancje stają się ulotne. Po zatwierdzeniu instancji obiekty znajdujące się w stanie pustym oraz zwykłe, ulotne obiekty języka Java pozostają pod kontrolą instancji klasy `PersistenceManager`.

4.3.5. Transaction.rollback

Na skutek odwołania transakcji obiekty znajdujące się w stanie `clean`, `dirty` lub `deleted` przechodzą do stanu `hollow`. Pozostałe obiekty stają się ulotne. Obiekty, które znajdowały się w stanie `hollow` lub `transient` powracają do swojego poprzedniego stanu.

4.3.6. PersistenceManager.refresh

Wywołanie metody `refresh` powoduje ponowne załadowanie danych obiektu z bazy na skutek czego wszelkie modyfikacje obiektu zostają utracone. Metoda ta przeprowadza obiekty ze stanu `persistent-dirty` do stanu `persistent-clean`.

4.3.7. PersistenceManager.evict

Metoda ta umożliwia zaoszczędzenie pamięci przez implementację buforowania stosowaną w klasie `PersistenceManager`. Na skutek jej użycia obiekty przechodzą ze stanu `persistent-clean` do stanu `hollow`. Wartości pól obiektów są przy tym usuwane, aby umożliwić mechanizmowi odzysku nieużytków zwolnienie podobiektów.

4.3.8. Odczyt pól wewnątrz transakcji

Po wczytaniu danych z bazy obiekty przechodzą ze stanu `hollow` do stanu `persistent-clean`. Gdy stosowany jest pesymistyczny algorytm zarządzania współbieżnością, to obiekt może zostać zablokowany dla zapobieżenia równoczesnej modyfikacji.

4.3.9. Zapis pól wewnątrz transakcji

Gdy podczas wykonywania transakcji zmodyfikowane zostaje pole obiektu znajdującego się pod kontrolą instancji klasy `PersistenceManager`, to implementacja JDO musi zapamiętać, że należy później zaktualizować dane tego obiektu w bazie. Obiekt przechodzi wtedy ze stanu `persistent-clean` lub `hollow` do stanu `persistent-dirty`. Jeśli stosowany jest pesymistyczny algorytm zarządzania współbieżnością, to implementacja JDO może zablokować dostęp do obiektu innym klientom tej samej bazy danych. Modyfikacja wartości pól obiektów znajdujących się w stanie `transient`, `persistent-dirty` lub `persistent-new` pozostaje bez wpływu na stan takich obiektów. Gdy polu obiektu przypisywany jest inny obiekt, to zmodyfikowany obiekt przechodzi do stanu `persistent-dirty`, a obiekt przypisany nie zmienia swojego stanu.

4.3.10. PersistenceManager.retrieve

Wywołanie metody `PersistenceManager.retrieve()` ma ten sam skutek, co odczyt pola obiektu podczas bieżącej transakcji.

4.4. Wywołania zwrotne

Implementując metody wywoływane zwrotnie aplikacja może przejąć obsługę części omówionych wydarzeń występujących w cyklu życia obiektu. Istnieją w zasadzie trzy sytuacje, w których programista aplikacji może zaimplementować w ten sposób specjalne zachowanie obiektów trwałych:

- odczyt lub zapis wartości pól, które nie są trwałe lub nie powinny być przetwarzane przez domyślny algorytm JDO,
- usuwanie obiektów składowych,
- usuwanie zbędnych referencji.

Metody wywoływane zwrotnie umieszczone zostały w osobnym interfejsie, którego obsługa przypomina sposób obsługi interfejsu etykietowego `java.io.Serializable`. Jeśli klasa zdolna do trwałości implementuje interfejs `InstanceCallbacks`, to jest to sygnałem dla implementacji JDO, że należy użyć wywołań zwrotnych. W przeciwnym razie nie są one wykorzystywane.

4.4.1. Zastosowania metody jdoPostLoad

Metoda `jdoPostLoad` wywoływana jest za każdym razem, gdy obiekt trwały zostaje załadowany z bazy danych czyli, gdy jego stan zmienia się z `hollow` na `persistent-clean`. Dokładniej rzecz ujmując, metoda ta jest wywoływana w momencie, gdy dane obiektu stają się spójne z danymi zapisanymi w bazie danych.

> **Interfejs InstanceCallbacks**
>
> Jeśli klasa implementuje interfejs `InstanceCallbacks`, to metody `jdoPostLoad`, `jdoPreStore`, `jdoPreClear` i `jdoPreDelete` są automatycznie wywoływane przez implementację JDO, gdy pola instancji są odczytywane, modyfikowane, zerowane lub obiekty są usuwane. W przeciwieństwie do interfejsu `Serializable` klasa odpowiedzialna jest za wywołanie metod interfejsu `InstanceCallbacks` swojej klasy bazowej. Metody interfejsu `InstanceCallbacks` zadeklarowane są jako metody o dostępie publicznym i dlatego należy zwrócić uwagę, by nie wywoływać ich w innych sytuacjach.

Implementując metodę `jdoPostLoad`, możemy zaoszczędzić na kodzie sprawdzającym początkowe przypisanie wartości ulotnych, ponieważ JDO definiuje, że metoda ta wywoływana jest dokładnie raz. Klasa może więc zaimplementować metodę `jdoPostLoad`, aby przypisać wartości początkowe polom ulotnym, zwłaszcza gdy zależą one od wartości pól trwałych. Możliwość taka jest szczególnie przydatna, gdy ustalenie tych wartości wymaga skomplikowanych obliczeń, które nie powinny być wykonywane wielokrotnie.

Inne zastosowanie może polegać na rejestrowaniu przez obiekty aplikacji załadowania obiektów trwałych z bazy danych. Na przykład w celu odroczenia inicjacji kolekcji do momentu rzeczywistego jej użycia, co pozwala zaoszczędzić pamięć i zwiększyć efektywność działania programu:

```
public class Book
  implements javax.jdo.InstanceCallbacks
{
  transient Map myMap; // na razie nie jest zainicjowana

  public Book(String title) { // używany przez aplikację
    . . .
  }

  private Book() { // używany przez JDO
  }

  public void jdoPostLoad() {
    myMap = new HashMap(); // inicjowana podczas ładowania
  }
```

```
  // pozostałe metody wywoływane zwrotnie
  public void jdoPreStore() {
  }
  public void jdoPreDelete() {
  }
  public void jdoPreClear() {
  }
}
```

4.4.2. Zastosowania metody jdoPreStore

Metoda ta wywoływana jest przed zapisaniem pól do bazy danych. Jedno z zastosowań tej metody polega na aktualizacji wartości pól trwałych na podstawie pól, które nie są trwałe przed zakończeniem transakcji. Metoda może być również wykorzystana w celu sprawdzenia, czy wartości pól trwałych są spójne lub spełniają ograniczenia. W przypadku, gdy wartość pola narusza zdefiniowane ograniczenia, aplikacja może uniemożliwić ich zapisanie w bazie danych, zgłaszając wyjątek. Jednak obiekty biznesowe powinny udostępniać metody walidacji umożliwiające sprawdzenie zgodności z nałożonymi ograniczeniami. Metody te powinny być wywoływane przez aplikację przed zapisaniem danych w bazie. Powiązanie tego rodzaju walidacji ze szkieletem trwałości nie jest dobrą praktyką. Metody `jdoPreStore` należy raczej używać w celu wykrycia błędnych wartości pojawiających się w programie.

Poniżej przedstawiamy sposób aktualizacji pola będącego tablicą bajtów za pomocą strumienia serializowanych obiektów, które nie są trwałe:

```
public class Book
   implements InstanceCallbacks
{
  transient Color myColor; // pole nie jest trwałe
  private byte[] array; // obsługiwana przez większość implementacji JDO

  public void jdoPostLoad() {
    ObjectInputStream in = new ObjectInputStream(
      new ByteArrayInputStream(array));
    myColor = (Color) in.readObject();
    in.close();
  }

  public void jdoPreStore() {
    ByteArrayOutputStream bo = new ByteArrayOutputStream();
    ObjectOutputStream out = new ObjectOutputStream(bo);
    out.writeObject(myColor);
    out.close();
    array = bo.getBytes();
  }

  // pozostałe metody wywoływane zwrotnie
  public void jdoPreDelete() {
  }
  public void jdoPreClear() {
  }
}
```

4.4.3. Zastosowania metody jdoPreDelete

Metoda `jdoPreDelete` wywoływana jest przed usunięciem obiektu z bazy danych — na przykład, gdy aplikacja wywołała metodę `PersistenceManager.deletePersistent`. Metoda `jdoPreDelete` nie jest wywoływana, gdy obiekt usuwany jest przez innego klienta tej samej bazy danych lub przez inne, wykonywane równolegle transakcje. Metoda `jdoPreDelete` powinna być używana jedynie dla obiektów znajdujących się w pamięci. Typowym jej zastosowaniem jest usuwanie podobiektów zależnych od danego obiektu:

```
public class BookShelf
  implements InstanceCallbacks
{
  // instancja klasy BookShelf może zawierać:
  Collection shelves;
  Collection books;
  public void jdoPreDelete() {
     // pobiera właściwą instancję PersistenceManager
    PersistenceManager pm =
      JDOHelper.getPersistenceManager(this);
      // usuwa shelves, ale nie books!
      pm.deleteAll(shelves);
  }
   // pozostałe metody wywoływane zwrotnie
  public void jdoPostLoad() {
  }
  public void jdoPreStore() {
  }
  public void jdoPreClear() {
  }
}
```

Społeczność związana z JDO toczyła długie dyskusje dotyczące tego, czy obsługa obiektów zależnych powinna odbywać się w kodzie źródłowym w języku Java czy raczej być zadeklarowana w pliku metadanych JDO. Kolejny przykład zastosowania metody `jdoPreDelete` pokazuje sposób wyrejestrowania obiektu z innego obiektu, który posiada jego referencję. Implementacja tej operacji może być całkiem skomplikowana, gdy model obiektowy danej aplikacji rozrośnie się. A oto prosty przykład:

```
public class Aisle
  implements InstanceCallbacks
{
  // instancja klasy Aisle zawiera kolekcję instancji klasy BookShelf
  Collection shelves;
}

public class BookShelf
  implements InstanceCallbacks
{
  final Aisle aisle; // położenie półki

  // mogą być tworzone razem:
  public BookShelf(Aisle ai) {
    this.aisle = ai;
    aisle.shelves.add(this);
  }
```

```
  private BookShelf() { /* wymagany przez JDO */ }

  // wyrejestrowuje usuwane obiekty
  // z obiektu, który je zawiera
  public void jdoPreDelete() {
    aisle.shelves.remove(this);
  }

  // pozostałe metody wywoływane zwrotnie
  public void jdoPostLoad() {
  }
  public void jdoPreStore() {
  }
  public void jdoPreClear() {
  }
}
```

Używając JDO mamy zawsze możliwość wyboru modelowania związku pomiędzy obiektami za pomocą referencji wprzód (instancja `Aisle` posiada instancje `BookShelf`) lub referencji wstecz (instancje `BookShelf` należą do instancji `Aisle`). Nawigacja od obiektu nadrzędnego do podrzędnego wymaga zapytania (odnalezienia instancji `BookShelf` posiadającej referencję wskazanej instancji `Aisle`). Wybór sposobu modelowania związków pomiędzy obiektami należy do aplikacji. Gdy do wyrejestrowania obiektów używane są zwrotne wywołania metod, to konieczne jest zastosowanie referencji wstecz (`BookShelf.aisle`).

4.4.4. Zastosowania metody jdoPreClear

Metoda `jdoPreClear` zostaje wywołana, gdy obiekt przechodzi do stanu `hollow` lub `transient` ze stanu `persistent-deleted`, `persistent-new-deleted`, `persistent-clean` lub `persistent-dirty` na skutek zakończenia transakcji. Wywołanie tej metody może być użyte w celu wyzerowania nietrwałych referencji innych obiektów, aby umożliwić zwolnienie zajmowanych przez nie zasobów mechanizmowi odzyskiwania nieużytków. Inne potencjalne zastosowania to wyrejestrowanie jednych obiektów z innych obiektów bądź zerowanie pól ulotnych lub nadawanie im niedozwolonych wartości dla zapobieżenia przypadkowemu dostępowi do wyzerowanych instancji.

4.5. Stany opcjonalne

JDO definiuje stany opcjonalne, które mogą zostać zaimplementowane przez producenta implementacji JDO. Zadaniem tych stanów jest umożliwienie posługiwania się obiektami trwałymi, które mogą nie zawierać spójnych danych na zewnątrz aktywnej transakcji. Stany te mogą być również wykorzystane dla obiektów ulotnych, które powinny zachowywać się jak obiekty trwałe w obrębie aktywnej transakcji. Zanim jednak aplikacja rozpocznie korzystanie z tych opcjonalnych możliwości, powinna najpierw sprawdzić ich dostępność za pomocą metody `PersistenceManagerFactory.supportedOptions()`.

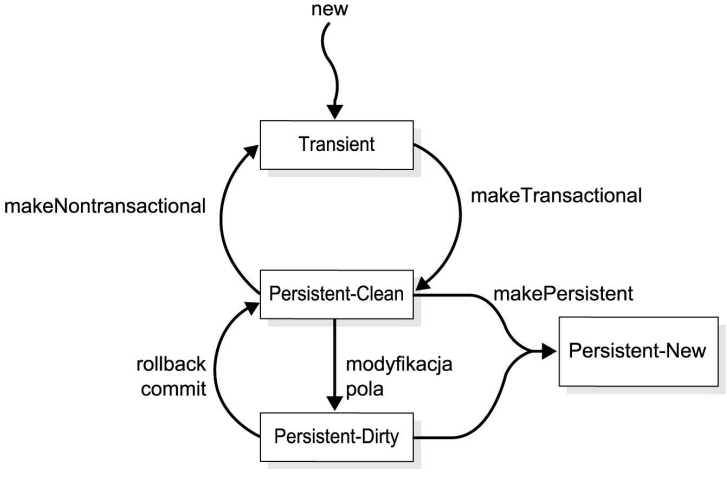
4.5.1. Ulotne instancje transakcyjne

Ulotne instancje klas zdolnych do trwałości mogą stać się transakcyjne, ale nie trwałe na skutek wywołania metody `PersistenceManager.makeTransactional`. Obiekty takie nazywamy ulotnymi instancjami transakcyjnymi. Ich dane zapisywane są w pewien sposób podczas wywołania metody `makeTransactional` i odtwarzane w przypadku odwołania transakcji, jednak nie są utrwalane w bazie danych. Możliwość taka jest szczególnie przydatna w sytuacji, gdy skomplikowana operacja modyfikuje wiele obiektów i może zostać odwołana. Ulotne obiekty znajdujące się w pamięci zostają również powiązane w sensie transakcji z obiektami trwałymi będącymi pod kontrolą tej samej instancji klasy `PersistenceManager`.

Ulotne instancje transakcyjne są dostępne w JDO jedynie opcjonalnie, jednak już podczas pisania tej książki oferowała je większość producentów implementacji JDO. Po wywołaniu metody `makeTransactional` ulotna instancja przechodzi do stanu `transient-clean`. Mimo że jej dane są nowe lub zmodyfikowane, to wywołanie to stanowi jedynie informację dla instancji `PersistenceManager`, że powinna śledzić modyfikacje tego obiektu. Gdy ulotna instancja zostaje zmodyfikowana, przechodzi w stan `transient-dirty` i jej pola trwałe są w celu późniejszego odtworzenia kopiowane. Istnieją dwa przypadki, w których metoda `makeTransactional` może zostać wywołana: przed i po rozpoczęciu aktywnej transakcji. W pierwszym przypadku zapisane zostaną dane obiektu w momencie rozpoczęcia transakcji. W drugim dane obiektu, które posiadał w momencie wywołania metody `makeTransactional`.

Instancje przechodzą ze stanu `transient-dirty` do stanu `transient-clean` w momencie zatwierdzenia lub odwołania transakcji. Jeśli transakcja została odwołana, przywracane są wcześniej zachowane wartości pól. Instancje znajdujące się w stanie `transient-clean` mogą również przejść do stanu `non-transactional` na skutek wywołania metody `makeNonTransactional`. Zarówno instancje znajdujące się w stanie `transient-clean`, jak i `transient-new` mogą przejść do stanu `persistent-new` na skutek wywołania metody `makePersistent`.

Diagram przedstawiony na rysunku 4.4 prezentuje możliwe przejścia pomiędzy różnymi stanami ulotnych instancji transakcyjnych.



Rysunek 4.4. *Cykl życia ulotnych instancji transakcyjnych*

4.5.2. Zastosowania ulotnych instancji transakcyjnych

Podstawowym celem stosowania ulotnych instancji transakcyjnych jest zapewnienie przezroczystego posługiwania się obiektami przez aplikacje. Dzięki temu nie ma już znaczenia, czy dany obiekt jest ulotny czy trwały — w obu przypadkach jego dane zostaną przywrócone, gdy transakcja zostanie odwołana. Przykładem zastosowanie może być tutaj aplikacja wykorzystująca okno dialogowe umożliwiające konfigurację różnych parametrów. Okno takie może zawierać zarówno trwałe dane o konfiguracji, jak i lokalne, ulotne informacje. Posługując się ulotnymi instancjami transakcyjnymi, implementacja okna dialogowego może po prostu odwołać wszelkie modyfikacje, gdy użytkownik kliknie przycisk *Cancel*.

4.5.3. Instancje nietransakcyjne

Instancje nietransakcyjne umożliwiają obsługę rzadko zmieniających się, niespójnych danych oraz realizację transakcji optymistycznych. W niektórych sytuacjach aplikacja może chcieć jawnie wczytać dane do obiektów, ale nie interesują jej modyfikacje wykonane przez innych klientów tej samej bazy. Instancje nietransakcyjne mogą być również wynikiem zakończonych transakcji.

JDO definiuje cztery różne właściwości zmieniające cykl życia obiektów JDO:

- `NontransactionalRead`
- `NontransactionalWrite`
- `RetainValues`
- `Optimistic`

Aplikacja wybiera jedną lub więcej wymienionych właściwości, aby włączyć opcje działania instancji klasy `PersistenceManager` przed jej pierwszym użyciem. Opcje te stosowane są przede wszystkim ze względu na bezpieczeństwo. Aplikacja może bowiem przypadkowo zmodyfikować obiekt poza kontekstem transakcji, ale modyfikacja taka zostanie utracona lub wczytane zostaną nieprawidłowe dane. A oto przykład sytuacji, w której modyfikacja zostaje zagubiona:

Klient A pobiera instancję.

Klient B pobiera tę samą instancję.

Klient A modyfikuje instancję.

Klient B również modyfikuje instancję.

Klient B zatwierdza transakcję.

Klient A zatwierdza transakcję (zastępując modyfikacje dokonane przez klienta B).

Podobna sytuacja może wystąpić, gdy dane zostają zmodyfikowane pomiędzy dwiema kolejnymi operacjami odczytu:

Klient A zlicza wszystkie wypożyczone książki.

Klient B wypożycza książkę.

Klient B zatwierdza transakcję.

Klient A zlicza wszystkie książki w bibliotece (bez jednej brakującej).

4.5.3.1. Opcja NontransactionalRead

Opcja ta umożliwia aplikacji odczyt pól instancji, która nie jest transakcyjna lub znajduje się w stanie `hollow` poza jakąkolwiek transakcją. Umożliwia również nawigację i wykonywanie zapytań poza aktywną transakcją. Stan umożliwiający obsługę danych pól, które nie są spójne ani transakcyjne na zewnątrz aktywnej transakcji nosi nazwę `persistent-non-transactional`. Aplikacja może wymusić przejście obiektu znajdującego się w stanie `persistent-clean` do stanu `persistent-non-transactional` poprzez wywołanie metody `PersistenceManager.makeNontransactional`. Należy jednak zwrócić uwagę na efekty uboczne i inne problemy związane z odczytem obiektów poza transakcjami.

4.5.3.2. Opcja NontransactionalWrite

Aktywacja tej opcji umożliwia modyfikacje instancji, które nie są transakcyjne na zewnątrz jakichkolwiek transakcji. Jednak modyfikacje takie nie zostają nigdy zapisane w bazie danych. JDO nie definiuje sposobu przejścia instancji do stanu `persistent-non-transactional` na skutek modyfikacji instancji poza transakcją (gdy opcja `NontransactionalWrite` ma wartość `true`).

4.5.3.3. Zastosowania instancji nietransakcyjnych

Z punktu widzenia architektury programu zastosowanie instancji nietransakcyjnych nie jest zalecane z powodu potencjalnych problemów ze spójnością danych. Zasady projektowania związane z obsługą transakcji omówione zostaną w rozdziale 11. Istnieją jednak wyjątkowe sytuacje, w których zastosowanie obiektów nietransakcyjnych ujawnia swoje zalety:

- Gdy aplikacja musi wyświetlać pewne dane statystyczne co pewien czas, ale nie jest istotne, by dane te były aktualne w ściśle określonym momencie. Motywacją takiego rozwiązania może być chęć uniknięcia zakłóceń procesu obliczeniowego przez proces prezentacji. W pewnych warunkach konieczne może okazać się nawet buforowanie wyświetlanych danych i odczytywanie jedynie ich zmian.
- Gdy aplikacja wyposażona w graficzny interfejs użytkownika używa JDO do przechowywania danych o swojej konfiguracji, która zostaje załadowana w określonym momencie przez inną część systemu. Podczas edycji konfiguracji transakcyjne zachowanie danych jest zbędne. Jedynie zmiana całej konfiguracji powinna być atomowa, spójna, izolowana i trwała.

4.5.3.4. Opcja RetainValues

JDO zeruje wszystkie obiekty trwałe, gdy przechodzą one do stanu `hollow`. Umożliwia to pozbycie się zbędnych referencji i odzyskanie zajmowanej pamięci. Jeśli opcja `RetainValues` posiada wartość `true`, to podczas zatwierdzania transakcji obiekty znajdujące się w stanie

persistent-clean, persistent-new lub persistent-dirty przechodzą do stanu persistent-non-transactional zamiast do stanu hollow. Wartości pól zostają zachowane w celu późniejszego wykorzystania. Ich odczyt po zakończeniu transakcji jest dozwolony. W porównaniu do opcji NontransactionalRead po zakończeniu transakcji nie jest możliwa nawigacja po obiektach znajdujących się w stanie hollow i odczyt ich danych.

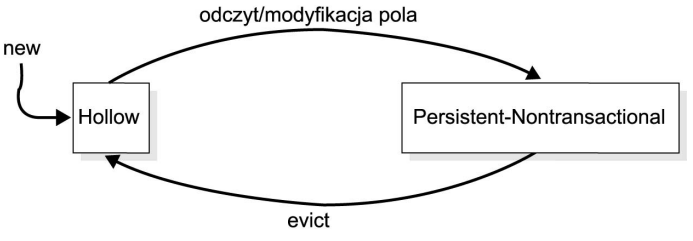
4.5.3.5. Opcja RestoreValues

Jeśli opcja RestoreValues nie ma wartości true, to opisany wyżej efekt działania opcji RetainValues jest możliwy tylko w przypadku zatwierdzenia transakcji. W przypadku odwołania transakcji instancje przejdą bowiem do stanu hollow i tym samym odczyt ich pól nie będzie możliwy (przestaną być dostępne dla aplikacji). Natomiast gdy opcja RestoreValues ma wartość true, to JDO musi odtworzyć poprzednie wartości pól obiektów znajdujących się w stanie persistent-dirty w momencie odwołania transakcji. Odwołanie transakcji sprawia, że obiekty znajdujące się w stanie persistent-clean lub persistent-dirty przechodzą do stanu persistent-non-transactional i tym samym stają się dostępne dla aplikacji.

4.5.4. Transakcje optymistyczne

JDO umożliwia także zarządzanie transakcjami optymistycznymi. Ich działanie przypomina działanie systemu zarządzania wersjami kodu źródłowego, który, zamiast blokować dostęp do poszczególnych dokumentów, umożliwia równoczesną pracę wielu programistów. Warto przy tym zauważyć, że zarządzanie transakcjami optymistycznymi zmienia sposób przejść obiektów trwałych pomiędzy stanami. Rozpoczynając transakcję optymistyczną, aplikacja zakłada, że zablokowanie obiektów niepotrzebnie opóźni dostęp innych użytkowników, dlatego też konflikty pomiędzy różnymi operacjami rozwiązywane są dopiero podczas zatwierdzania transakcji. JDO osiąga to poprzez umożliwienie modyfikacji obiektów trwałych, które zawierają dane transakcyjne. Gdy aplikacja żąda od JDO zatwierdzenia transakcji, to implementacja JDO sprawdza najpierw, czy dane zostały zmodyfikowane w czasie, który minął od ich załadowania (zwykle używając wewnętrznego mechanizmu identyfikacji wersji danych) i jeśli tak, to odwołuje transakcję, sygnalizując aplikacji wystąpienie problemu z wersją danych. W ten sposób możliwy jest jednoczesny dostęp do danych dla wielu klientów, który gwarantuje, że żadne dane nie zostaną utracone. Wadą takiego rozwiązania jest oczywiście to, że aplikacja musi ponownie zażądać zatwierdzenia transakcji, uwzględniając nową, aktualną wersję danych. Z tego powodu mechanizm zarządzania stanami obiektów dopuszcza możliwość odświeżenia danych obiektów znajdujących się w stanie hollow lub non-transactional, zanim aplikacja użyje ich wewnątrz transakcji.

Jeśli aplikacja zdecyduje się używać transakcji optymistycznych, to podczas odczytu pola instancji znajdujących się w stanie hollow przechodzą one do stanu persistent-non-transactional zamiast do stanu persistent-clean (patrz rysunek 4.5). Modyfikacja wartości pola trwałego powoduje przejście instancji do stanu persistent-dirty — tak samo jak w przypadku transakcji pesymistycznych. Aplikacja musi jawnie zażądać odświeżenia danych zapisanych w bazie zanim wykonana zostanie ich aktualizacja na podstawie bazy danych. Zalety i wady transakcji optymistycznych zostaną wyjaśnione w rozdziale 11.



Rysunek 4.5. *Cykl życia obiektu: dostęp poza transakcjami*

4.6. Przykłady

Zamieszczone poniżej przykłady kodu stanowią ilustrację sposobów sprawdzania stanu obiektów oraz zastosowania wywołań zwrotnych w celu identyfikacji typowych problemów. Przedstawiona zostanie klasa pomocnicza umożliwiająca uzyskanie informacji o stanie obiektów, licznie obiektów w pamięci, liczbie obiektów w wybranym stanie i tak dalej. Ponieważ większość producentów implementacji JDO nie udostępnia metod umożliwiających uzyskanie informacji o zawartości buforów podręcznych wykorzystywanych przez instancję klasy `PersistenceManager`, to posiadanie własnego, niezależnego rozwiązania w tym zakresie jest bardzo przydatne.

Wszystkie przedstawione poniżej metody umieszczone zostały w pliku źródłowym *DebugStates.java*.

Pierwsza z omawianych metod umożliwi uzyskanie identyfikatora obiektu w pamięci. Prosty sposób jego uzyskania polega na wywołaniu metody `identityHashCode`. Chociaż uzyskany w ten sposób identyfikator nie jest unikalny, to jednak wystarcza w przypadku większości maszyn wirtualnych i umożliwia śledzenie obiektów znajdujących się w ich pamięci:

```
private static String getMemoryId(Object obj)
{
  long id = System.identityHashCode(obj);
  return Long.toString(id, 16); // wartość w systemie szesnastkowym
}
```

Kolejna metoda będzie zwracać łańcuch, który zawiera zarówno identyfikator obiektu trwałego w pamięci, jak i jego skrót będący wynikiem zastosowania metody mieszającej. Jeśli obiekt nie jest jeszcze trwały, to metoda zwróci jedynie jego identyfikator:

```
private static String getId(Object obj)
{
  if (obj instanceOf PersistentCapable) {
    PersistenceManager pm =
      JDOHelper.getPersistenceManager(obj);
    if (pm != null) {
      return "@" + getMemoryId(obj) +
         " OID:" + pm.getObjectId(obj);
    }
  }
  return "@" + getMemoryId(obj) + " brak OID";
}
```

Aby uzyskać więcej informacji o cyklu życia obiektów, posłużymy się tablicą liczników śledzących liczbę różnych instancji trwałych. Liczniki te będą zwiększane przez metody należące do interfejsu `InstanceCallbacks` oraz konstruktory nie posiadające argumentów, których implementacja JDO używa do tworzenia nowych obiektów w stanie pustym. Ponieważ tablica ta posługuje się słabymi referencjami obiektów, umożliwia również zliczenie obiektów, których zasoby zostały zwolnione przez mechanizm odzyskiwania nieużytków.

Poniżej przedstawiamy przykład klasy, która posługuje się klasą `DebugStates` do zliczania wywołań zwrotnych swoich metod:

```
public class Author
  implements InstanceCallbacks
{
  protected String name;

  public Author( String name ) {
    this.name = name;
  }

  public String toString()
  {
     return "Autor: " + name;
  }

 /*
  * Poniższy konstruktor wywoływany jest w celu
  * tworzenia nowych obiektów w stanie pustym.
  * Konstruktor ten rejestruje owe obiekty w klasie DebugStates.
  */
  private Author()
  {
    DebugStates.constructor(this);
  }

  // Metody wywoływane zwrotnie przez JDO
  // oddelegowane do klasy DebugStates:
  public void jdoPostLoad() {
    DebugStates.postLoad(this);
  }
  public void jdoPreStore() {
    DebugStates.preStore(this);
  }
  public void jdoPreClear() {
    DebugStates.preClear(this);
  }
  public void jdoPreDelete() {
    DebugStates.preDelete(this);
  }
}
```

Metody klasy `DebugStates`, do których klasa `Author` oddelegowała metody wywoływane zwrotnie, aktualizują odpowiedni element tablicy w oparciu o identyfikator obiektu. Jeśli element tablicy dla takiego obiektu jeszcze nie istnieje, to zostaje dodany:

```
private static ObjectTable objectTable = new ObjectTable();

private static ObjectTableEntry tableEntry(Object obj)
{
  return objectTable.register(obj);
}

/*
 * Poniższa metoda powinna być wywoływana przez konstruktor
 * klasy zdolnej do trwałości, który nie posiada argumentów.
 */

  public static void constructor(Object obj)
  {
    tableEntry(obj).constructor++;
  }

  //... odpowiedniki metod interfejsu InstanceCallbacks ...

  public ObjectTableEntry register(Object obj)
  {
    Long id = new Long(System.identityHashCode(obj));
    ObjectTableEntry entry = (ObjectTableEntry) table.get(id);
    if (entry == null) {
      entry = new ObjectTableEntry(id, obj);
      table.put(id, entry);
    }
    return entry;
  }
```

Gdy każdy obiekt reprezentowany jest w tablicy, to łatwo możemy policzyć wszystkie obiekty lub obiekty pewnego rodzaju. Poniższa metoda pozwala uzyskać informacje o zużyciu pamięci przez obiekty trwałe:

```
public String getStatistics()
{
  int constructors = 0;
  int postLoad = 0;
  int preStore = 0;
  int preClear = 0;
  int preDelete = 0;
  int objects = 0;
  int gced = 0;
  for (Iter i = new Iter(); i.hasNext(); ) {
    ObjectTableEntry e = i.next();
    objects++;
    if (e.get() == null) gced++;
    constructors += e.constructor;
    postLoad += e.postLoad;
    preStore += e.preStore;
    preClear += e.preClear;
    preDelete += e.preDelete;
  }
  String NL = System.getProperty("line.separator");
  String result =
    "liczba obiektów:            " + objects + NL +
    "zwolnionych jako nieużytki: " + gced + NL +
```

```
    "wywołań konstruktorów:      " + constructors + NL +
    "wywołań postLoad:           " + postLoad + NL +
    "wywołań preStore:           " + preStore + NL +
    "wywołań preClear:           " + preClear + NL +
    "wywołań preDelete:          " + preDelete + NL;
  return result;
}
```

Wynik działania tej metody może być interesujący już w przypadku utworzenia dwóch obiektów przez następujący fragment kodu:

```
pm.currentTransaction().begin();
Author a = new Author("Heiko Bobzin");
Book b = new Book("Core JDO", a);
```

Wywołanie metody `getStatistics` spowoduje w tym przypadku wyświetlenie następujących informacji:

```
liczba obiektów:           2
zwolnionych jako nieużytki: 0
wywołań konstruktorów:     2
wywołań postLoad:          0
wywołań preStore:          0
wywołań preClear:          0
wywołań preDelete:         0
```

Chociaż konstruktor bez argumentów nie został jawnie wywołany przez przedstawiony wyżej fragment kodu, to jednak klasa `DebugStates` wyśledziła utworzenie dwóch obiektów. Obiekty te używane są przez implementację JDO jako obiekty fabryki służące do tworzenia instancji klas `Author` i `Book`. Wywołanie konstruktora bez argumentów zostało dodane podczas rozszerzania kodu bajtowego z powodów związanych z bezpieczeństwem, które omówione zostaną w rozdziale 10.

Podczas utrwalania obu obiektów (w tym obiektu klasy `Author` poprzez osiągalność z obiektu klasy `Book`):

```
pm.makePersistent(b);
pm.currentTransaction().commit();
```

wywołane zostaną zwrotnie dla każdego z nich metody `preStore` i `preClear`:

```
liczba obiektów:           2
zwolnionych jako nieużytki: 0
wywołań konstruktorów:     2
wywołań postLoad:          0
wywołań preStore:          2
wywołań preClear:          2
wywołań preDelete:         0
```

Kolejny test spowoduje wyświetlenie liczby obiektów zawartych w ekstensji klasy `Book`:

```
pm.currentTransaction().begin();
Extent e = pm.getExtent(Book.class, true);
Iterator iter = e.iterator();
while (iter.hasNext()) {
  Book a = (Book)iter.next();
  String title = a.title; // tworzy pusty obiekt klasy Author!
}
```

Jeśli baza zawierała po 20 instancji każdej z klas `Book` i `Author`, to utworzone zostały 42 obiekty (w tym dwa jako fabryki). Metody `postLoad` i `preClear` wywołane zostały jedynie dla instancji klasy `Book`, ponieważ pozostałe obiekty znajdowały się w stanie pustym:

```
liczba obiektów:            42
zwolnionych jako nieużytki: 10
wywołań konstruktorów:      42
wywołań postLoad:           20
wywołań preStore:           0
wywołań preClear:           20
wywołań preDelete:          0
```

4.7. Podsumowanie

W rozdziale tym omówiliśmy cykl życia obiektów JDO niezbędny do zrozumienia dynamicznego aspektu instancji trwałych i transakcyjnych. Przedstawiliśmy również sposoby uzyskania informacji o stanie obiektów oraz zastosowania metod wywoływanych zwrotnie. Omówione zostały również opcjonalne stany obiektów JDO oraz przedstawiona została przydatna klasa umożliwiająca śledzenie stanu obiektów JDO.

Znajomość cyklu życia obiektów JDO jest konieczna do zrozumienia bardziej zaawansowanych problemów związanych z zastosowaniami JDO. Dodatek A zawiera kompletną tabelę przejść pomiędzy stanami obiektów JDO.