ORM w Javie

Adam Michalik
2007

ORM

- ORM
- O/RM
- O/R mapping
- Object-Relational Mapping

- Mapowanie obiektowo relacyjne
- Odwzorowanie obiektowo-relacyjne

Co to jest ORM?

Odwzorowanie obiektowo-relacyjne pozwala przechowywać struktury obiektowe w relacyjnych bazach danych

O czym będziemy rozmawiać

- Po co nam ORM?
- Krótko o przydatnych nam elementach Javy (wersja 5+)
  - Adnotacje
  - Serializacja obiektów
- Trochę o Javie EE
  - Kontener/serwer aplikacji
  - Enterprise JavaBeans (EJB) 3.0
- Gwóźdź programu: Java Persistence API

Po co nam ORM?

Chcemy przechowywać obiekty w relacyjnej bazie danych.

Obiekty:

- mają atrybuty
  - typy „proste” (liczby, napisy, daty)
  - referencje do innych obiektów
- są w hierarchii klas
- są w relacjach (mają do siebie referencje)

Po co nam ORM?

ORM pozwala programiście na przezroczyste posługiwanie się bazą danych w systemach obiektowych:

- Jedyne, na czym programiście zależy to żeby obiekty, którymi się posługuje, były "trwałe" (ang. persistent). Jeśli robi zmiany, to chce, żeby te zmiany się zachowywały. Programista nie chce zajmować się niskopoziomową obsługą DB - łączeniem z DB, operacjami na relacjach, tabelach, pisaniem zapytań SQL itp. To jest odpowiedzialność ORM managera.

Po co nam ORM?

ORM pozwala programiście na przezroczyste posługiwanie się bazą danych w systemach obiektowych:

- Baza danych automatycznie dostosowuje się do modelu danych - odwzorowywanych klas, atrubutów obiektów, relacji między obiektami. Skoro model obiektowy ma już raz utworzone zależności (w kodzie), to nie ma potrzeby tworzyć tych zależności drugi raz w DB - mogą zostać wygenerowane automatycznie.

Po co nam ORM?

ORM pozwala programiście na przezroczyste posługiwanie się bazą danych w systemach obiektowych:

- ORM manager oraz serwer aplikacji zarządzają optymalizacjami wydajnościowymi (np. pulą połączeń do bazy, cachem obiektów)
- Programista nie musi wiedzieć, co to za baza ani jakie są jej specyficzne cechy – ORM manager przekształca uniwersalny obiektowy język zapytań na SQL, a sterowniki pozwalają mu na obsługę tej konkretnej bazy danych

Java 5+ - adnotacje (*annotations*)

Adnotacje:

- nowa rodzina obiektów w Javie, koncepcyjnie inna niż klasy i interfejsy
- metainformacje zapisane na stałe w kodzie
- wartości muszą być stałe w fazie kompilacji (literały, typy wyliczeniowe)
- stosowane do: klas, atrybutów, metod
- używane podczas przetwarzania: plików źródłowych, plików skompilowanych (.class) i obiektów w czasie działania
- nie wpływają stricte na działanie kodu ale na obsługę obiektów/klas przez inne narzędzia

Java 5+ - adnotacje (*annotations*)

Przykład deklaracji klasy adnotacji

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date() default "[unimplemented]";
}
```

Java 5+ - adnotacje (*annotations*)

Klasa adnotacji deklaruje metody, które muszą zwracać:

- typy proste (`int, char, long, double, boolean`...)
- String
- obiekt klasy (Class)
- typy wyliczeniowe (enum)
- adnotacje
- tablice powyższych

Java 5+ - adnotacje (*annotations*)

Metody adnotacji:

- nie mogą rzucać wyjątków
- nie mogą przyjmować parametrów
- mogą mieć domyślne wartości (po słowie kluczowym `default`)

Java 5+ - adnotacje (*annotations*)

Adnotacja – marker

- deklaracja

```
public @interface Preliminary { }
```

- użycie

```
@Preliminary public class TimeTravel
{ ... }
```

Java 5+ - adnotacje (*annotations*)

Adnotacja z pojedynczą wartością

- deklaracja

```
public @interface Copyright {
    String value();
}
```

- użycie

```
@Copyright("2002 Yoyodyne Propulsion
Systems")
public class OscillationOverthruster
{ ... }
```

Java 5+ - adnotacje (*annotations*)

Przykład użycia „wszystko na raz”:

```
@A1(v1=”aaa”, v2=123, v3=Object.class)
public class MyClass {
   @A2 Object o;
   @A3(”ddd”)
   public Object getO() {
      return o;
   }
}
```

Java 5+ - adnotacje (*annotations*)

Plusy adnotacji:

- nie mamy osobnego pliku z metainformacjami, który musimy osobno pielęgnować i który może nam się rozspójnić z kodem
- są w tym samym języku, co kod
- w wielu przypadkach można zrezygnować z poprzednio stosowanego dziedziczenia po Bardzo Specjalnych Klasach lub implementowania Bardzo Specjalnych Interfejsów i używać zwykłych, adnotowanych obiektów (POJOs – Plain Old Java Objects)

Java 5+ - adnotacje (*annotations*)

Minusy adnotacji:

- żeby zmienić metaiformacje, trzeba przekompilować kod

Java 5+ - adnotacje (*annotations*)

Rozwiązanie:

- część danych w adnotacjach, część w deskryptorze XML (stałe, statyczne dane w @, zmienne, dynamiczne w XML):
  - deskryptor ma większy priorytet od anotacji
  - daje pewną elastyczność kosztem spójności
- pozwala osobie składającej aplikację ustawić cechy konfigurowalne, a programiście cechy stałe (np. fakt bycia w relacji 1-1 lub 1-*)

Java – serializacja obiektów

- Java umożliwia zamianę stanu obiektu na ciąg bajtów - serializację.
- Serializacja jest przydatna gdy:
  - chcemy trwale zapisać obiekt (w pliku, w bazie danych)
  - chcemy przesyłać obiekt przez sieć
- Żeby obiekt był serializowalny, musi implementować pusty interfejs Serializable.

Java – serializacja obiektów

- Domyślnie serializowany jest cały graf zależności – obiekt serializowany i wszystkie jego atrybuty. Ale atrybuty też musza być serializowalne.
- Jeśli nie chcemy serializować jakiegoś atrybutu, oznaczamy go słowem kluczowym `transient`:
  ```
  private transient Object myAttr;
  ```
  Przy deserializacji ustawiane na domyślną wartość (`0`, `null`, `false`).
- Możemy sami zakodować ręcznie sposób serializacji
  ```
  private void writeObject(ObjectOutputstream os)
  private void readObject(ObjectInputstream os)
  ```

Java – serializacja obiektów

- W metodach readObject i writeObject kodujemy domyślną (de)serializację (`defaultWriteObject`, `defaultReadObject`)i ręczne ustawianie atrybutów nieserializowanych
- Klasy dziedziczące po klasach serializowalnych są serializowalne
- Klasy serializowalne dziedziczące po klasach nieserializowalnych wołają ich nieprywatny bezargumentowy konstruktor. Jeśli chcemy dodatkowo poustawiać coś w nadklasach, musimy zrobić to ręcznie.

Java – serializacja obiektów

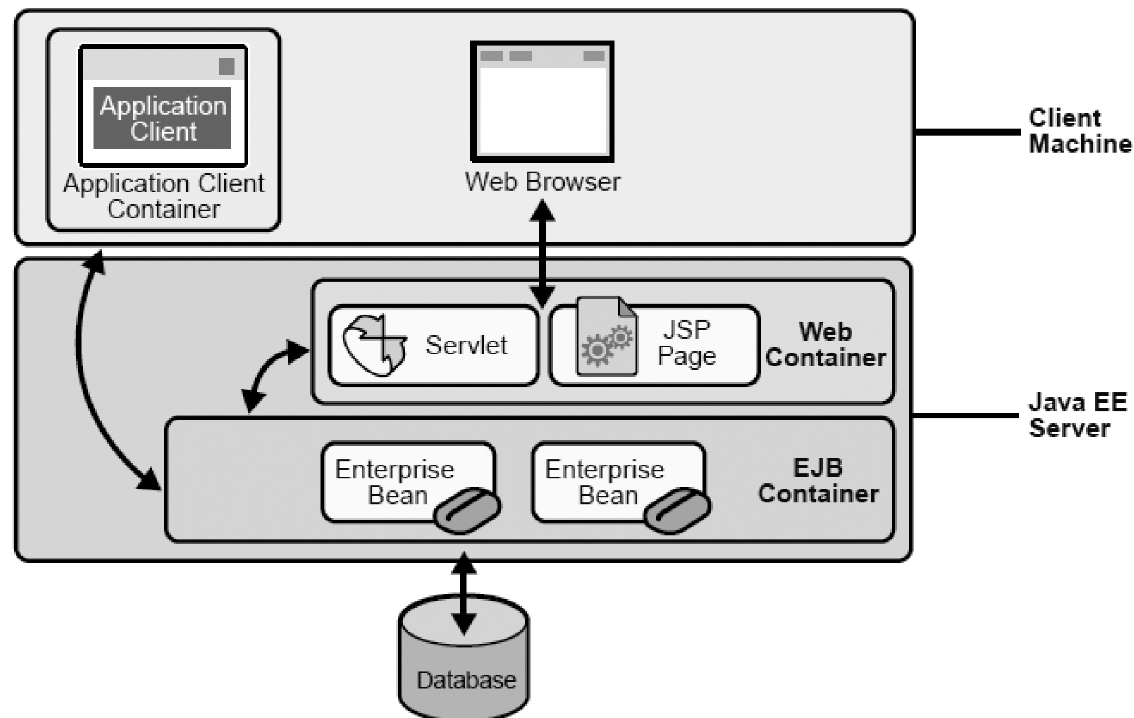
- Serializowane są tylko atrybuty instancji, nigdy klasy (i lepiej tego nie kodować ręcznie)

Java EE

- Java EE – zbiór standardów (API) dla programowania dużych aplikacji biznesowych
- Różne firmy mają różne implementacje tych API
- Java Persistence API (JPA) jest częścią Java EE
- Najczęściej aplikacje biznesowe korzystają z usług serwera aplikacji, który udostępnia dużo ciekawych funkcji. Koduje się pod API, więc można zmieniać producentów serwerów.

Java EE - kontener

- Kontener to część serwera aplikacji odpowiedzialna za obsługę pewnego rodzaju usług



Java EE (5.0) – kontener

Kontener udostępnia m.in. usługi:

- **EJB 3.0**
- **JPA**
- **JTA**
- JDBC
- RMI-IIOP
- JNDI
- Java IDL
- Różne usługi dla technologii XML
- JMS
- Web Services
- JavaMail

EJB 3.0

Enterprise JavaBeans

EJB – logika aplikacji

Instancjami EJB zarządza kontener – nie tworzymy przez `new`

EJB:

- Stateless Session Bean
- Stateful Session Bean
- Message-driven Bean

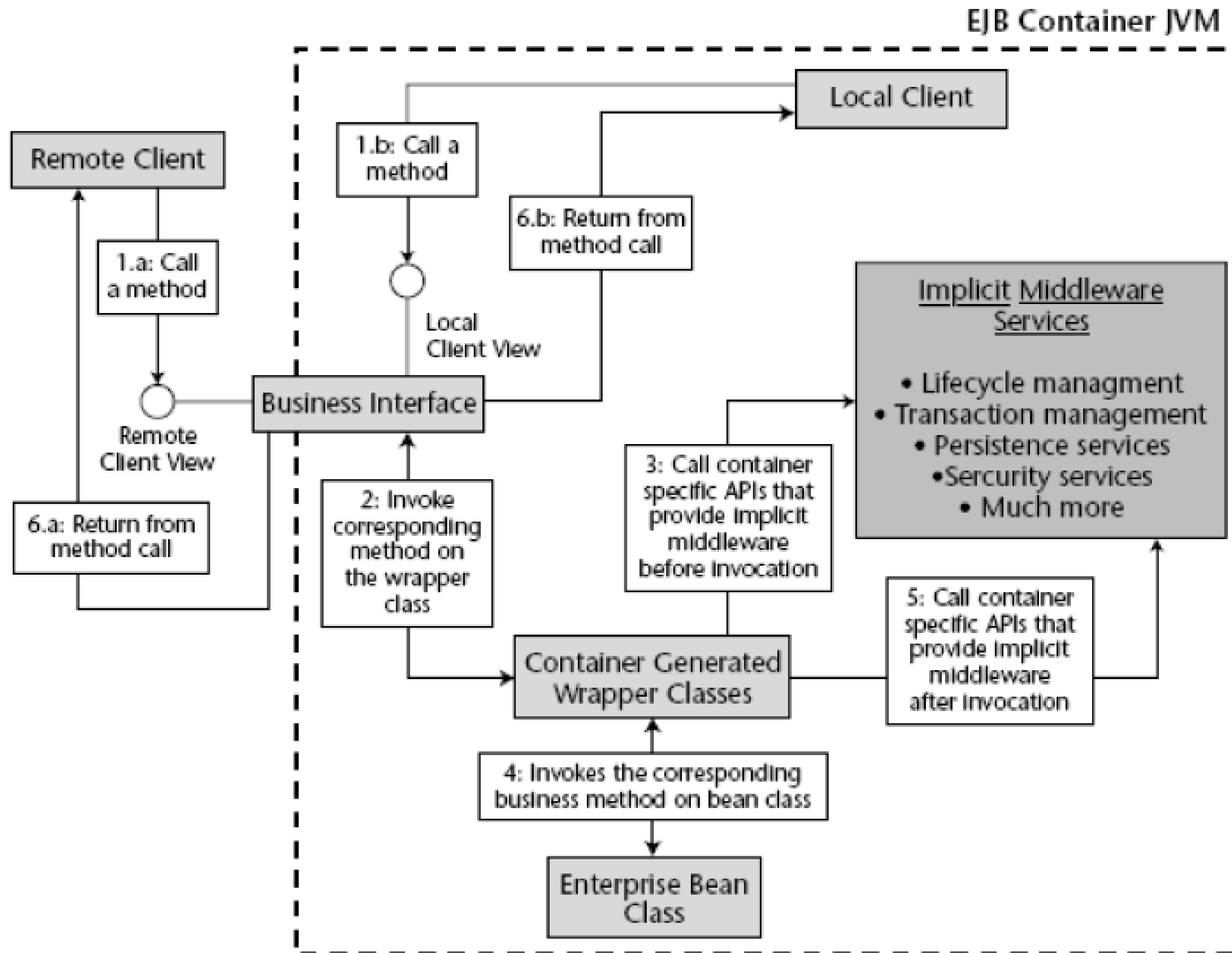
EJB 3.0

EJB to para: interfejs + klasa

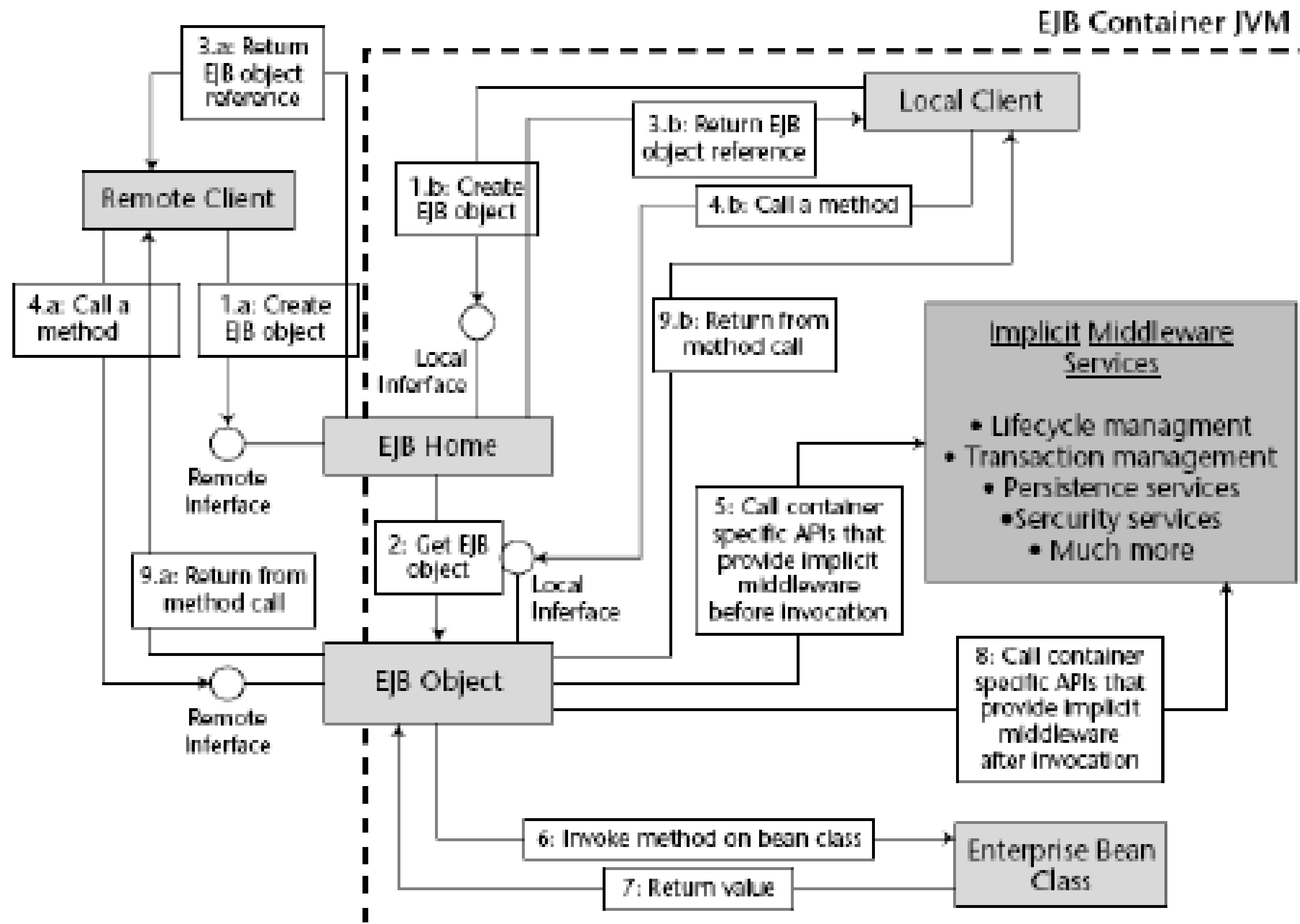
Sens EJB:

- Kontener tworzy proxy, do którego odwołujemy się przez interfejs
- Dzięki temu kontener może m.in.:
  - trzymać pulę utworzonych obiektów (oszczędność czasu na konstrukcję/destrukcję i pamięci)
  - zarządzać wątkami i połączeniami przez sieć
  - równoważyć obciążenie na wiele serwerów
  - automatycznie „wstrzykiwać” zależności
  - zarządzać transakcjami

EJB 3.0



EJB 3.0 vs. EJB 2.1



JPA

- Obiekty podlegające JPA to encje (*entities*)
- Encje nie są niczym szczególnym – to zwykłe obiekty (POJO) z pewnymi ograniczeniami
- Żeby obiekt był encją w rozumieniu JPA:
  - musi być oznaczony jako @Entity
  - musi mieć ID
  - musi mieć publiczny albo chroniony bezargumentowy konstruktor
  - nie może być z klasy '`final`' ani mieć '`final`' atrybutów

JPA

Jakie atrybuty może mieć encja:

- typy proste i ich wrappery
- tablice bajtów i znaków oraz ich wrapperów
- typy serializowalne
- daty (`java.util.Date, java.util.Calendar...`)
- typy wyliczeniowe (enum)
- encje
- kolekcje i mapy powyższych

JPA

Prosta encja:

```
@Entity public class Person {
  @Id private BigInteger pesel;
  private String name;
  private String surname;

}
```

JPA

Prosta encja:

```
@Entity public class Person {
  private BigInteger pesel;
  private String name;
  private String surname;

  @Id public getPesel() {
    return pesel;
  }
  //getters and setters
}
```

JPA - dziedziczenie

- Encje mogą być abstrakcyjne (`abstract class`)
- Encje mogę dziedziczyć po encjach i zwykłych klasach
- Z encji mogą dziedziczyć encje i zwykłe klasy
- Zwykłe klasy dziedziczące z encji nie są trwałe
- Jeśli encja dziedziczy ze zwykłej klasy, to może atrybuty z tej klasy:
  - tracić (domyślne zachowanie) – adnotacje JPA w nadklasie są nieważne
  - mapować do encji – tylko gdy nadklasa jest @MappedSuperclass; adnotacje JPA są ważne

JPA - dziedziczenie

Jak zrealizować hierarchię encji w relacyjnej DB?

- @Inheritance(strategy=InheritanceType.???)
- SINGLE_TABLE – jedna wielka tabela na całą hierarchię klas. Szybka, ale marnuje miejsce. Klasy rozróżniane po dodatkowej kolumnie (*discriminator*)
- TABLE_PER_CLASS – każda klasa ma swoją tabelę, a w niej całość informacji – mało polimorficzne, niska wydajność zapytań do nadklas – trzeba użyć UNION
- JOINED – każda klasa ma tabelę tylko ze swoimi polami. Polimorficzne, ale trzeba użyć JOIN

JPA - klucze

- Klucze encji mogą być:
  - proste (@Id): pojedynczy atrybut encji typu prostego całkowitego, jego wrappera, String lub Date
  - złożone: kilka atrybutów encji lub obiekt
- Atrybuty kluczy złożonych mają takie same ograniczenia jak typy proste
- Klucze proste całkowitoliczbowe można generować automatycznie (@GeneratedValue)

```
@Entity
  public class Employee {
  @Id @GeneratedValue
  public Long getId() {
     return id;
  }
  ...
  }
```

JPA – klucze złożone

- Kluczem złożonym jest obiekt osobnej klasy
- Klucz złożony można reprezentować w encji jako:
  - referencja do obiektu (@EmbeddedId + klucz @Embeddable)
  - atrybuty klasy klucza (@Id + @IdClass)

JPA – klucze złożone

Referencja do obiektu

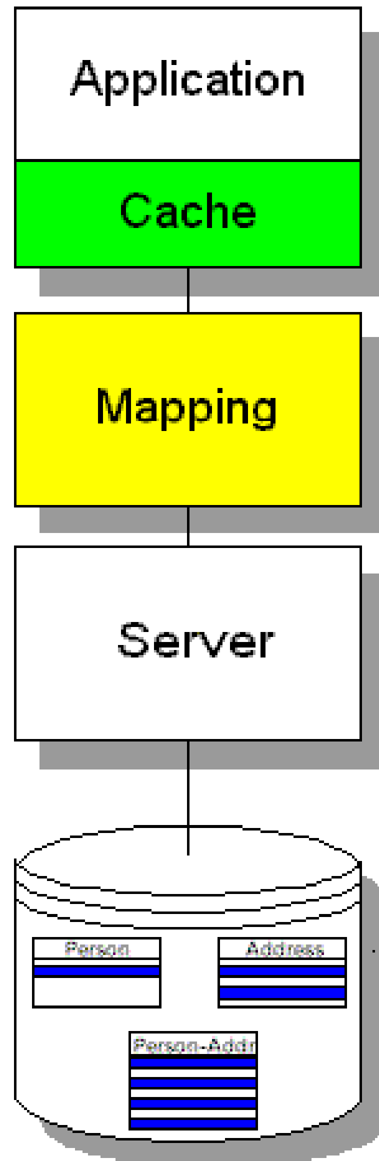
```
@Embeddable
public class EmployeePK implements Serializable {
    private String name;
    private long id;
//... gettery, settery, no-arg konstruktor, equals() i hashcode()
}

@Entity public class Employee implements Serializable
{
    EmployeePK primaryKey;
    public Employee(){}

    @EmbeddedId
    public EmployeePK getPrimaryKey(){...}

    public void setPrimaryKey(EmployeePK pk){...}

    ...
}
```

JPA – klucze złożone

Atrybuty klasy klucza

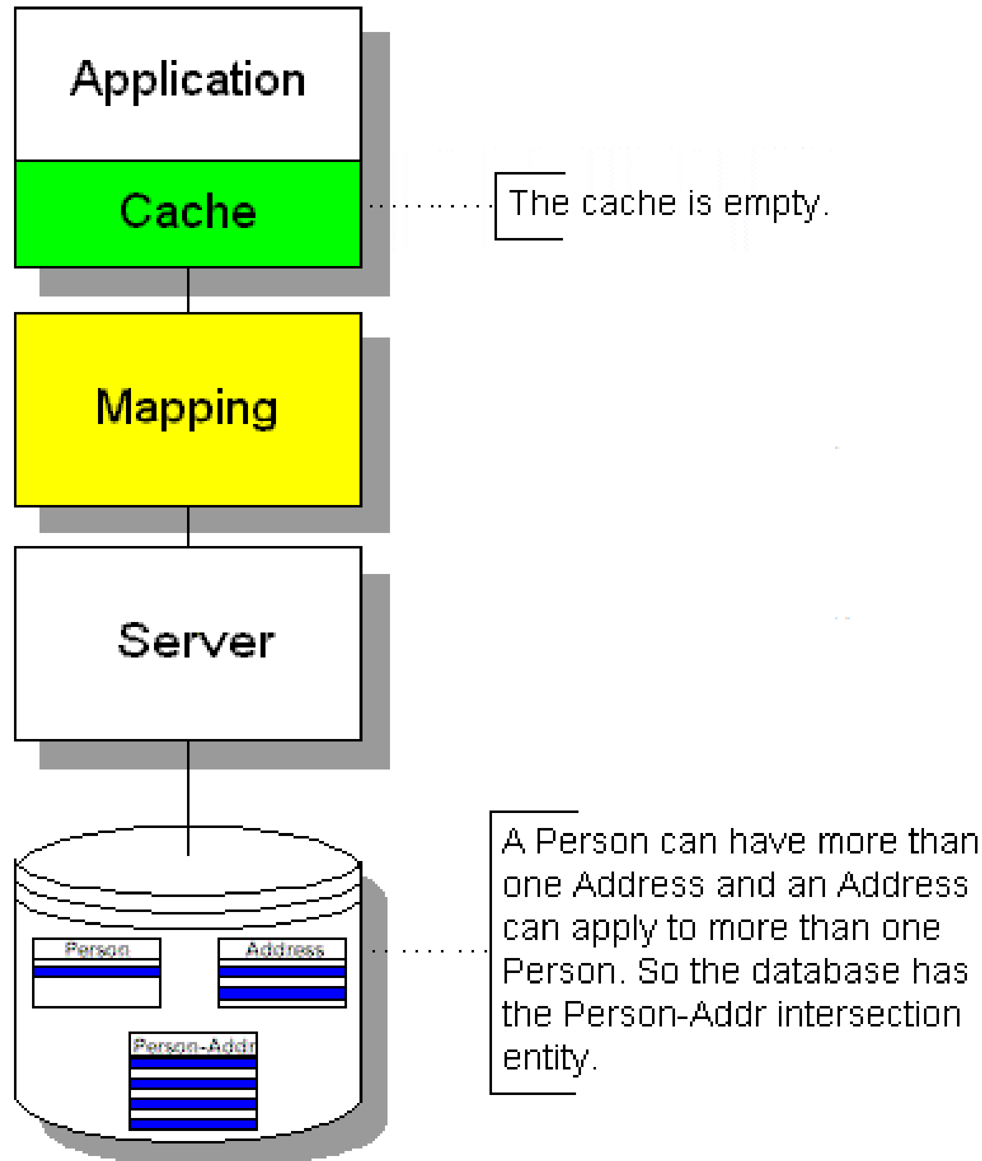
```
public class EmployeePK implements Serializable {
    private String empName;
    private Date birthDay;
//... gettery, settery, no-arg konstruktor, equals() i hashcode()
}

@IdClass(EmployeePK.class)
@Entity
public class Employee
{
    @Id String empName;
    @Id Date birthDay;
...
}
```

JPA – jak działa

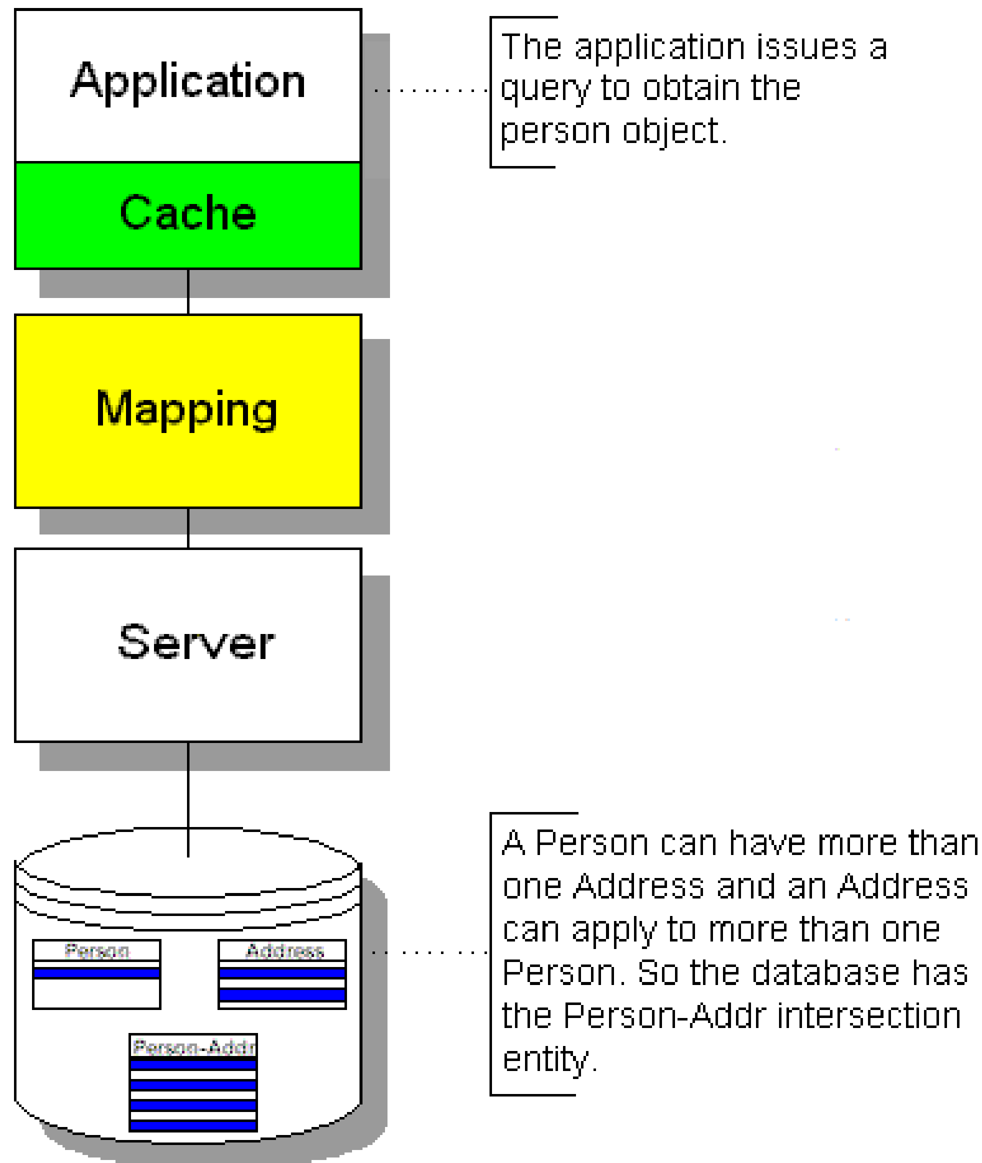


A Person can have more than one Address and an Address can apply to more than one Person. So the database has the Person-Addr intersection entity.

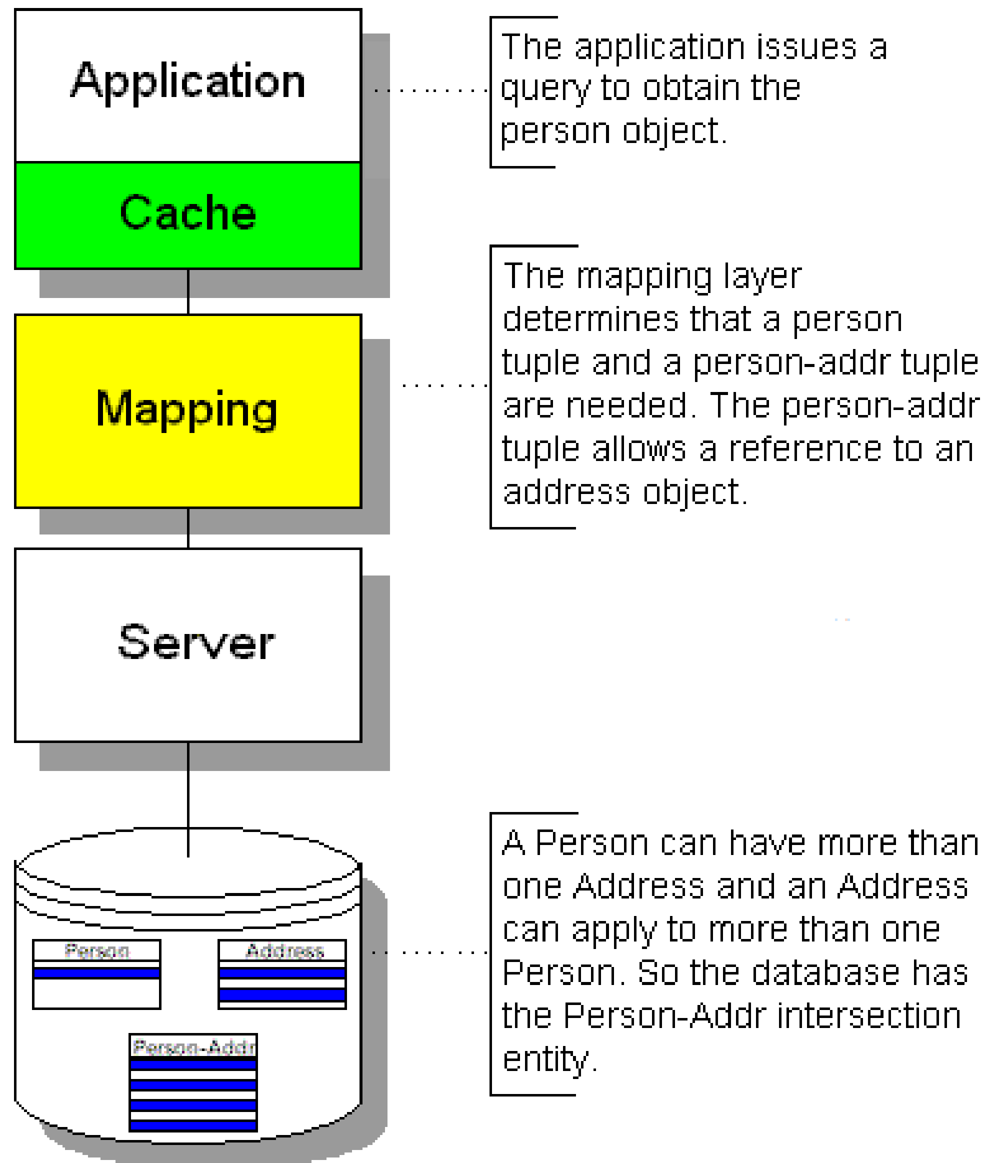
JPA – jak działa



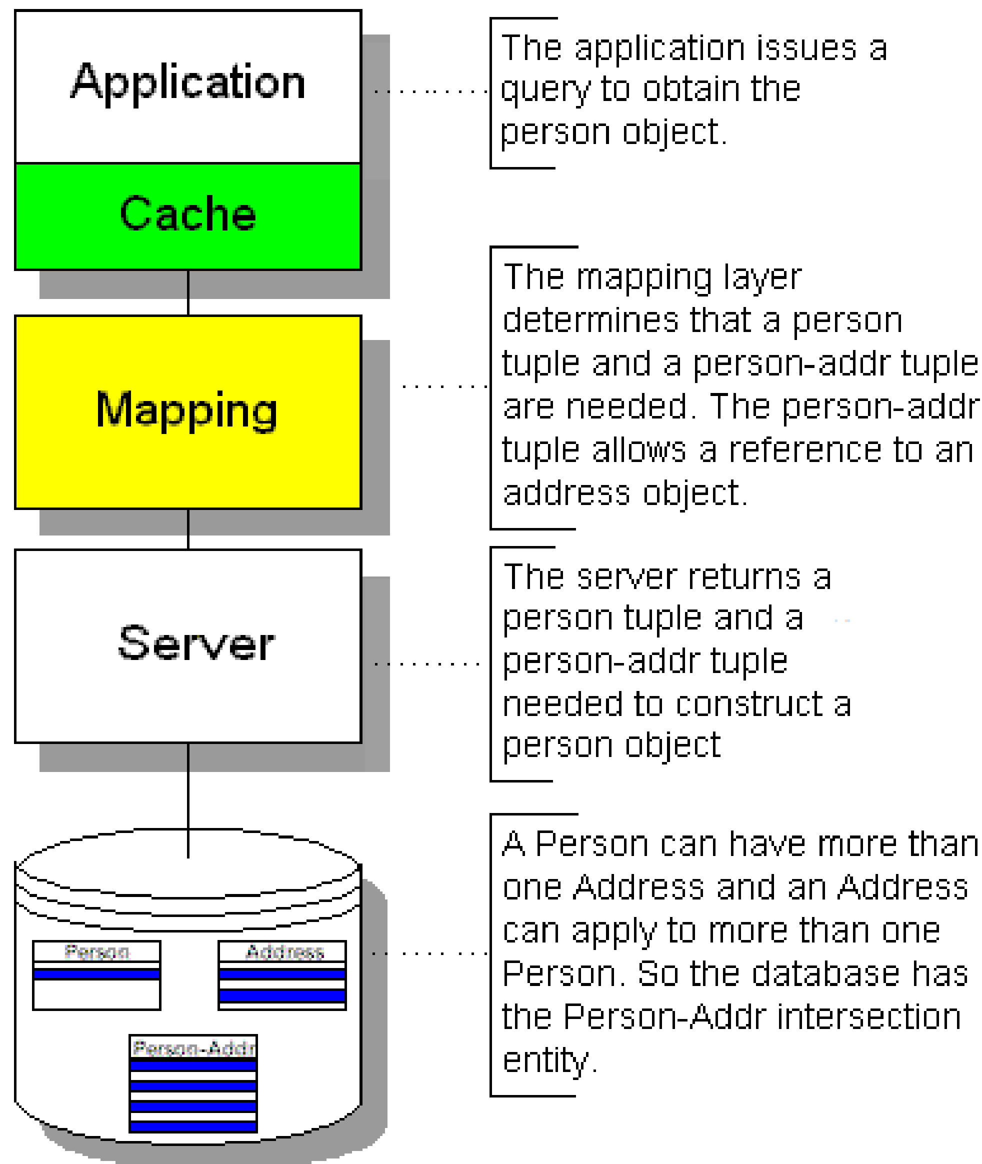
JPA – jak działa



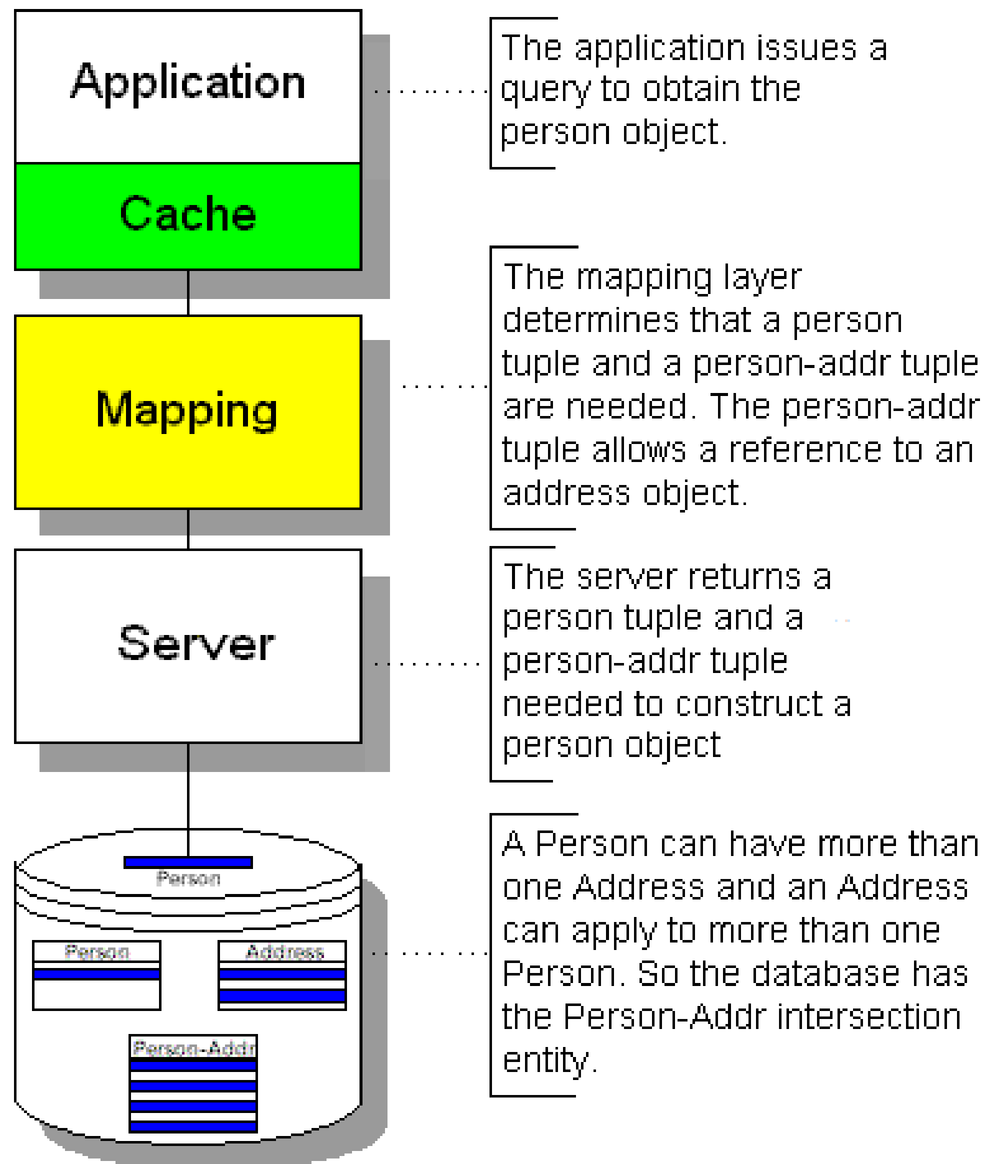
JPA – jak działa



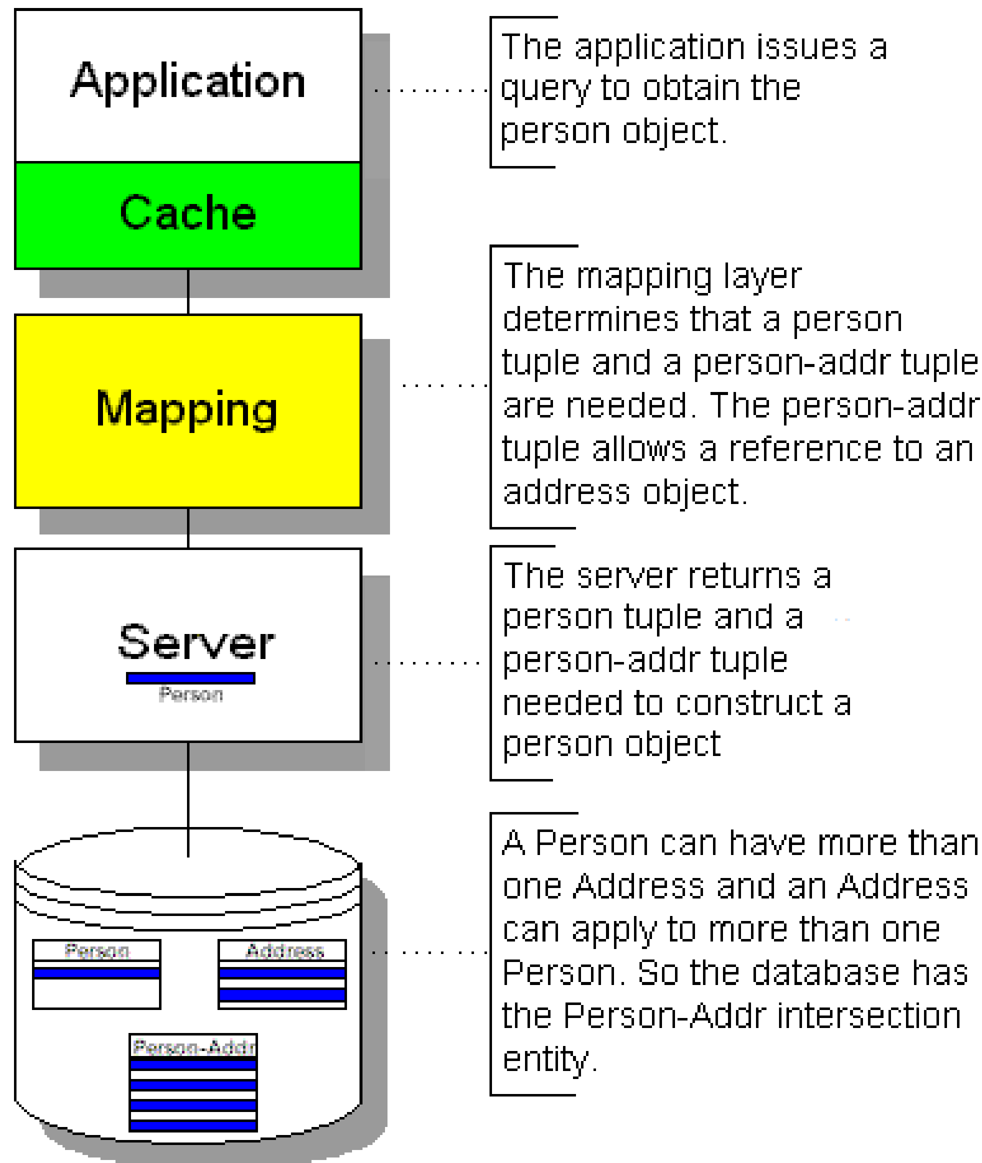
JPA – jak działa



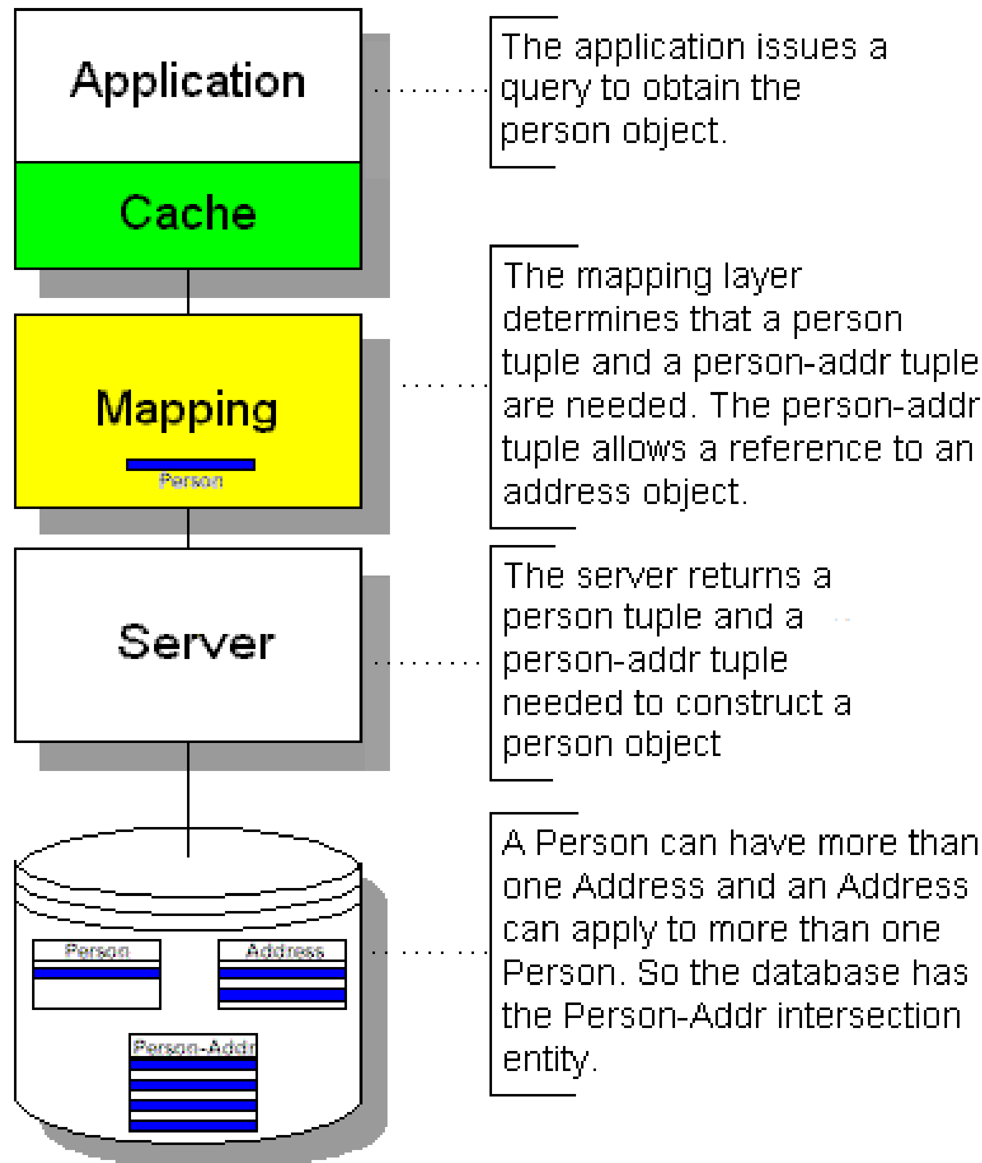
JPA – jak działa



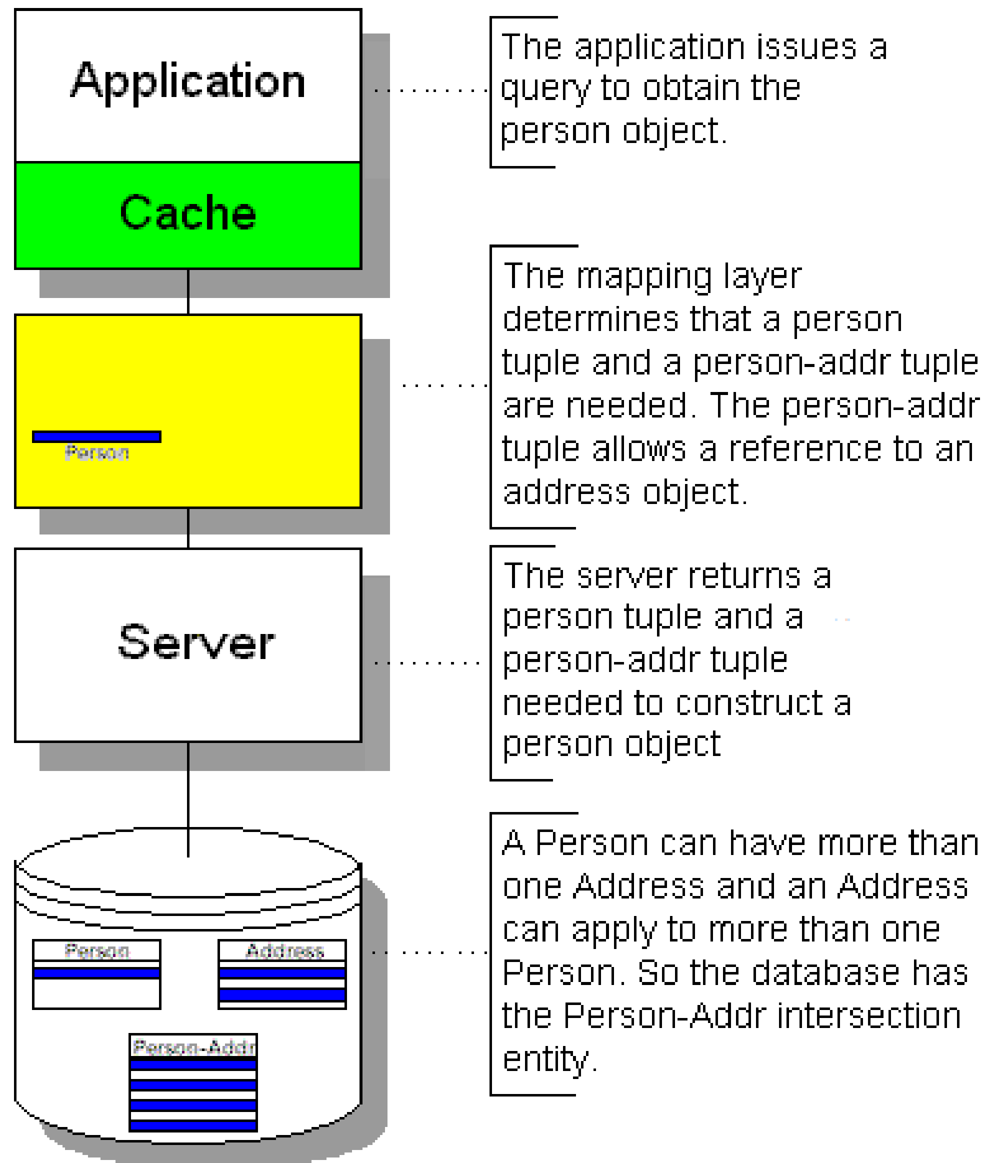
JPA – jak działa



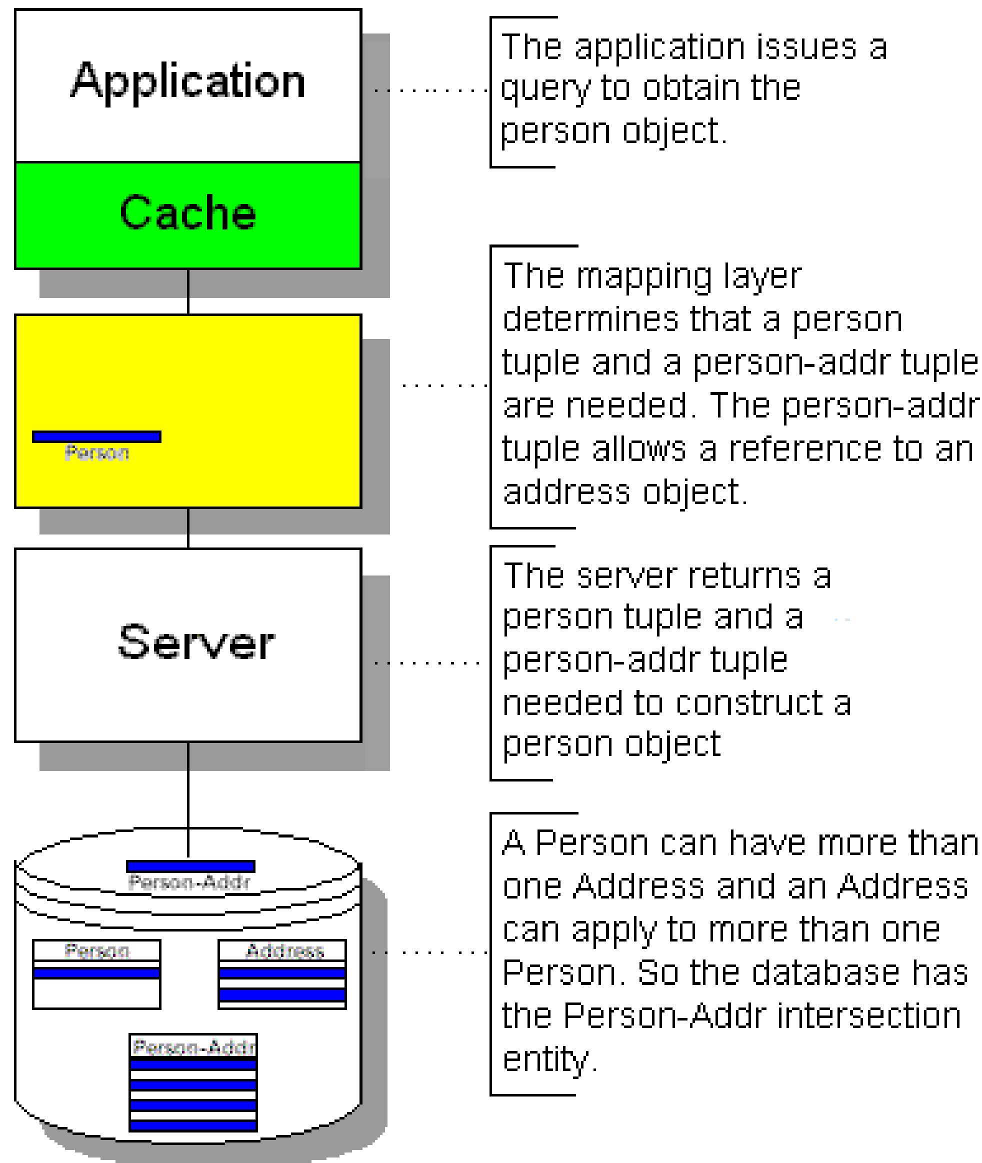
JPA – jak działa



JPA – jak działa



JPA – jak działa



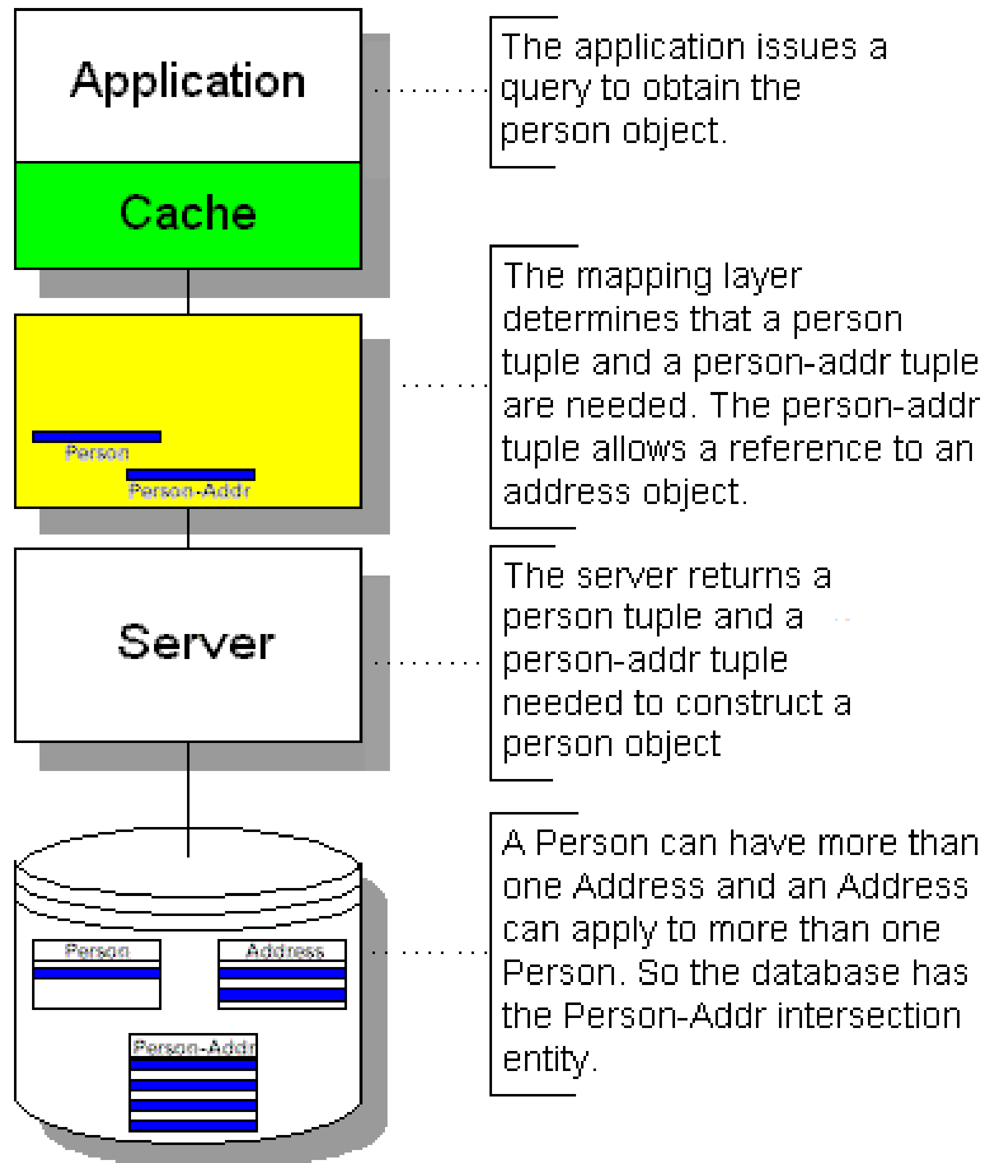
The application issues a query to obtain the person object.

The mapping layer determines that a person tuple and a person-addr tuple are needed. The person-addr tuple allows a reference to an address object.

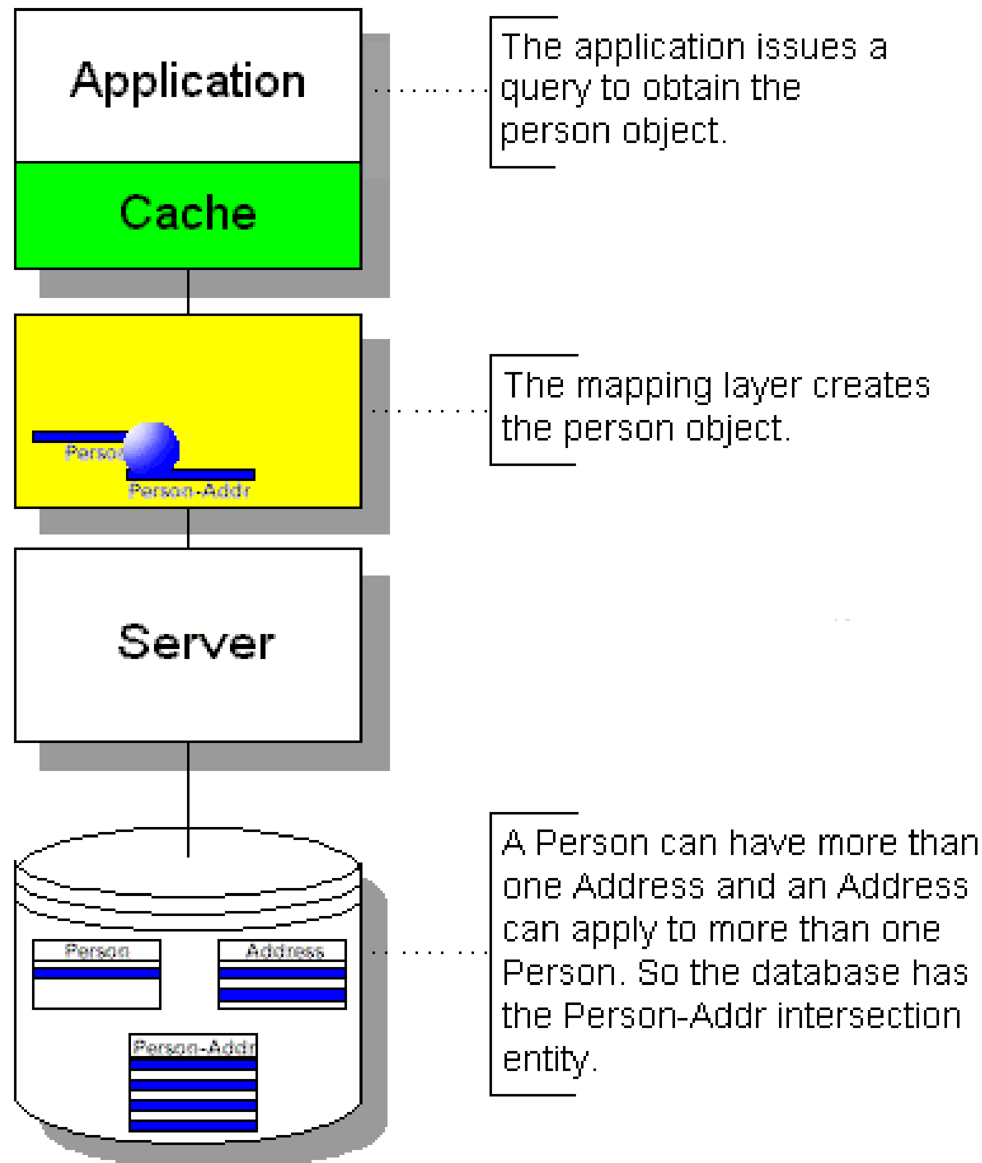
The server returns a person tuple and a person-addr tuple needed to construct a person object

A Person can have more than one Address and an Address can apply to more than one Person. So the database has the Person-Addr intersection entity.

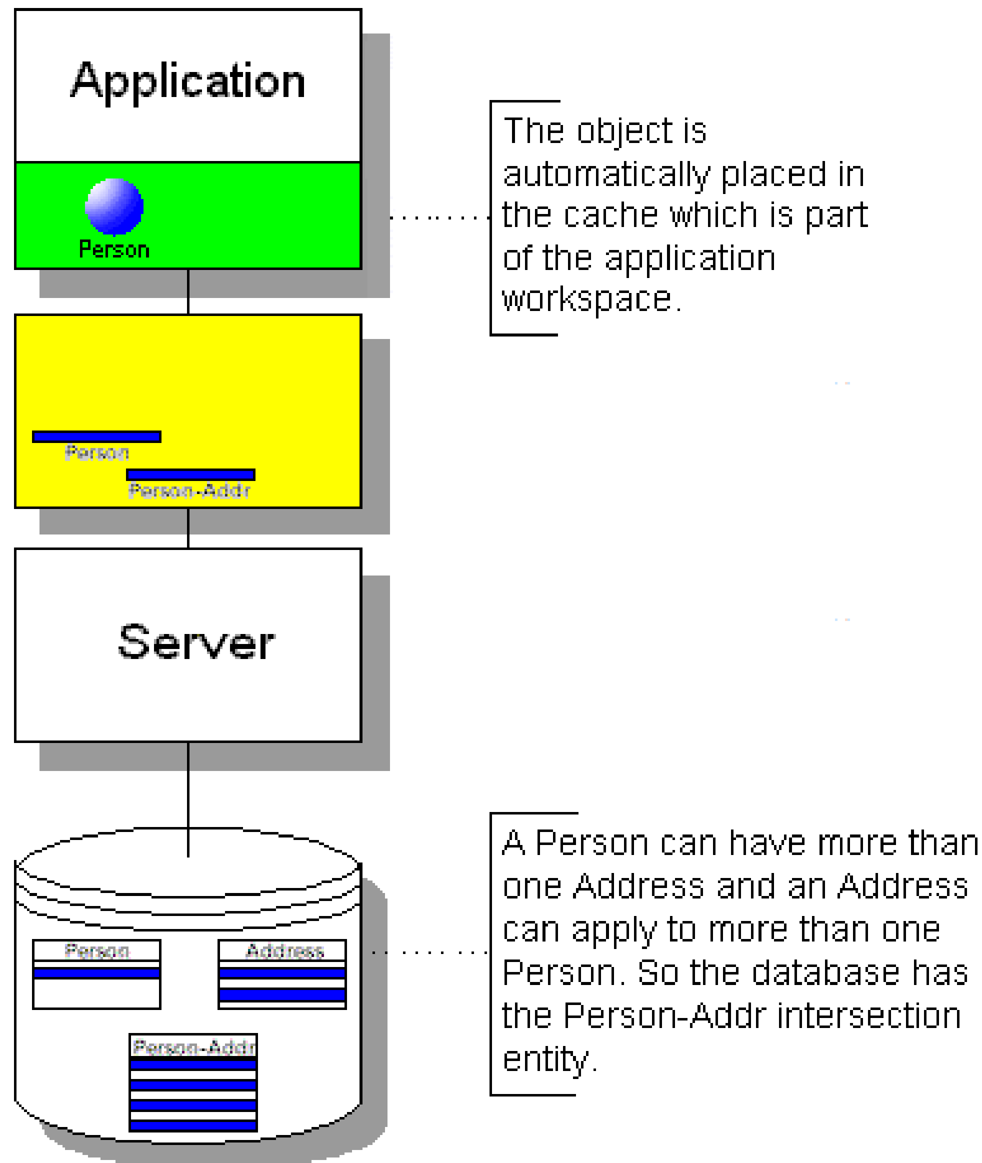
JPA – jak działa



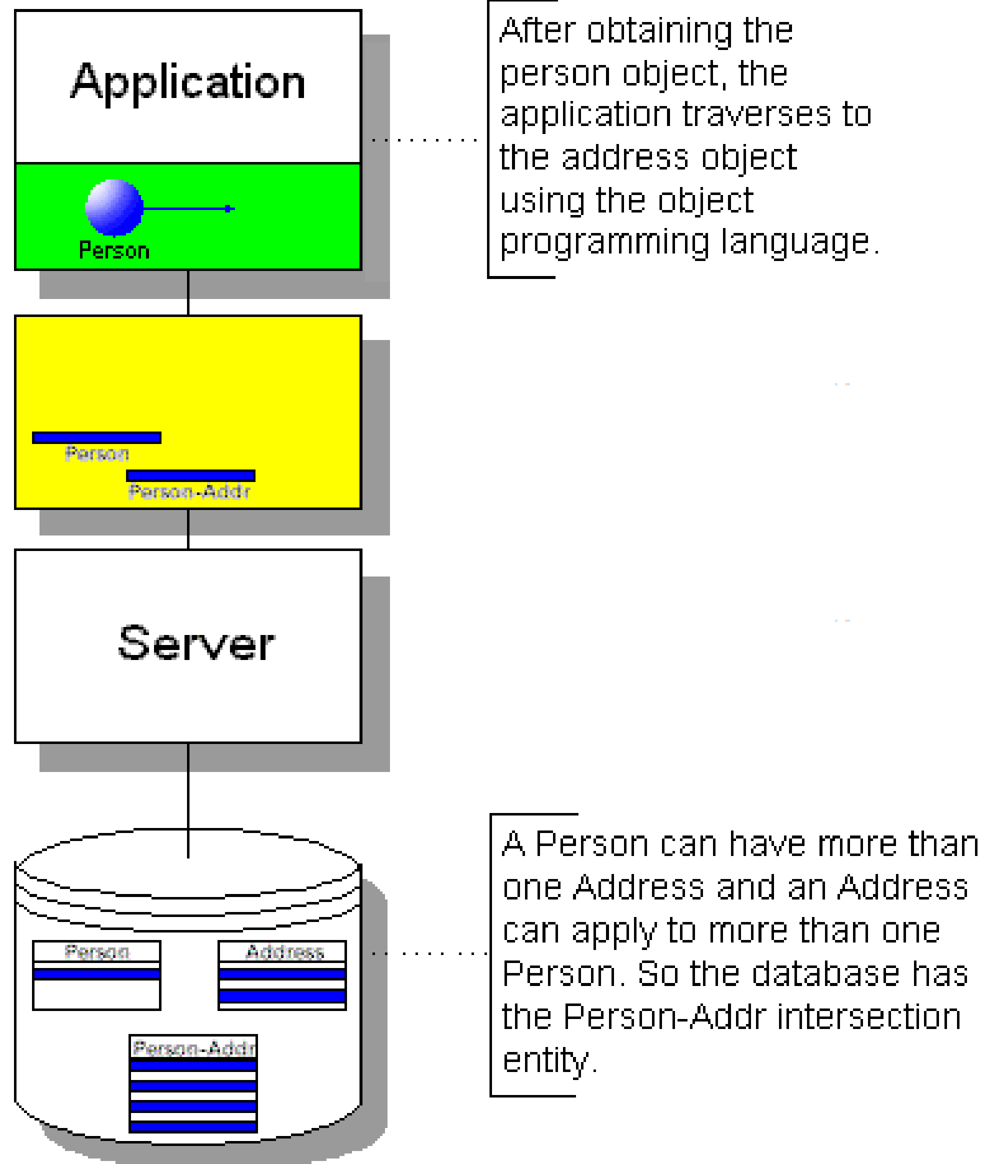
JPA – jak działa



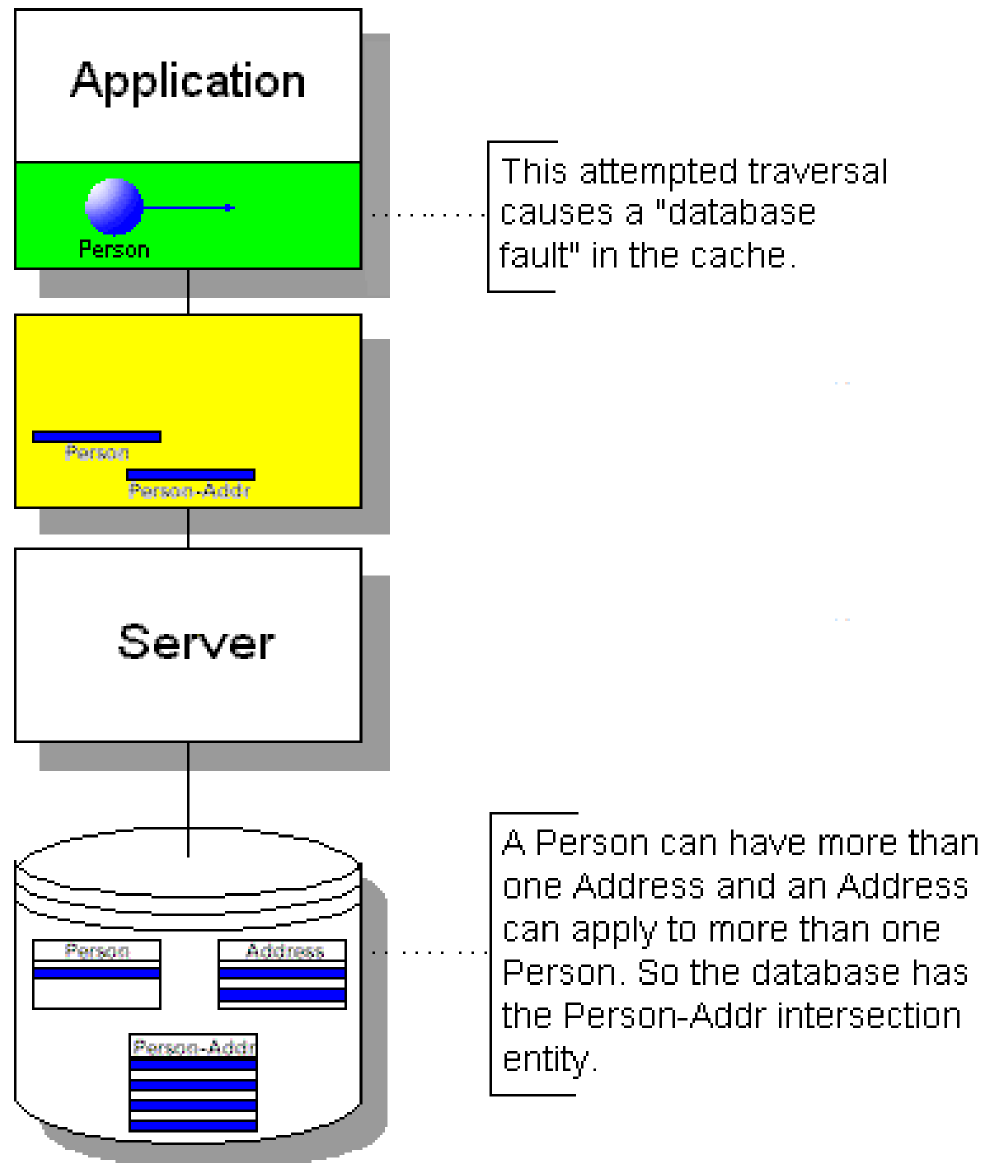
JPA – jak działa



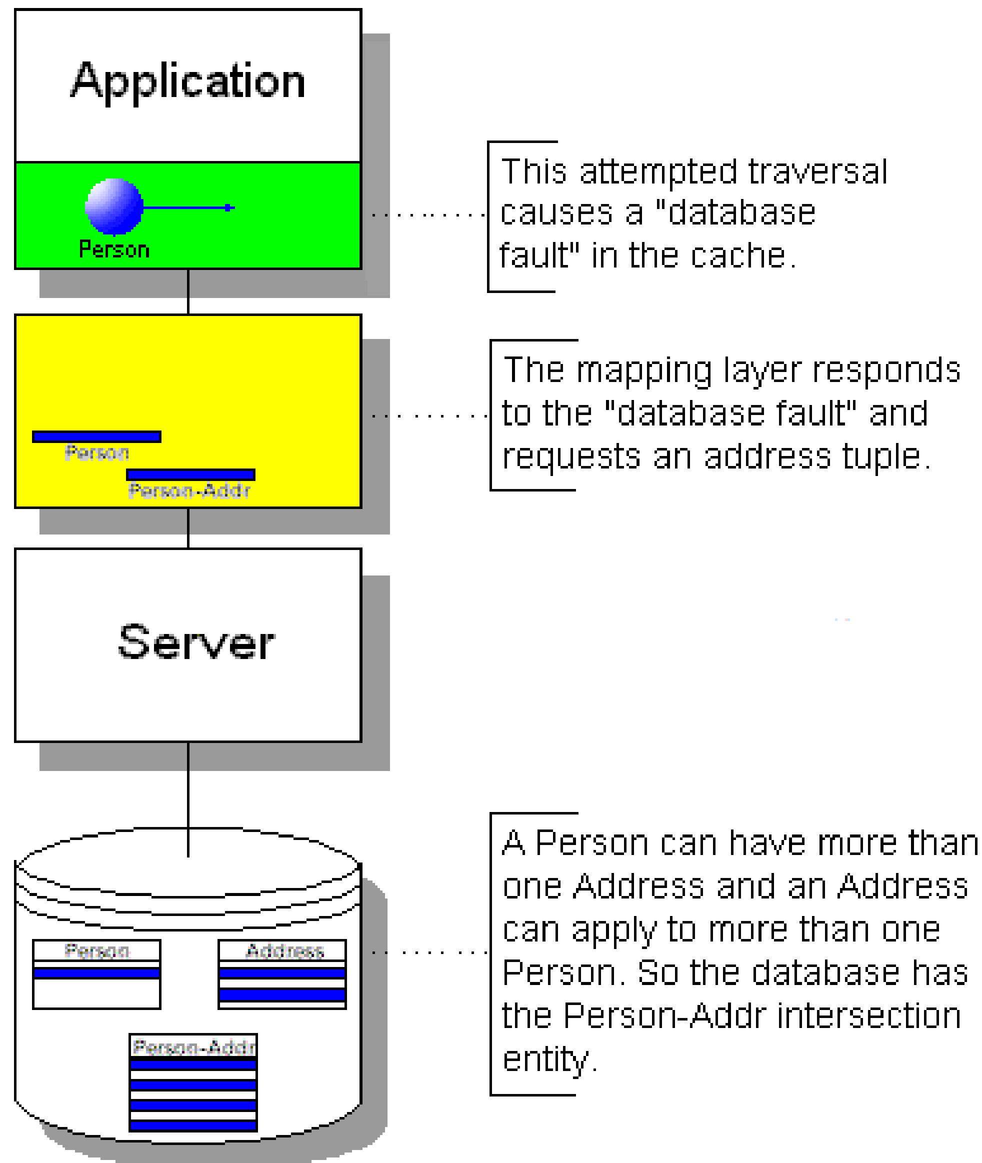
JPA – jak działa



JPA – jak działa



JPA – jak działa

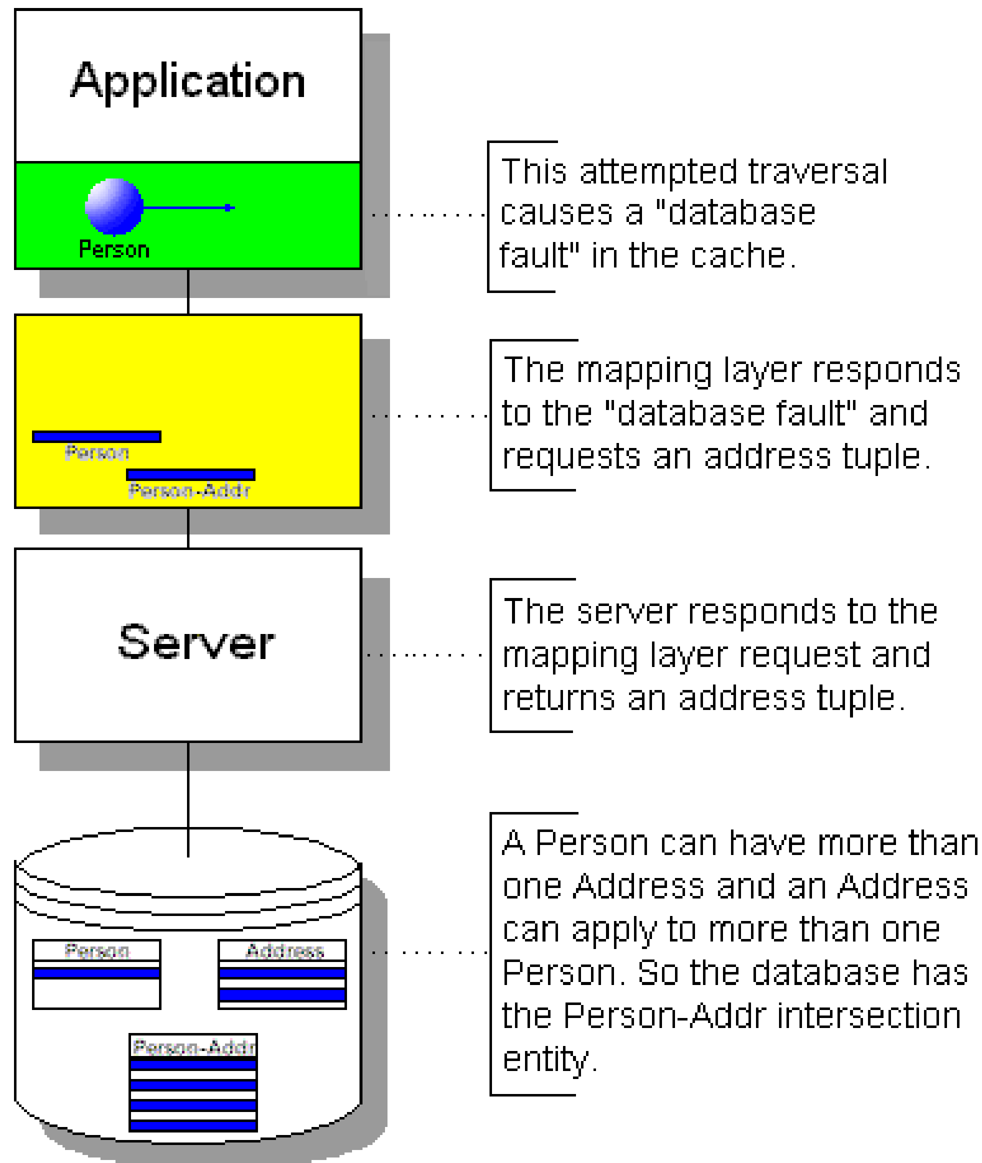


This attempted traversal causes a "database fault" in the cache.

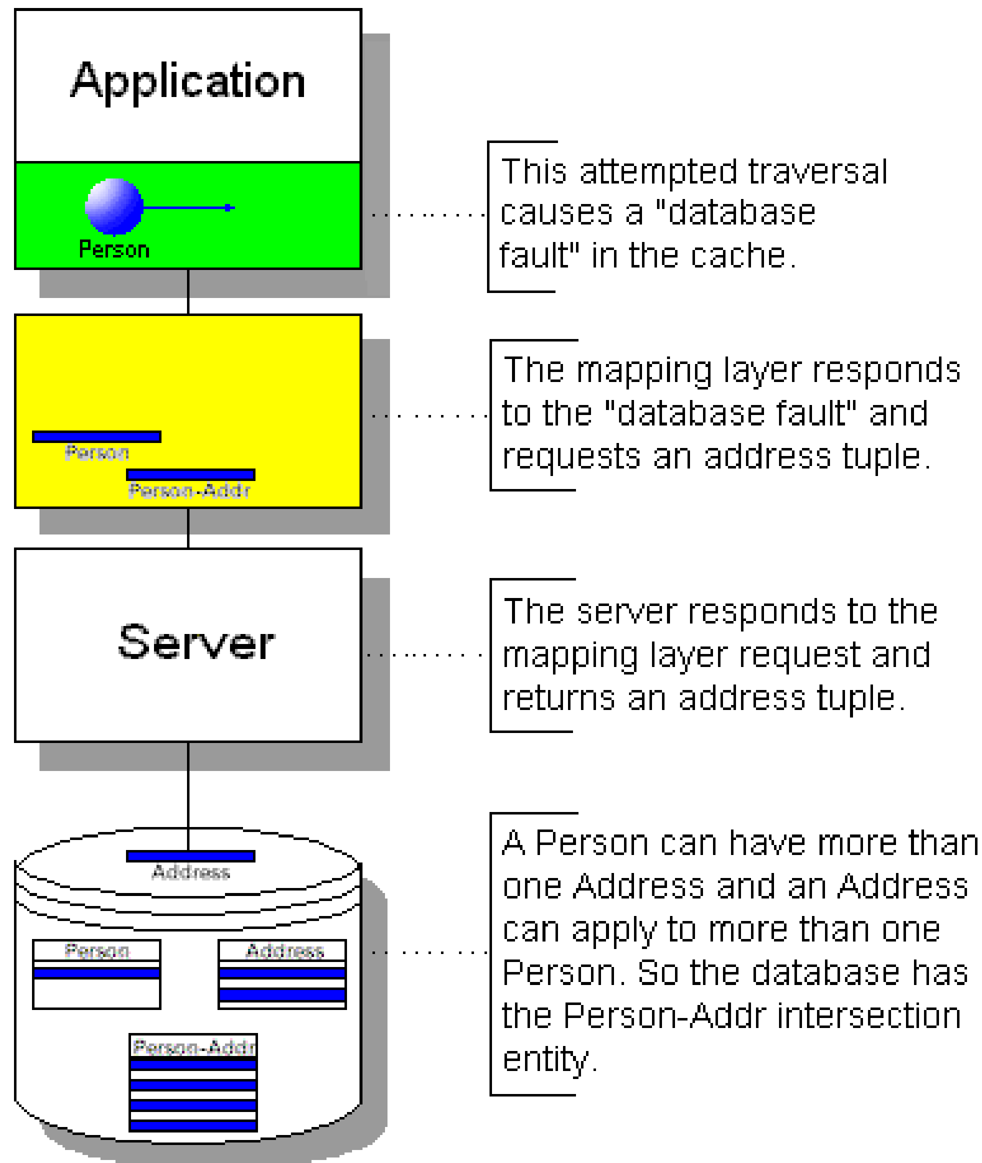
The mapping layer responds to the "database fault" and requests an address tuple.

A Person can have more than one Address and an Address can apply to more than one Person. So the database has the Person-Addr intersection entity.

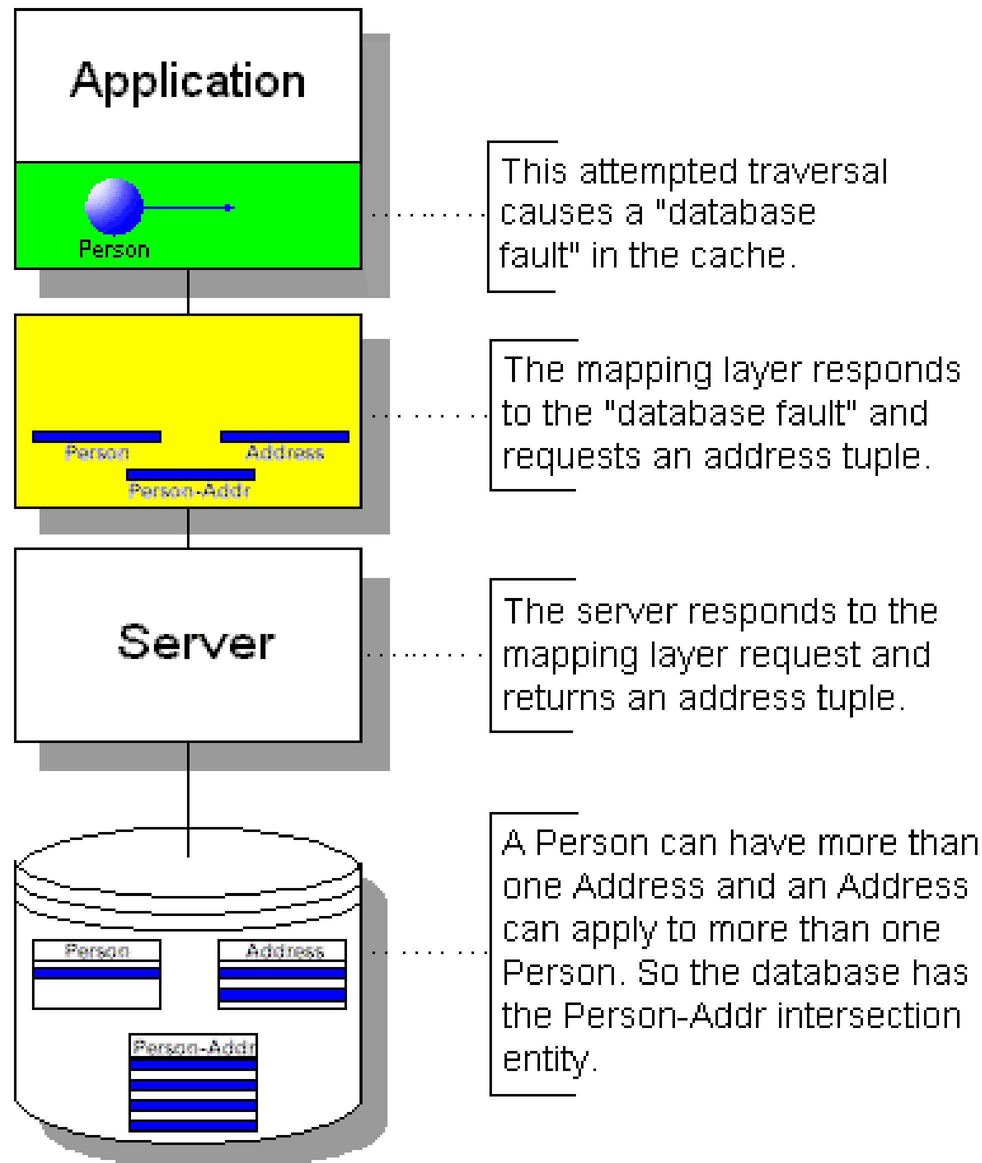
JPA – jak działa



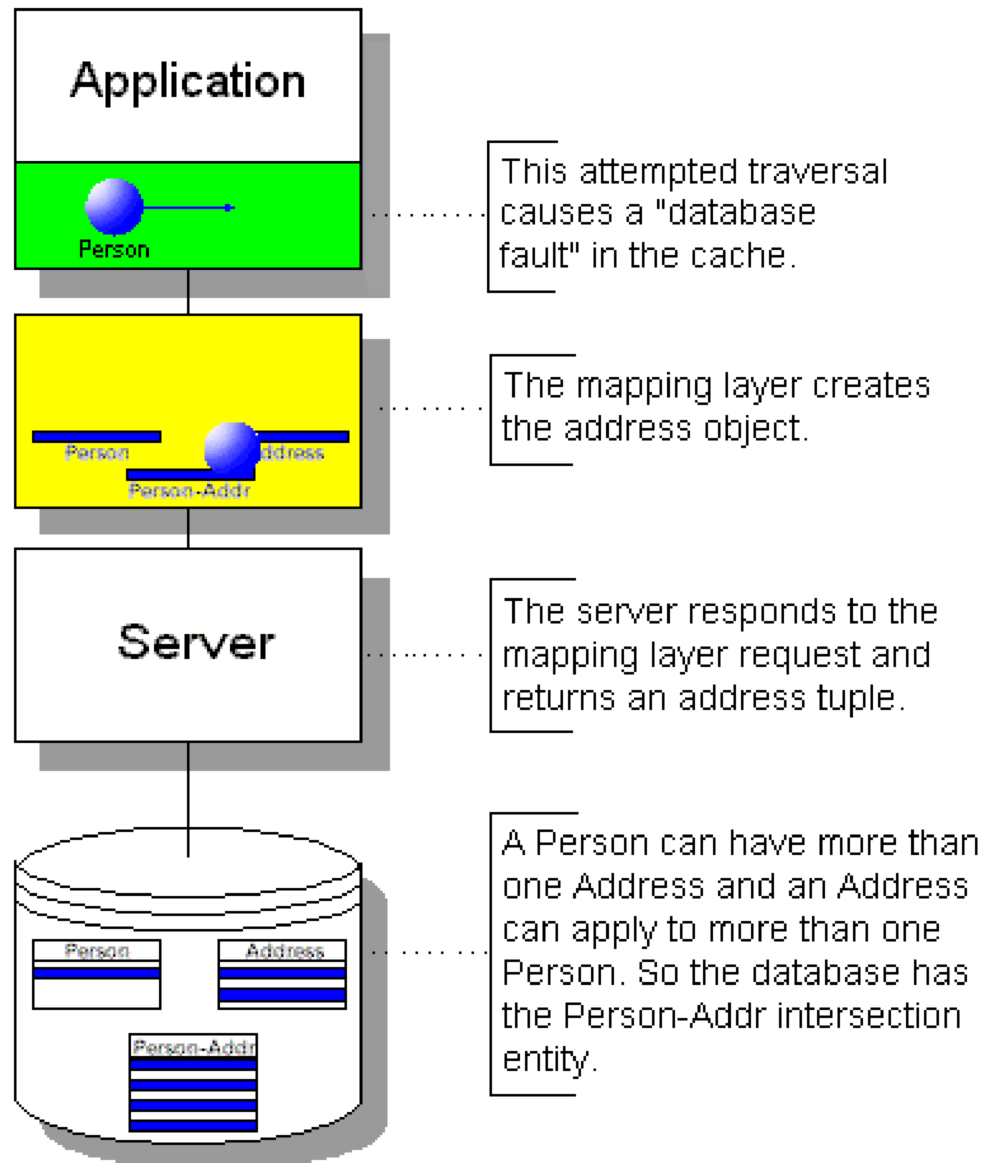
JPA – jak działa



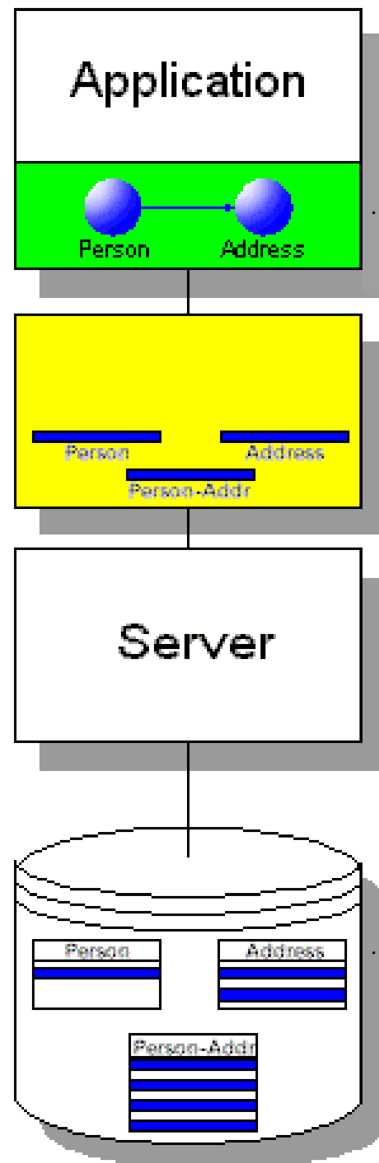
JPA – jak działa



JPA – jak działa



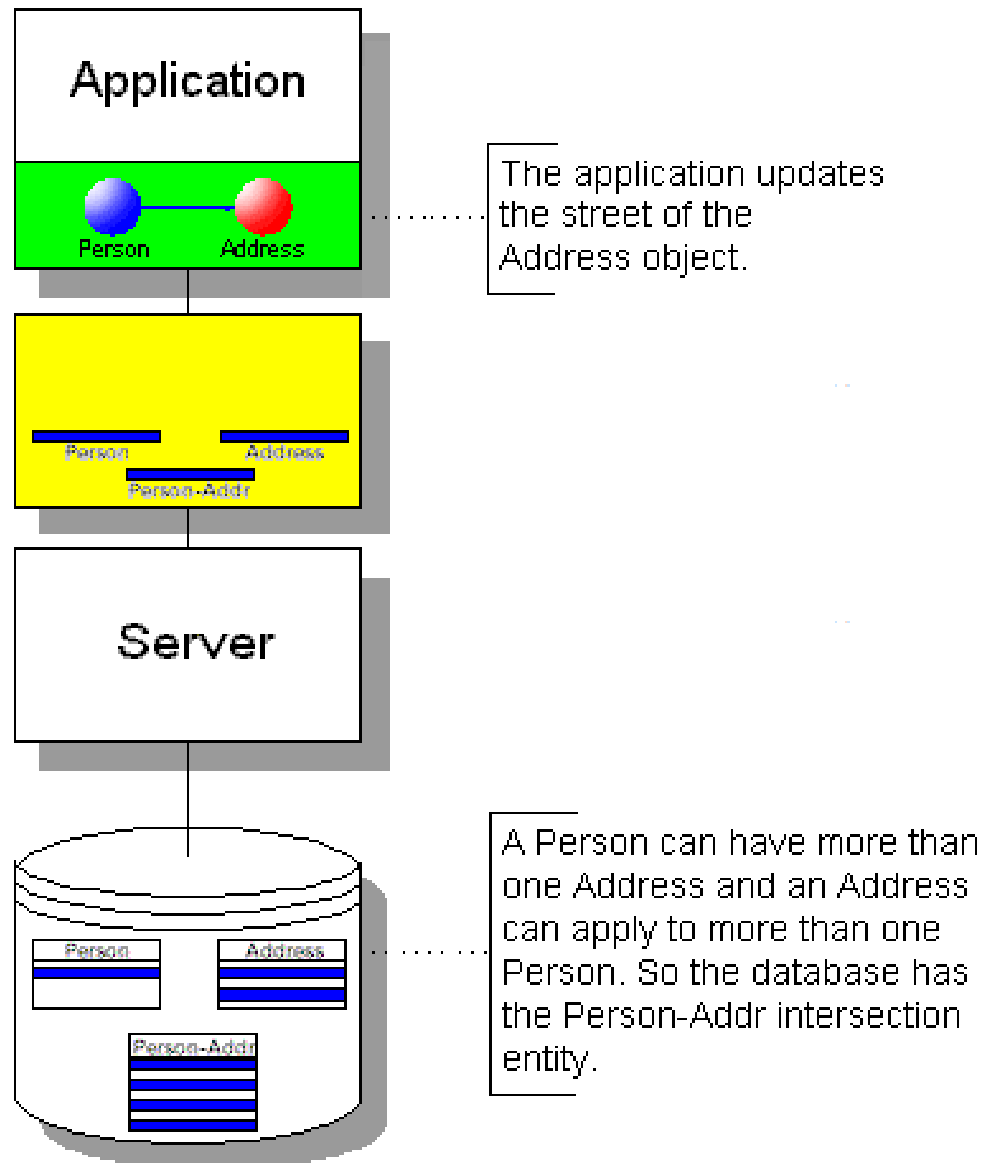
JPA – jak działa



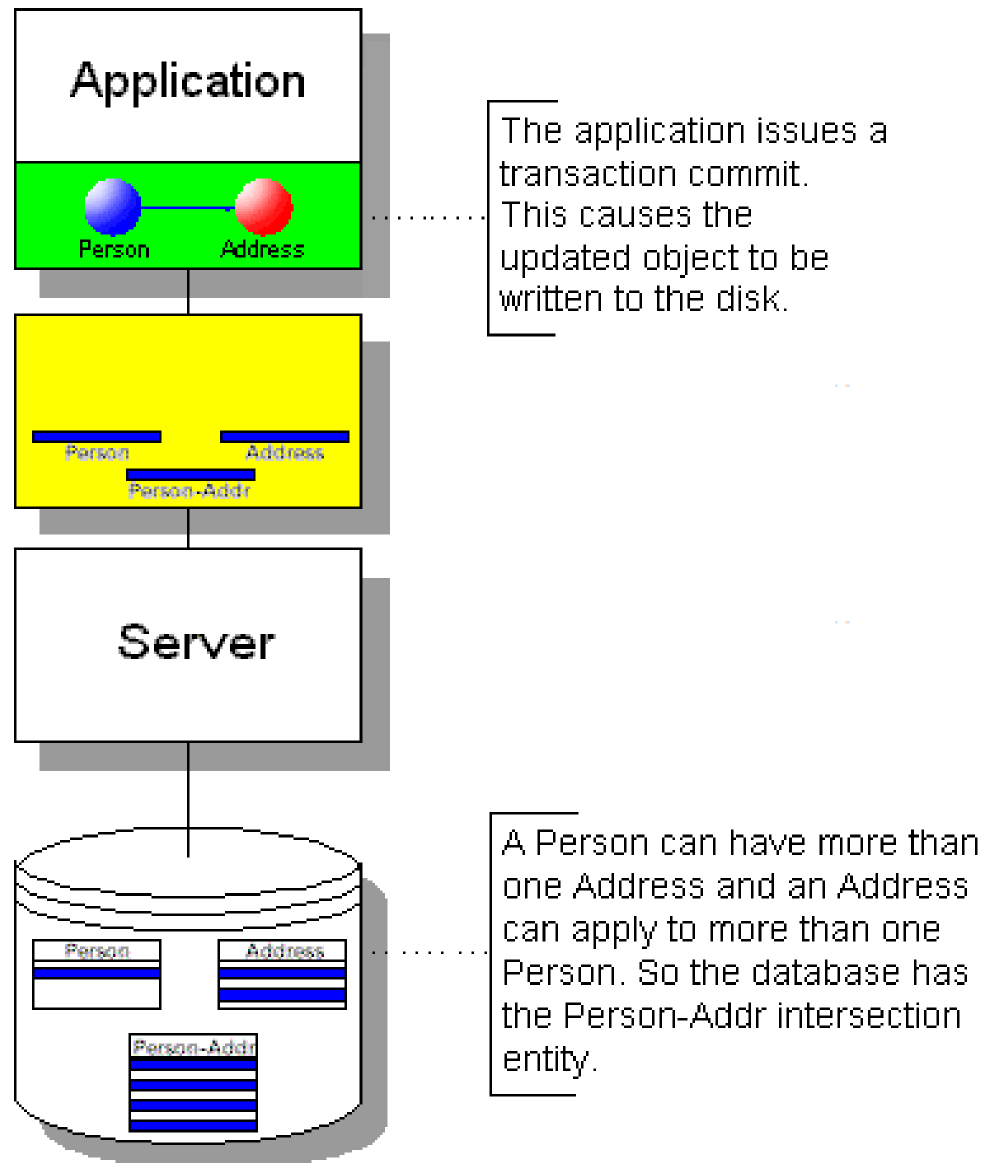
The application workspace now has the person object referencing the address object. The objects may be directly manipulated by the object programming language.

A Person can have more than one Address and an Address can apply to more than one Person. So the database has the Person-Addr intersection entity.

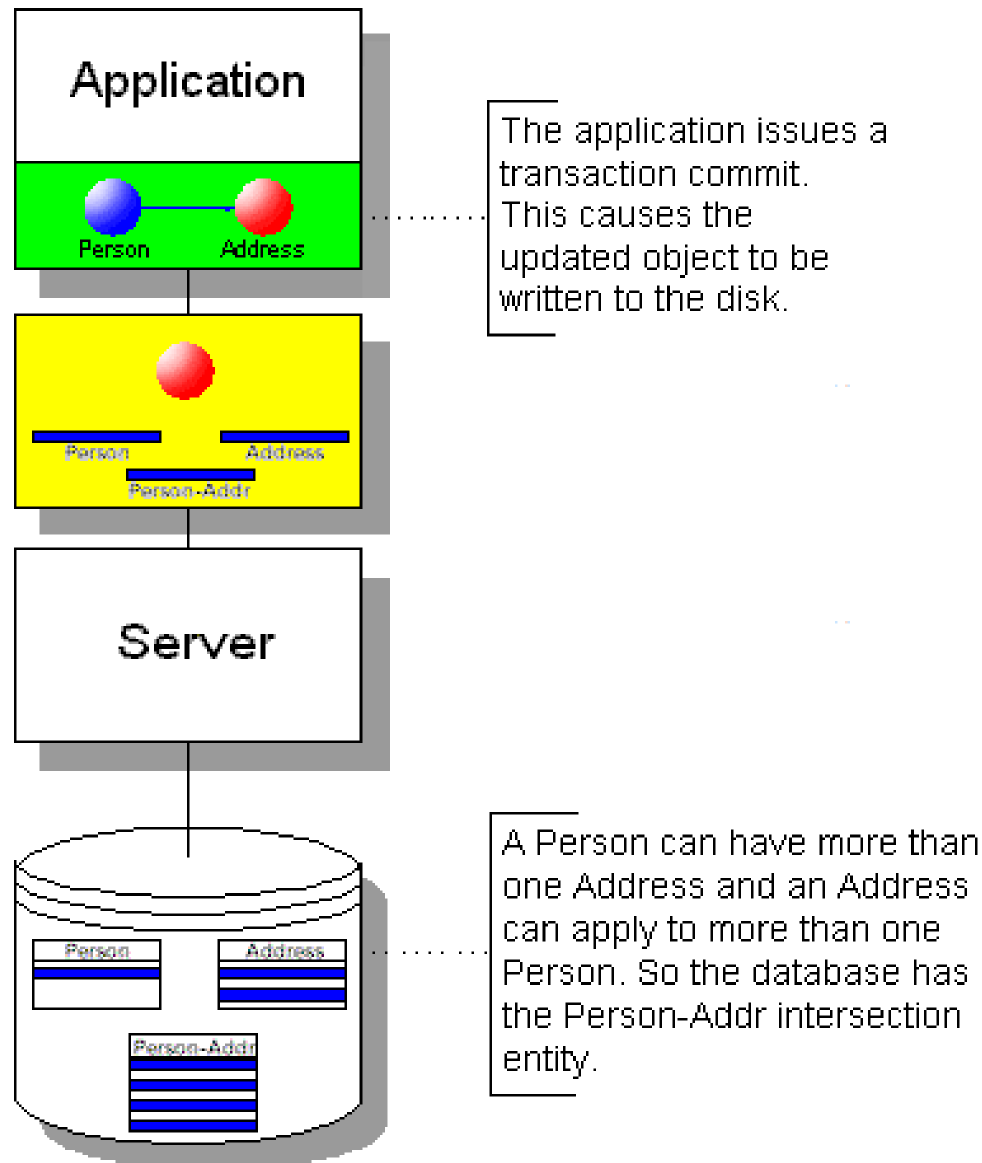
JPA – jak działa



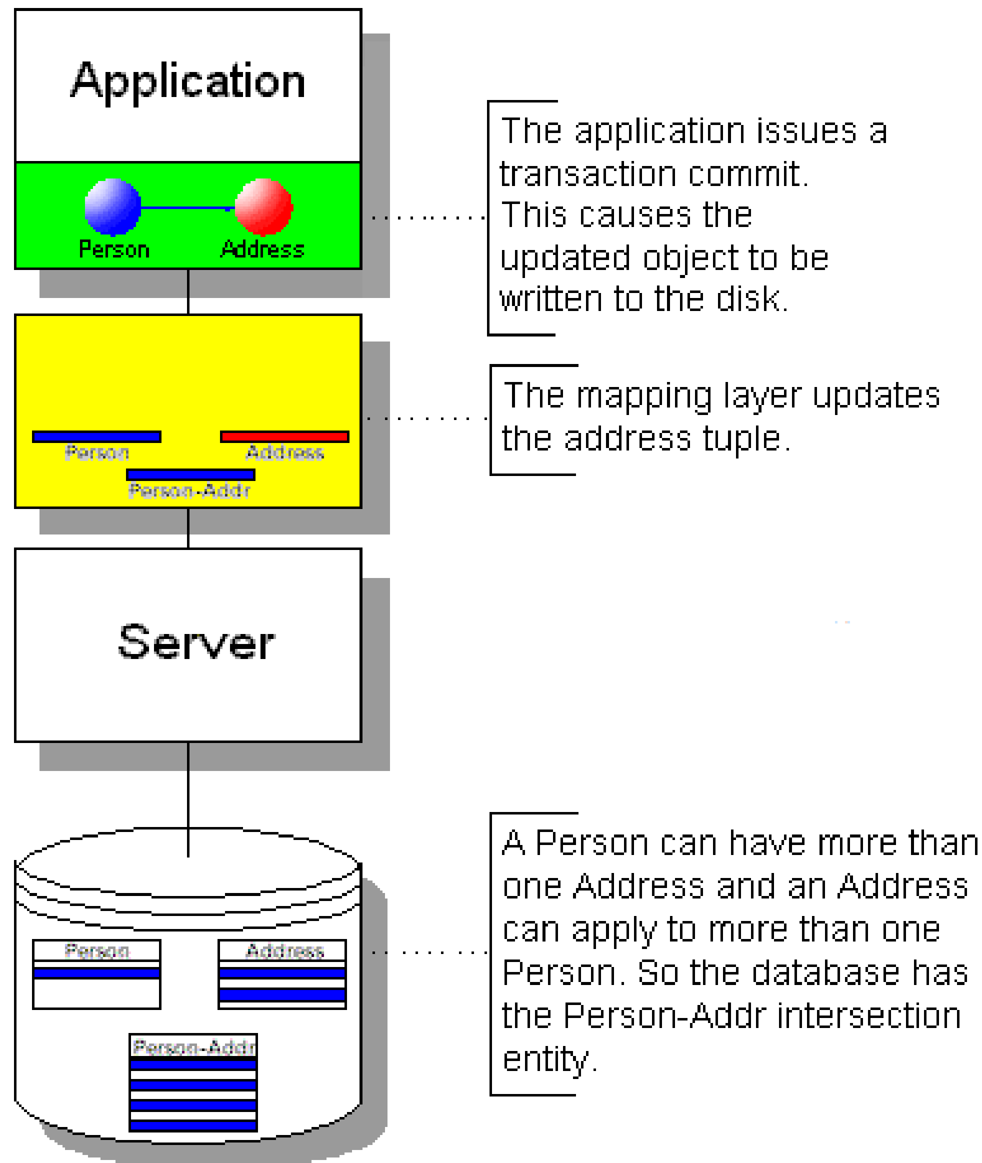
JPA – jak działa



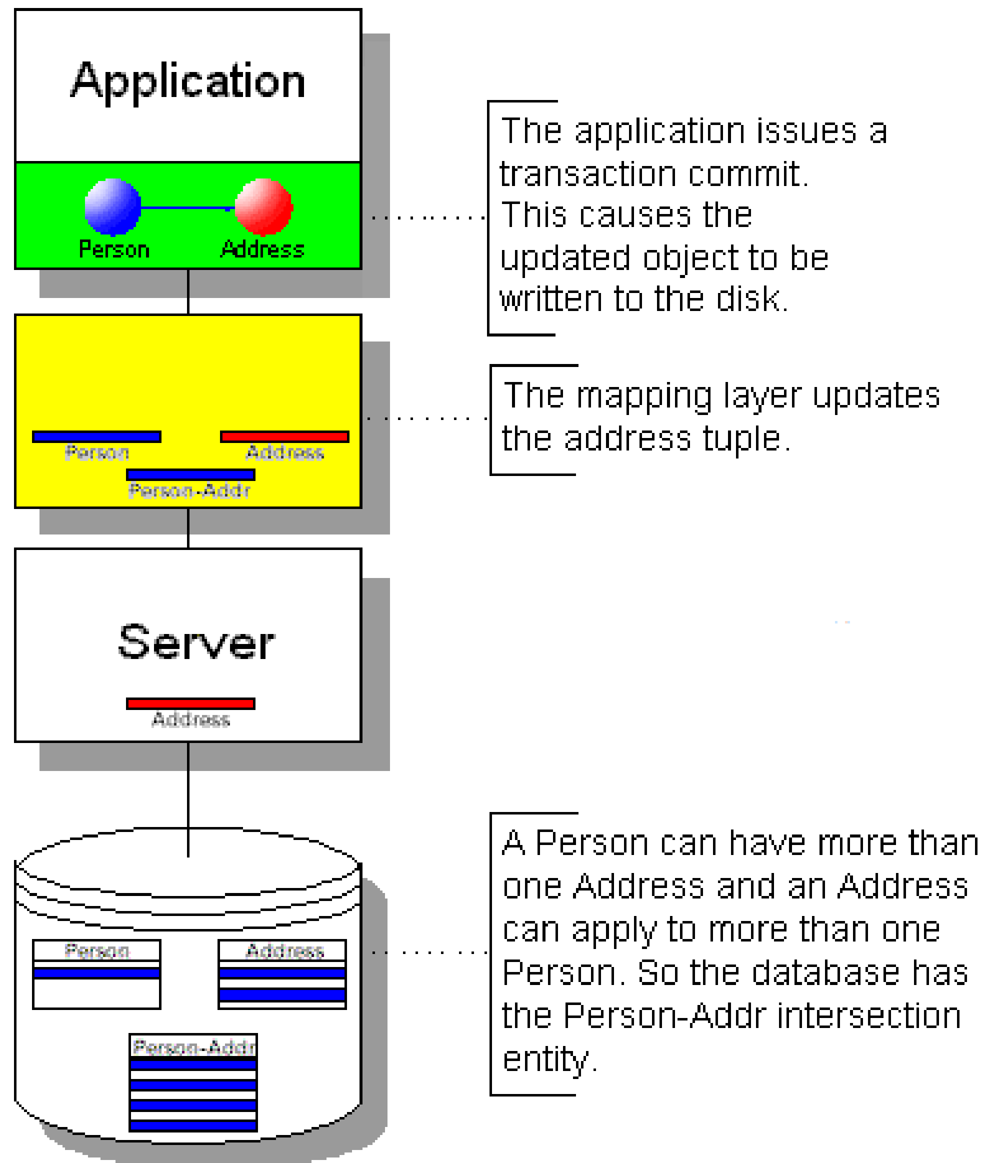
JPA – jak działa



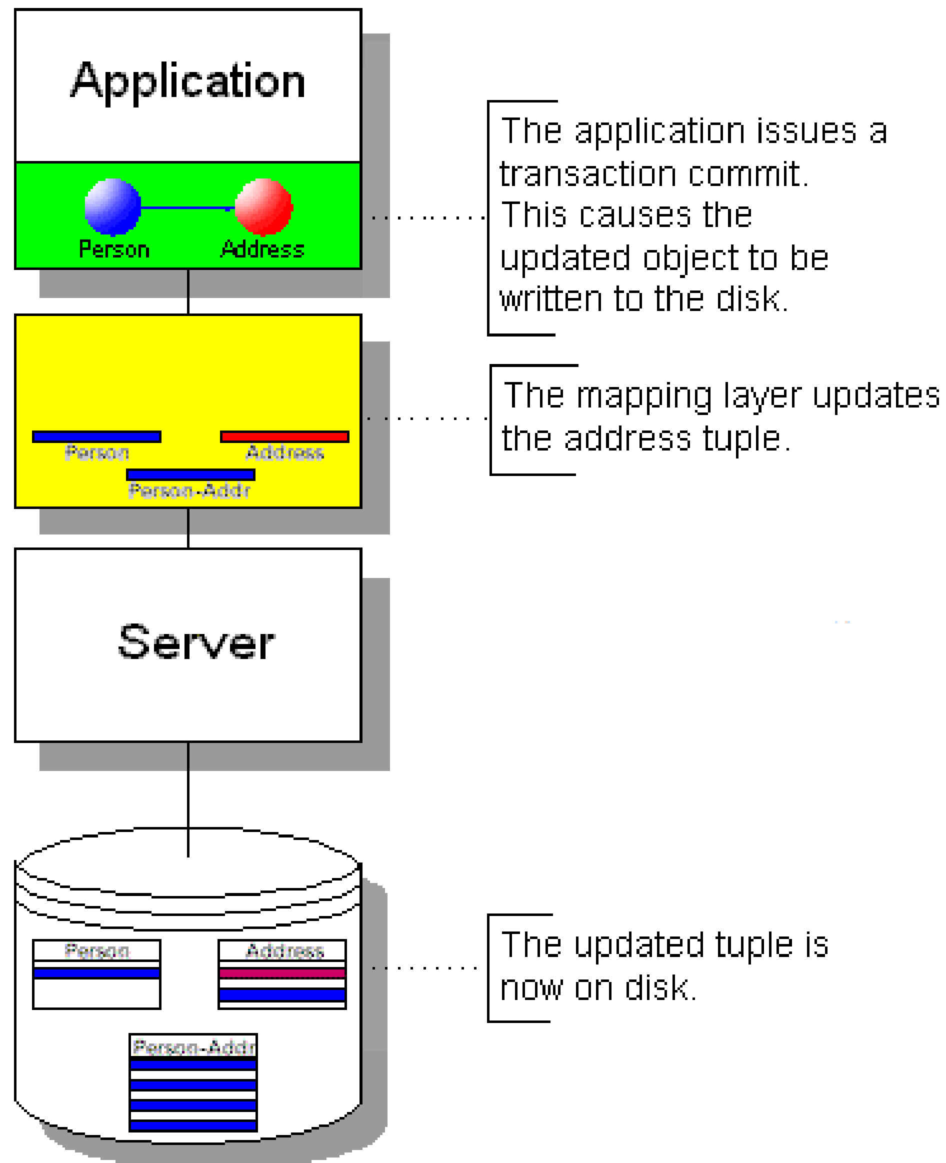
JPA – jak działa



JPA – jak działa



JPA – jak działa

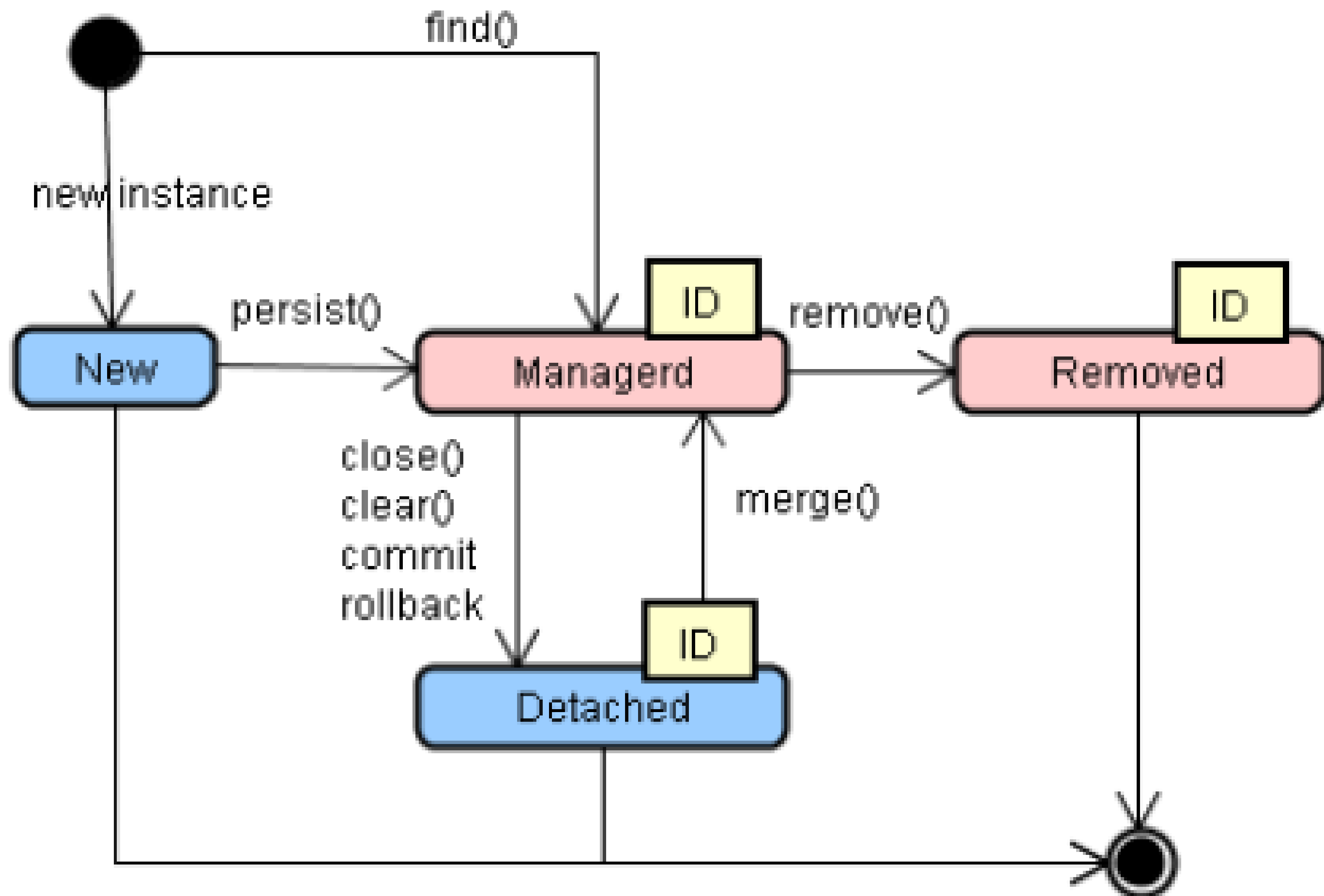


JPA – PersistenceManager

PersistenceManager – zarządca encji

- Automatycznie wstrzykiwany przez serwer
- Programista nie musi się przejmować cache'owaniem, odczytem z bazy, updatem, deletem, zapisem ani transakcjami
- Programista może budować zapytania w Java Persistence QL, tłumaczone przez PM na SQL

JPA – cykl życia encji



JPA - związki

- Związki między obiektami:
  - 1-1
  - 1-*
  - *-1
  - *-*
- Każdy rodzaj związku może być jedno- lub dwukierunkowy
- Związek ma stronę „właściciela” (*owning*) i „odwrotną” (*reverse*)

JPA - związki

Typowe właściwości związków (parametry adnotacji):

- cascade – jakie operacje wykonywać kaskadowo na związku (wszystkie, zapis do DB, usuwanie, update, odświeżanie)
- fetch – LAZY|EAGER
- mappedBy – po stronie „inverse” określenie, który atrybut strony „owning” wskazuje na dany obiekt. Związki są zawsze mapowane po stronie właściciela.
- optional – czy referencja może być nullem

JPA - związki

@OneToOne

```
@Entity public class Customer { //owning
    ...
    @OneToOne(optional=false)
    public CustomerRecord getCustomerRecord(){
        return customerRecord;
    }
    ...
}

@Entity
public class CustomerRecord { //owned - inverse
  ...
  @OneToOne(optional=false, mappedBy="customerRecord")
  public Customer getCustomer() {...}
  ...
}
```

JPA - związki

@OneToMany, @ManyToOne

```
@Entity public class Customer {
    ...
    @OneToMany(cascade=ALL, mappedBy="customer")
    public Set<Order> getOrders() {
        return orders;
    }
    ...
}

@Entity public class Customer {
    ...
    @ManyToOne
    public Customer getCustomer() {
        return customer;
    }
    ...
}
```

JPA - związki

@ManyToMany

```
@Entity public class Customer {
    ...
    @ManyToMany
    public Set<PhoneNumber> getPhones() {
        return phones;
    }
    ...
}

@Entity public class PhoneNumber {
    ...
    @ManyToMany(mappedBy="phones")
    public Set<Customer> getCustomers() {
        return customers;
    }
    ...
}
```

Java Persistence Query Language

Obiektowy język zapytań przekształcany na SQL

Przykłady:

- Wybranie wszystkich graczy
  - `SELECT p FROM Player p`
  - `FROM Player AS p`
    - Player – nazwa encji
    - p – nazwa zmiennej – elementu wyniku
- Anonimowa parametryzacja
  - `SELECT DISTINCT p FROM Player p WHERE p.position = ?1`
    - ?1 – pierwszy parametr

Java Persistence Query Language

- Parametry nazwane
  - `SELECT DISTINCT p FROM Player p WHERE p.position = :position AND p.name = :name`
    - :position, :name – parametry, pod które trzeba programistycznie podstawić wartości
- Gracze, którzy mają drużynę
  - `SELECT DISTINCT p FROM Player p, IN(p.teams) t`
  - `SELECT DISTINCT p FROM Player p JOIN p.teams t`
  - `SELECT DISTINCT p FROM Player p WHERE p.team IS NOT EMPTY`

Java Persistence Query Language

- Nawigacja w kolekcjach
  - `SELECT DISTINCT p FROM Player p, IN (p.teams) AS t WHERE t.city = :city`
    - Nie: `WHERE p.teams.city = :city`
- Nawigacja w pojedynczych relacjach
  - `SELECT DISTINCT p FROM Player p, IN (p.teams) t WHERE t.league.sport = :sport`
- Inne operatory w WHERE
  - `LIKE, IS (NOT) NULL, IS EMPTY, BETWEEN, MEMBER OF`
  - wiele innych znanych z SQL