Java Data Objects (JDO)

Part 3
Entity Relationships in JDO

- **Owned**
  - one of the objects cannot exist without the other
- **unowned,**
  - both objects can exist independently of their relationship with one another.
- The App Engine support
  - owned one-to-one relationships
  - owned one-to-many relationships,
  - both unidirectional and bidirectional.
- Unowned relationships
  - you can manage these relationships
Entity Relationships in JDO

• Owned One-to-One Relationships
• Owned One-to-Many Relationships
• Unowned Relationships
• Relationships, Entity Groups, and Transactions
• Dependent Children and Cascading Deletes
• Polymorphic Relationships
import com.google.appengine.api.datastore.Key;
// ... imports ...

@PersistenceCapable
public class ContactInfo {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private String streetAddress;

    // ...
}  

Employee.java

import ContactInfo;
// ... imports ...

@PersistenceCapable
public class Employee {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private ContactInfo contactInfo;

    ContactInfo getContactInfo() {
        return contactInfo;
    }
    void setContactInfo(ContactInfo contactInfo) {
        this.contactInfo = contactInfo;
    }

    // ...
}
Owned One-to-One Relationships

• The child's key uses the parent's key as its entity group parent.
  – Child == ContactInfo
  – Parent == Employee

• When the app accesses the child object using the parent object's field, the JDO implementation performs an entity group parent query to get the child.

• The child class must have a key field
  – whose type can contain the parent key information
  – either a Key, or a Key value encoded as a string
ContactInfo.java

import Employee;

// ...
  @Persistent(mappedBy = "contactInfo")
  private Employee employee;

Child objects are loaded from the datastore when they are accessed for the first time. If you do not access the child object on a parent object, the entity for the child object is never loaded. If you want to load the child, you can either "touch" it before closing the PersistenceManager (e.g. by calling getContactInfo() in the above example) or explicitly add the child field to the default fetch group so it’s retrieved and loaded with the parent:

Employee.java

import ContactInfo;

// ...
  @Persistent(defaultFetchGroup = "true")
  private ContactInfo contactInfo;
Owned One-to-Many Relationships

• To create a one-to-many relationship between objects of one class and multiple objects of another, use a Collection of the related class.

```java
import java.util.List;

// ...  
@Persistent
private List<ContactInfo> contactInfoSets;
```
A one-to-many bidirectional relationship is similar to a one-to-one, with a field on the parent class using the annotation `@Persistent(mappedBy = "...")`, where the value is the name of the field on the child class:

**Employee.java**

```java
import java.util.List;

// ...
@Persistent(mappedBy = "employee")
private List<ContactInfo> contactInfoSets;
```

**ContactInfo.java**

```java
import Employee;

// ...
@Persistent
private Employee employee;
```
How Ordered Collections Maintain Their Order

• Ordered collections, such as List<...>, preserve the order of objects when the parent object is saved.
  – JDO requires that databases preserve this order by storing the position of each object as a property of the object.
  – App Engine stores this as a property of the corresponding entity, using a property name equal to the name of the parent's field followed by _INTEGER_IDX.
  – **Position properties are inefficient.**
• If an element is added, removed or moved in the collection, all entities subsequent to the modified place in the collection must be updated.
  – This can be slow, and error prone if not performed in a transaction.
Unowned One-to-Many Relationships

• Now suppose we want to let a person have multiple favorite foods.
• Again, a favorite food does not belong to the person because it can be the favorite food of any number of people.

```java
Person.java

// ... imports ...

@PersistenceCapable
public class Person {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private Set<Key> favoriteFoods;

    // ...
}
```
Notes

• rather than giving Person a member of type Set<Food> to represent the person's favorite foods,
  – We give Person a member of typeSet<Key>,
  – where the set contains the unique identifiers of Food objects.

• Note that, unless an instance of Person and an instance of Food contained in Person.favoriteFoods are in the same entity group,
  – it is not possible to update the person and that favorite food in a single transaction.
Many-to-Many Relationships

• We can model a many-to-many relationship by maintaining collections of keys on both sides of the relationship.

• Let's adjust our example to let Food keep track of the people that consider it a favorite:
Person.java

```java
import java.util.Set;
import com.google.appengine.api.datastore.Key;

// ...
@Persistent
private Set<Key> favoriteFoods;
```

Food.java

```java
import java.util.Set;
import com.google.appengine.api.datastore.Key;

// ...
@Persistent
private Set<Key> foodFans;
```

In this example, the Person maintains a Set of Key values that uniquely identify the Food objects that are favorites, and the Food maintains a Set of Key values that uniquely identify the Person objects that consider it a favorite.
When modeling a many-to-many using **Key** values, be aware that it is the app's responsibility to maintain both sides of the relationship:

**Album.java**

```java
// ...
public void addFavoriteFood(Food food) {
    favoriteFoods.add(food.getKey());
    food.getFoodFans().add(getKey());
}

public void removeFavoriteFood(Food food) {
    favoriteFoods.remove(food.getKey());
    food.getFoodFans().remove(getKey());
}
```
Relationships, Entity Groups, and Transactions

• When your application saves an object with owned relationships to the datastore,
  – all other objects that can be reached via relationships and need to be saved (they are new or have been modified since they were last loaded) are saved automatically.

• This has important implications for transactions and entity groups.
Consider the following example using a unidirectional relationship between the `Employee` and `ContactInfo` classes above:

```java
Employee e = new Employee();
ContactInfo ci = new ContactInfo();
e.setContactInfo(ci);

pm.makePersistent(e);
```

When the new `Employee` object is saved using the `pm.makePersistent()` method, the new related `ContactInfo` object is saved automatically. Since both objects are new, App Engine creates two new entities in the same entity group, using the `Employee` entity as the parent of the `ContactInfo` entity. Similarly, if the `Employee` object has already been saved and the related `ContactInfo` object is new, App Engine creates the `ContactInfo` entity using the existing `Employee` entity as the parent.
Notice, however, that the call to `pm.makePersistent()` in this example does not use a transaction. Without an explicit transaction, both entities are created using separate atomic actions. In this case, it is possible for the creation of the Employee entity to succeed, but the creation of the ContactInfo entity to fail. To ensure that either both entities are created successfully or neither entity is created, you must use a transaction:

```java
Employee e = new Employee();
ContactInfo ci = new ContactInfo();
e.setContactInfo(ci);

try {
    Transaction tx = pm.currentTransaction();
    tx.begin();
    pm.makePersistent(e);
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
}
```
Dependent Children and Cascading Deletes

• An owned relationship can be "dependent,“
  – the child cannot exist without its parent.
  – a parent object is deleted, all child objects are also deleted.

• Breaking an owned, dependent relationship by assigning a new value to the dependent field on the parent
  – also deletes the old child.

• You can declare an owned one-to-one relationship to be dependent
  – by adding dependent="true" to the Persistent annotation of the field on the parent object that refers to the child:
You can declare an owned one-to-many relationship to be dependent by adding an `@Element(dependent = "true")` annotation to the field on the parent object that refers to the child collection:

```java
import javax.jdo.annotations.Element;

// ...
@Persistent
@Element(dependent = "true")
private List contactInfos;
```

As with creating and updating objects, if you need every delete in a cascading delete to occur in a single atomic action, you must perform the delete in a transaction.
Polymorphic Relationships

• Not supported
• Alternative
  – store a Key reference.
  – For example, if you have a Recipe base class with Appetizer, Entree, and Dessert specializations, and you want to model the favoriteRecipe of a Chef,
Recipe.java

```java
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.Inheritance;
import javax.jdo.annotations.InheritanceStrategy;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

@PersistenceCapable
@Inheritance(strategy = InheritanceStrategy.SUBCLASS_TABLE)
public abstract class Recipe {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private int prepTime;
}
```

Appetizer.java

```java
// ... imports ...

@PersistenceCapable
public class Appetizer extends Recipe {
    // ... appetizer-specific fields
}
```
Entree.java

```java
// ... imports ...

@PersistenceCapable
public class Entree extends Recipe {
// ... entree-specific fields
}
```

Dessert.java

```java
// ... imports ...

@PersistenceCapable
public class Dessert extends Recipe {
// ... dessert-specific fields
}
```

Chef.java

```java
// ... imports ...

@PersistenceCapable
public class Chef {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent(dependent = "true")
    private Recipe favoriteRecipe;
}
```
Unfortunately, if you instantiate an `Entree` and assign it to `Chef.favoriteRecipe` you will get an `UnsupportedOperationException` when you try to persist the `Chef` object. This is because the runtime type of the object, `Entree`, does not match the declared type of relationship field, `Recipe`. The workaround is to change the type of `Chef.favoriteRecipe` from a `Recipe` to a `Key`:

```
Chef.java

// ... imports ...

@PersistenceCapable
public class Chef {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private Key favoriteRecipe;
}
```

Since `Chef.favoriteRecipe` is no longer a relationship field, it can refer to an object of any type. The downside is that, as with an unowned relationship, you need to manage this relationship manually.