Java Data Objects (JDO)

Part 2
Creating, Getting and Deleting Data in JDO

- Making Objects Persistent
- Keys
- Getting an Object By Key
- Updating an Object
- Deleting an Object
Making Objects Persistent

```java
PersistenceManager pm = PMF.get().getPersistenceManager();

Employee e = new Employee("Alfred", "Smith", new Date());
try {
    pm.makePersistent(e);
} finally {
    pm.close();
}
```

The call to `makePersistent()` is synchronous, and doesn't return until the object is saved and indexes are updated.
Keys

• Every entity has a key that is unique over all entities in App Engine.
• A complete key includes several pieces of information, including
  – the application ID,
  – the kind, and
  – an entity ID.
  – (Keys also contain information about entity groups; see Transactions for more information.)
• An object's key is stored in a field on the instance.
  – You identify the primary key field using the @PrimaryKey annotation
Keys

• Key generation
  – The app can provide as a string when the object is created, or
  – Automatically (by datastore)

• The complete key must be unique across all entities in the datastore.
  – In other words, an object must have an ID that is unique across all objects of the same kind and with the same entity group parent (if any).
  – You select the desired behavior of the key using the type of the field and annotations.
Key types - long

A `long` integer (`java.lang.Long`), an entity ID automatically generated by the datastore. Use this for objects without entity group parents whose IDs should be generated automatically by the datastore. The `long` key field of an instance is populated when the instance is saved.

```java
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

// ...
@PrimaryKey
@Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY) ENTITY
private Long id;
```
Key types - Unencoded String

A **String** (`java.lang.String`), an entity ID ("key name") provided by the application when the object is created. Use this for objects without entity group parents whose IDs should be provided by the application. The application sets this field to the desired ID prior to saving.

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

// ...

@PrimaryKey
private String name;
```
Key types - key

A Key instance (com.google.appengine.api.datastore.Key). The key value includes the key of the entity group parent (if any) and either the app-assigned string ID or the system-generated numeric ID. To create the object with an app-assigned string ID, you create the Key value with the ID and set the field to the value. To create the object with a system-assigned numeric ID, you leave the key field null. (For information on how to use entity group parents, see Transactions.)

```java
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;
import com.google.appengine.api.datastore.Key;

// ...
@PrimaryKey
@Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
ENTITY
private Key key;

public void setKey(Key key) {
    this.key = key;
}
```
The app can create a Key instance using the KeyFactory class:

```java
import com.google.appengine.api.datastore.Key;
import com.google.appengine.api.datastore.KeyFactory;

// ...
    Key key = KeyFactory.createKey(Employee.class.
    getSimpleName(), "Alfred.Smith@example.com");
    Employee e = new Employee();
    e.setKey(key);
    pm.makePersistent(e);
```
Key as Encoded String

Similar to Key, but the value is the encoded string form of the key. Encoded string keys allow you to write your application in a portable manner and still take advantage of App Engine datastore entity groups.

```java
import javax.jdo.annotations.Extension;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

// ...
@PrimaryKey
@Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
@Entity
@Extension(vendorName="datanucleus", key="gae.encoded-pk", value="true")
private String encodedKey;
```

The app can populate this value prior to saving using a key with a name, or it can leave it null. If the encoded key field is null, the field is populated with a system-generated key when the object is saved.

Key instances can be converted to and from the encoded string representation using the KeyFactory methods `keyToString()` and `stringToKey()`. 
When using encoded key strings, you can provide access to an object's string or numeric ID with an additional field:

```java
@PrimaryKey
@Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
@Extension(vendorName="datanucleus", key="gae.encoded-pk", value="true")
private String encodedKey;

@Persistent
@Extension(vendorName="datanucleus", key="gae.pk-name", value="true")
private String keyName;

// OR:

@Persistent
@Extension(vendorName="datanucleus", key="gae.pk-id", value="true")
private Long keyId;
```
Getting an Object by Key

To retrieve an object given its key, use the PersistenceManager's `getObjectById()` method. The method takes the class for the object, and key:

```java
Key k = KeyFactory.createKey(Employee.class.getSimpleName(), "Alfred.Smith@example.com");
Employee e = pm.getObjectById(Employee.class, k);
```

If the class uses a key field that is an unencoded string ID (String) or numeric ID (Long), `getObjectById()` can take the simple value as the key parameter:

```java
Employee e = pm.getObjectById(Employee.class, "Alfred.Smith@example.com");
```
key argument

• The key argument can be of any of the supported key field types
  – string ID, numeric ID, Key value, encoded key string, and
  – can be of a different type than the key field in the class.
• App Engine must be able to derive the complete key from the class name and the provided value.
• String IDs and numeric IDs are exclusive,
  – so a call using a numeric ID never returns an entity with a string ID.
  – If a Key value or encoded key string is used, the key must refer to an entity whose kind is represented by the class.
public void updateEmployeeTitle(User user, String newTitle) {
    PersistenceManager pm = PMF.get().getPersistenceManager();
    try {
        Employee e = pm.getObjectById(Employee.class, user.getEmail());
        if (titleChangeIsAuthorized(e, newTitle) {
            e.setTitle(newTitle);
        } else {
            throw new UnauthorizedTitleChangeException(e, newTitle);
        }
    } finally {
        pm.close();
    }
}
Since the `Employee` instance was returned by the `PersistenceManager`, the `PersistenceManager` knows about any modifications that are made to Persistent fields on the `Employee` and automatically updates the datastore with these modifications when the `PersistenceManager` is closed. It knows this because the `Employee` instance is "attached" to the `PersistenceManager`.

You can modify an object after the `PersistenceManager` has been closed by declaring the class as "detachable." To do this, add the `detachable` attribute to the `@PersistenceCapable` annotation:

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

@PersistenceCapable(detachable="true")
public class Employee {
    // ...
}
```
Now you can read and write the fields of an Employee object after the PersistenceManager that loaded it has been closed. The following example illustrates how a detached object might be useful:

```java
public Employee getEmployee(User user) {
    PersistenceManager pm = PMF.get().getPersistenceManager();
    Employee employee, detached = null;
    try {
        employee = pm.getObjectById(Employee.class,
            "Alfred.Smith@example.com");

        // If you're using transactions, you can call
        // pm.setDetachAllOnCommit(true) before commit
        // to automatically
        // detach all objects without calls to detachCopy
        // or detachCopyAll.
        detached = pm.detachCopy(employee);
    } finally {
        pm.close();
    }
    return detached;
}

class EmployeeTitleChangeException
    extends Exception
{
    public EmployeeTitleChangeException(Employee e, String oldTitle, String newTitle)
    {
        super("employee " + e + " cannot change title from " + oldTitle + " to " + newTitle);
    }
}
```
Deleting an Object

```
pm.deletePersistent(e);
```

To delete multiple objects in JDO, call the `deletePersistentAll(...)` method with a Collection of objects. This method will use a single low-level batch delete operation that is more efficient than a series of individual `deletePersistent(...)` invocations.
Transactions

• Using Transactions
• What Can Be Done In a Transaction
• Isolation and Consistency
• Uses for Transactions
• Transactional Task Enqueuing
Using Transactions

• A transaction is a datastore operation or a set of datastore operations that either succeed completely, or fail completely.

• If the transaction succeeds, then all of its intended effects are applied to the datastore.

• If the transaction fails, then none of the effects are applied.
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService()
Transaction txn = datastore.beginTransaction();
try {
    Key employeeKey = KeyFactory.createKey("Employee", "Joe");
    Entity employee = datastore.get(employeeKey);
    employee.setProperty("vacationDays", 10);
    datastore.put(employee);
    txn.commit();
} finally {
    if (txn.isActive()) {
        txn.rollback();
    }
}