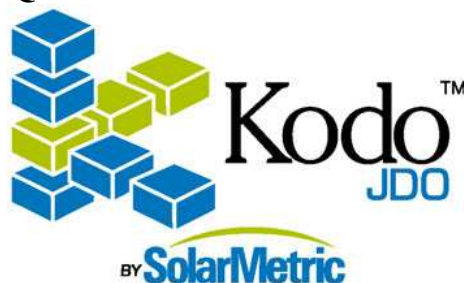


JDO 2.0 API Quick Reference Guide



Query Results

Non unique queries return results as a List of items

```
Query q = pm.newQuery(Car.class);
List<Car> cars = (List<Car>) q.execute();
for (Car car : cars)
```

```
    System.out.println(car.getTopSpeed());
```

Aggregate queries return numbers or arrays of numbers.

```
Query q = pm.newQuery("select avg(price), "
    + "max(price) from com.example.Car");
Object[] stats = (Object[]) q.execute();
System.out.printf("average: %s; max: %s",
    stats[0], stats[1]);
```

Fetch Plans

A fetch plan allows you to specify which fields are to be loaded together. Fetch plans contain fetch groups. Fetch groups are defined in metadata and specify which fields to load when a query is executed or an object is retrieved by ID. For each class there is an implicit default fetch group that includes all primitive, wrapper, String, and Date fields. Fetch plans are also used at detachment time to define the graph of objects to be detached.

Example:

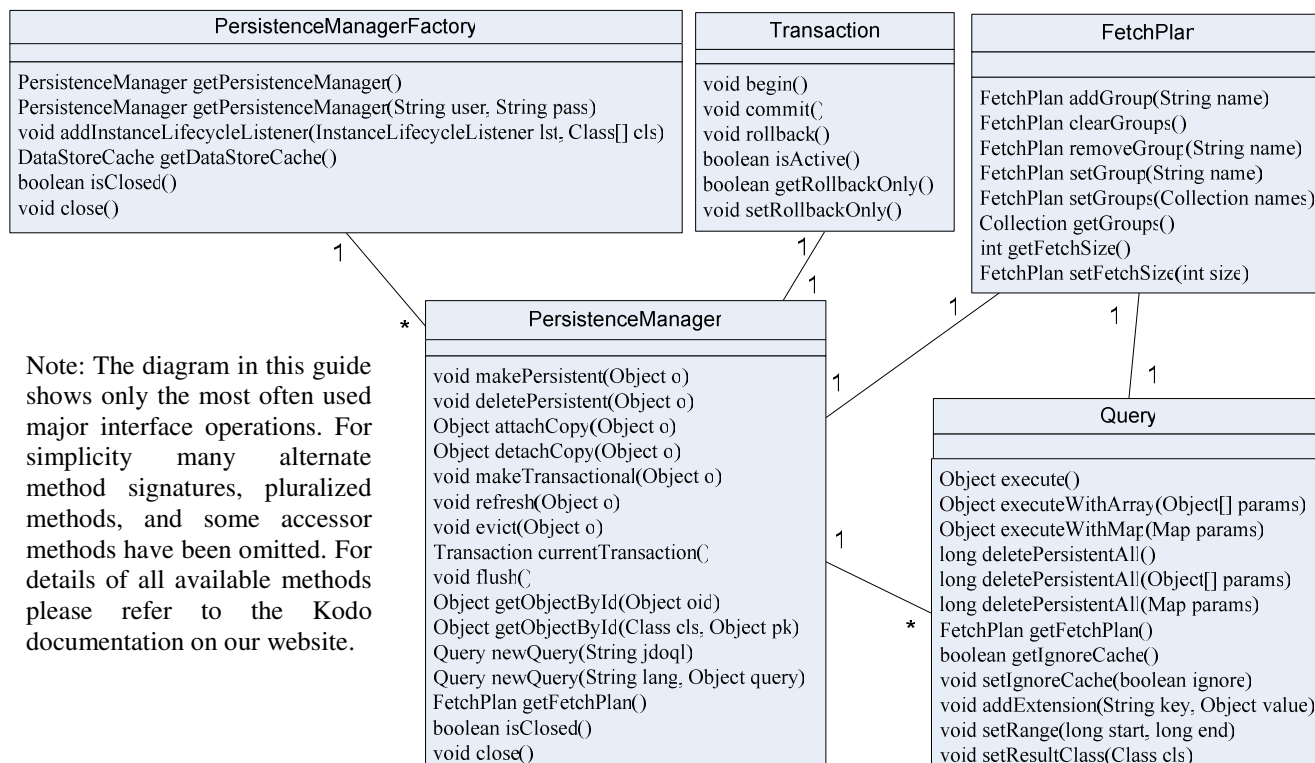
Retrieve all default fields for a Person along with their picture and information about his/her immediate children

```
<fetch-group name="details">
  <fetch-group name="default"/>
  <field name="picture"/>
  <field name="children" fetch-depth="1"/>
</fetch-group>
```

For additional information about JDO 2.0, including details about inheritance mappings, sequence generators, dependent relations, and more, see the Kodo documentation at <http://docs.solarmetric.com>.

JDO Architecture

The following diagram shows the relationships between the major JDO interfaces.



Note: The diagram in this guide shows only the most often used major interface operations. For simplicity many alternate method signatures, pluralized methods, and some accessor methods have been omitted. For details of all available methods please refer to the Kodo documentation on our website.

Bootstrapping

The JDOHelper class is used to get the initial PersistenceManagerFactory

Sample properties:

```
javax.jdo.PersistenceManagerFactoryClass: kodo.jdo.PersistenceManagerFactoryImpl
javax.jdo.option.ConnectionDriverName: oracle.jdbc.driver.OracleDriver
javax.jdo.option.ConnectionUserName: scott
javax.jdo.option.ConnectionPassword: tiger
javax.jdo.option.ConnectionURL: jdbc:oracle:thin:@127.0.0.1:1521:orasic
```

Example:

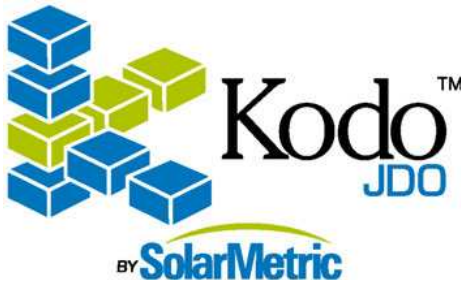
```
import javax.jdo.*;
```

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("kodo.properties");
PersistenceManager pm = pmf.getPersistenceManager();
pm.currentTransaction().begin();
Query q = pm.newQuery("select unique from com.example.Person where firstName == 'John'");
q.getFetchPlan().addGroup("details");
Person p = (Person) q.execute();
p.getPicture().sharpen();
pm.currentTransaction().commit();
```

Note: a File, a Properties instance or an InputStream could have been used in place of "kodo.properties"

JDO 2.0 API

Quick Reference Guide



Listeners and Callbacks

Instance callbacks: in package `javax.jdo`

Interface	Method
LoadCallback	<code>jdoPostLoad</code>
StoreCallback	<code>jdoPreStore</code>
ClearCallback	<code>jdoPreClear</code>
DeleteCallback	<code>jdoPreDelete</code>
DetachCallback	<code>jdoPreDetach</code> , <code>jdoPostDetach</code>
AttachCallback	<code>jdoPreAttach</code> , <code>jdoPostAttach</code>

Example:

```
class Person implements
javax.jdo.listener.DeleteCallback {
    // ...
    void jdoPreDelete() {
        emailHR("deleted" + this.lastName);
    }
}
```

Lifecycle callbacks: in package `javax.jdo.listener`

Interface	Method
CreateLifecycleListener	<code>postCreate</code>
LoadLifecycleListener	<code>postLoad</code>
StoreLifecycleListener	<code>preStore</code> , <code>postStore</code>
ClearLifecycleListener	<code>preClear</code> , <code>postClear</code>
DeleteLifecycleListener	<code>preDelete</code> , <code>postDelete</code>
DirtyLifecycleListener	<code>preDirty</code> , <code>postDirty</code>
DetachLifecycleListener	<code>preDetach</code> , <code>postDetach</code>
AttachLifecycleListener	<code>preAttach</code> , <code>postAttach</code>

Transient Transactional

Kodo supports the optional Transient-transactional state. This feature allows non-persistent objects to have their state rolled back if a transaction fails, and is most useful when the `RestoreValues` option is set to true.

Table Mapping

Classes can be mapped to one or multiple tables. Here, we map to a single table.

```
<class name="Address" table="ADDR">
  <field name="street" column="STREET"/>
  <field name="zip" column="ZIPCODE"/>
  <field name="deliveryInstructions">
    <column name="DELIV_INS" jdbc-type="CLOB"/>
  </field>
</class>
```

Inheritance Mapping

Inheritance strategies are specified relative to a class's superclass. Valid values are `new-table`, `superclass-table`, `no-table`.

```
<class name="Person" table="PEOPLE">
  <inheritance strategy="new-table">
    <discriminator value="P" column="TYPE"/>
  </inheritance>
  ...
</class>
<class name="PartTimeEmployee">
  <inheritance strategy="superclass-table">
    <discriminator value="E"/>
  </inheritance>
  ...
</class>
```

Detaching Objects

Objects may be detached via API calls, by closing a `PersistenceManager`, or by serialization. Detached objects can be changed in another JVM, and attached to any `PersistenceManager`.

```
Person person = (Person) pm.getObjectById(id);
Person detach = (Person) pm.detachCopy(person);
detach.setLastName("Smith");
Person smith = (Person) pm2.attachCopy(detach, true);
```

Configuration Properties

In addition to datasource configuration, the following standard properties may be set when creating a `PersistenceManagerFactory`.

javax.jdo.option.IgnoreCache: If false, queries must include changes in current transaction. Otherwise, changes might not be included.

javax.jdo.option.Multithreaded: Whether persistent instances and JDO components will be accessed simultaneously by multiple threads.

javax.jdo.option.NontransactionalRead: Whether you can read data outside of a transaction.

javax.jdo.option.NontransactionalWrite: Whether you can modify persistent fields outside of a transaction. If true, changes will be applied in the next transaction.

javax.jdo.option.Optimistic: Selects transaction mode.

javax.jdo.option.RestoreValues: Restore field values on rollback.

javax.jdo.option.RetainValues: Retain field values after commit.

Relationships

One-To-One:

Each person has a (possibly null) Address.

```
<class name="Person" table="PEOPLE">
  <field name="address" column="ADDR"/>
  ...
</class>
<class name="Address" table="ADDR">
  <datastore-identity column="AID"/>
  ...
</class>
```

One-to-Many:

A Customer has a collection of Orders.

```
<class name="Customer" table="CUST">
  <datastore-identity column="CID"/>
  <field name="orders">
    <element column="CUST_CID"/>
  </field>
  ...
</class>
<class name="Order" table="ORDERS">
  ...
</class>
```

Many-to-Many:

Customers have Phone numbers and vice versa.

```
<class name="Customer" table="CUST">
  <datastore-identity column="CID"/>
  <field name="phones"
    table="CUST_PHONES">
    <join column="CUST_CID"/>
    <element column="CUST_PID">
  </field>
  ...
</class>
<class name="Phone" table="NUMBERS">
  <datastore-identity column="PID"/>
  <field name="customers"
    mapped-by="phones"/>
  ...
</class>
```

Embedded:

A Person has an embedded SalaryInfo.

```
<class name="Person" table="PEOPLE">
  <field name="salaryInfo">
    <embedded>
      <field name="base" column="BASE"/>
      <field name="commission"
        column="COMMISSION"/>
    </embedded>
  </field>
</class>
```