

Easier Java Persistence (EJP)[™]

EJP is an extremely powerful and easy to use persistence API for Java with no need for mapping annotations or configuration, as it features automatic object/relational mapping (A-O/RM)

Copyright © 2006 - Present, EasierJava All Rights Reserved.

EasierJava and EJP are trademarks or registered trademarks of EasierJava, Inc. in the U.S. and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All other marks are trademarks or registered trademarks of their respective holders in the U.S. and other countries.

OVERVIEW AND QUICK START.....	3
SUPPORTED DATABASES.....	6
LOGGING.....	6
DEFINING CLASSES.....	7
PRIMITIVE TYPES.....	9
SAVING NULL VALUES.....	9
SQL FRAGMENTS.....	10
INHERITANCE.....	10
ASSOCIATIONS.....	10
AUTOMATIC TRANSACTIONS.....	11
TRANSACTION MANAGER.....	11
THREAD SAFETY.....	12
DATA TYPES AND LARGE OBJECTS (BLOBS/CLOBS).....	12
DATABASES LACKING SUPPORT FOR AUTO-GENERATED KEYS.....	13
METADATA CLASHES.....	14
RETRIEVING A DATABASEMANAGER VIA XML DEFINITIONS.....	14
THE EJP.DATABASE CLASS.....	15
CLOSING EJP.DATABASE.....	16
THE EJP.RESULT CLASS.....	16
CLOSING A RESULT.....	17
LISTITERATOR RELATED FUNCTIONALITY.....	17
APPENDIX A: TROUBLESHOOTING.....	19
DATABASE SUPPORT.....	19
EJP IS ALWAYS USABLE.....	19
AUTO-GENERATED KEYS AND THE JDBC SPECIFICATION.....	19

Overview And Quick Start

EJP is a powerful and easy to use persistence API for Java. EJP's main features include:

- automatic object/relational mapping (A-O/RM)
- automatic handling of all associations
- automatic persistence tracking

Therefore, EJP has no need for mapping annotations or XML configuration, and there is no need to extend any classes or implement interfaces. You truly do use your Plain Old Java Objects (POJOs).

EJP is as easy as:

```
public static void main(String[] args)
{
    DatabaseManager dbm = DatabaseManager.getDatabaseManager(...);

    dbm.saveObject(new Customer("smith", "John"));

    Customer customer = new Customer("Smith");

    if ((customer = dbm.loadObject(customer)) != null)
    {
        customer.getSupport().add(new Support(...));

        dbm.saveObject(customer);
    }

    Collection<Customer> list = new ArrayList<Customer>();

    list = dbm.loadObjects(list, Customer.class);
    ...
}
```

It's used with normal class definitions like:

```
public class Customer
{
    String firstName, lastName;
    List<Support> support;
    List<Order> orders;
    ...
    public Customer(String lastName) { this.lastName = lastName; }
    ...
    public getFirstName() { return firstName; }
    public setFirstName(String firstName) { this.firstName = firstName; }
```

```

public getLastName() { return lastName; }
public setLastName(String lastName) { this.lastName = lastName; }
...
// Associations (automatic)
public List<Support> getSupport() { return support; }
public void setSupport(List<Support> support) { this.support = support; }
public List<Order> getOrders() { return orders; }
public void setOrders(List<Order> orders) { this.orders = orders; }
...
}

```

Because mapping is automatic any changes to the database won't affect your application objects, and changes to your object classes won't affect the database. EJP looks at both your class definition and database table definition and automatically figures out what needs to happen when inserting, updating, querying, and deleting.

EJP has full support for database transactions and automatically handles transactions with all types of associations. Transactions can also be handled with the `ejp.TransactionManager` class or manually with `beginTransaction()`, `endTransaction()`, `commit()` and `rollback()` functionality (methods in `ejp.Database`).

EJP automatically handles inheritance via multi-table joins, but also supports single-table and concrete-table inheritance. EJP easily handles associations with support for single-instance objects, arrays, and collections.

EJP has a database manager class (`ejp.DatabaseManager`) that handles pooling of its own resources as well as managing JDBC data sources and connections, as needed (including connection pooling and third party connection pooling libraries).

EJP wraps and uses JDBC functionality and can work with any relational database, and any type of connection resource, including: `java.sql.DriverManager`, `JNDI`, `javax.sql.DataSource`, and third-party connection pooling libraries (`DBCP`, `C3PO`, etc.).

The database manager also provides several object-oriented access methods that allow queries, inserts, updates, deletes, etc., in a single line of code. While `ejp.DatabaseManager` provides several convenience methods for accessing your data, the real power resides in the `ejp.Database` and `ejp.Result` classes.

An instance of `ejp.Database` is retrieved with `DatabaseManager`'s `getDatabase()` method. The `ejp.Database` and `ejp.Result` classes can serve any of the database access needs you may have.

An example of using `ejp.Database` and `ejp.Result`:

```

Database db = dbm.getDatabase();

Result<Customer> result = db.queryObject(Customer.class);

for (Customer customer : result)
{
    System.out.println(customer);
}

```

```
}  
  
db.close();
```

ejp.Database has several methods (all that JDBC provides) to easily handle transactions, for example:

```
Database db = dbm.getDatabase();  
  
db.beginTransaction();  
  
try  
{  
    db.saveObject(new YourObject(...));  
}  
catch (YourException e)  
{  
    db.rollback();  
}  
finally  
{  
    db.endTransaction(); // also commits transaction  
}
```

You can use the methods beginTransaction(), endTransaction(), commit(), getSavepoint(), and rollback() at anytime.

EJP also has a transaction manager class that allows the following:

```
new TransactionManager(dbm)  
{  
    public void run() throws DatabaseException  
    {  
        getDatabase().saveObject(new Contact("alincoln", "Abraham", ...));  
        // or just  
        saveObject(new Contact("alincoln", "Abraham", ...));  
        ...  
    }  
}.executeTransaction();
```

All in all, EJP's sole reason for being is to provide simple trouble-free POJO oriented access to your database data. With EJP and your POJOs, you should rarely have to do anything more than:

- Define a POJO which can be extended by other POJOs (relational hierarchy via joins).
- Call one of ejp.DatabaseManager's: loadObject(), saveObject(), and deleteObject(), or one of ejp.Database's: saveObject(), deleteObject(), and queryObject() with ejp.Result's loadObject().

- Group saves and deletes in a transaction using `ejp.TransactionManager` or `ejp.Database`'s transaction oriented methods.
- Most likely process a query result with:

```
while (result.hasNext())
{
    MyObject myObj = result.next();
    ...
}
```

That's not to say you won't have other JDBC needs, like calling stored procedures or executing SQL statements, but with EJP it's all easy to handle.

Supported Databases

Right now our focus is on adding support for as many databases as we can, and EJP currently has complete, and fully tested, support for:

MySQL
DB2
Oracle
Derby
HSQL
PostgreSQL
H2

You should be able to use EJP's complete functionality, without issue, with these databases. Of course, there is varying support from vendor to vendor for JDBC features. If your vendor doesn't support a certain feature (stored procedures, bi-directional cursors, etc.), you'll find the same is true with EJP, since it wraps JDBC.

Logging

NOTE: With logging set to DEBUG, you can see diagnostic information including the SQL that is generated from your objects.

EJP uses `slf4j` (www.slf4j.org). `Slf4j` is a simple logger facade 4 Java. It supports the major logging frameworks such as `log4j`, Java logging, `logback`, `commons logging`, etc.

`slf4j` (required):

Download from: www.slf4j.org
Must have `slf4j-api.jar` in your classpath.

`slf4j simple` (defaults to info):

Download from: www.slf4j.org

Copy slf4j-api.jar and slf4j-simple.jar to your classpath.

logback (defaults to debug):

Download from: logback.qos.ch

Copy slf4j-api.jar, logback-classic.jar and logback-core.jar to your classpath.

log4j (requires configuration):

Download from: <http://logging.apache.org/log4j>

Copy slf4j-api.jar, slf4j-log4j.jar and log4j.jar to your classpath.

Defining Classes

Defining a class to work with EJP is simple and no different than defining your typical POJO, except that you should always avoid using primitive types (see next section).

When executing an EJP “object-oriented” method (queryObject(), loadObject(), saveObject(), deleteObject(), etc.), EJP scans your objects for get/set methods that match columns in a table associated with the class name. The get method return types and set method parameter types should match the types defined in JDBC ResultSet.get.* and PreparedStatement.set.*, respectively.

With EJP, everything is based on naming, which can be overridden and/or manipulated with:

```
PersistentClassManager.setTableMapping()  
PersistentClassManager.setColumnMapping()
```

and

```
MetaData.setTablePrefixesToStrip()  
MetaData.setTableSuffixesToStrip()  
MetaData.setColumnPrefixesToStrip()  
MetaData.setColumnSuffixesToStrip()
```

You can also use:

```
Database.setMetaDataLimits();  
DatabaseManager.setMetaDataLimits();
```

to set limits on the database metadata that is accessible (pretty much required with some data bases, such as Oracle and DB2).

The name of your class, or its pluralized form, should be contained (*.name.*) in the database table name. For example:

```
class Customer {}
```

matches:

```
create table t_customers ();
```

However, it's also going to match anything else with `.customer.*`, so naming clashes are possible. But, they can be handled simply by defining longer/better names and/or with the above mentioned mapping overrides.

The same rules that apply to class/table matching also apply to method names (following the get/set) that will match up with table column names.

As an example:

```
CREATE TABLE t_contacts (  
    username varchar(40) NOT NULL PRIMARY KEY,  
    password varchar(20) NOT NULL,  
    first_name varchar(30) NOT NULL,  
    last_name varchar(40) NOT NULL,  
    ...);
```

will match up with:

```
public class Contact  
{  
    private String username, password, firstName, lastName,  
                companyName, email;  
    private Timestamp created;  
  
    public Contact() {}  
    public String getUsername() { return username; }  
    public void setUsername(String username) { this.username = username; }  
    public String getPassword() { return password; }  
    public void setPassword(String password) { this.password = password; }  
    public String getFirstName() { return firstName; }  
    public void setFirstName(String firstName) { this.firstName = firstName; }  
    public String getLastName() { return lastName; }  
    public void setLastName(String lastName) { this.lastName = lastName; }  
    ...  
}
```

Your classes can contain any methods you like, even methods not related to table columns in the database.

Methods that map to table columns can also perform any pre/post processing, as needed. The methods' only responsibility to EJP is to handle getting and setting values requested or presented by EJP.

Primitive Types

Important Note: It's very important to avoid using primitive types.

An important point to remember, as it relates to EJP, between a primitive type and its wrapper class (i.e. int and Integer) is primitive types always have a value (even if it's 0 or false), and should only be used in cases where the column should always be included in the generated SQL statement.

As an example:

```
public class Support
{
    private Long supportId;
    private String contactId, code, status, phone, email, request;

    public Long getSupportId() { return supportId; }
    public void setSupportId(Long id) { supportId = id; }
    ...
}
```

Notice in the Support class that "supportId" is defined as a Long (the wrapper version of long). With this definition, if we do `queryObject(new Support(new Long(10)))`, we get the support record with `supportId = 10`, and if we do `queryObject(new Support())`, we get all support records, because `supportId` is null. However, if "supportId" is primitive (long), then there is no way to say it's null, and `supportId` will always have a value, even if that value is 0. So instead of `queryObject(new Support())` returning all rows, it will return the row where `supportId = 0`.

Saving Null Values

As described above, EJP builds SQL statements from an object's non-null returning get methods. Often we may want to search the database for values that are null, or we might want to set a value in the database to null. EJP supports null values by simply defining the Null value to include with:

```
PersistentClassManager.setNullValuesToIncludeInQueries();
```

and/or:

```
PersistentClassManager.setNullValuesToIncludeInSaves();
```

This method takes a set of property names (the portion following get/set). With this option, values returned by methods corresponding to names defined in the set, will be included in the generated SQL statements regardless of whether they are null or non-null.

SQL Fragments

SQL fragments can be included in EJP's object-oriented database calls (`queryObject()`, `saveObject()`, `deleteObject()`, etc.). SQL fragments are parts of the SQL statement starting at any clause for the type of statement that will be generated.

For example:

```
db.queryObject(customer, "order by last_name");
```

or:

```
db.queryObject(customer, "where state = 'arizona' order by last_name");
```

EJP also allows the use of named parameters for including object properties (that will be translated to column names). For example:

```
db.queryObject(customer, "order by :lastName");
```

or:

```
db.queryObject(customer, "where :state = 'arizona' order by :lastName");
```

The named parameter can be anything that will match up to a column in the associated table, but is normally an object property (the name following the get/set).

EJP also allows JDBC IN parameters ('?'):

```
db.queryObject(customer, "where :state = ? order by :lastName",  
                new Object[] {"arizona"});
```

And, with the Java 1.5+ version's variable arguments support, you can use:

```
db.queryObject(customer, "where :state = ? order by :lastName", "arizona");
```

Inheritance

Inheritance is handled automatically and nothing further is required. Inheritance by default creates a multi-table join, but updates and inserts will be handled separately for each table. Deletes are only performed on the base table, as EJP relies on referential integrity (see the section on deleting objects).

EJP also supports single and concrete table inheritance. Any class in the inheritance hierarchy can use the annotation `ejp.annotations.Single[/Concrete]TableInheritance`.

Associations

Associations are handled automatically. Single instance objects, arrays and collections are supported.

Automatic Transactions

When calling `saveObject()`, regardless of state and persistence, if the connection's auto commit value is true, it will be set to false and a database level transaction will start (see `beginTransaction()`).

After all objects are saved (associations and inheritance relationships included), and if no errors occur, the transaction will be committed. If there are errors of any kind, the transaction will be rolled back.

However, if the connection is not in auto commit mode (`beginTransaction()` was already called), it's assumed the current call is part of a larger transaction. If the database supports save points, a save point will be set and any errors will be rolled back to the save point, preserving the larger transaction. If the database reports that it does not support savepoints, then EJP will not create a savepoint and it will not roll back the transaction to the savepoint (since the transaction was created outside), but it will throw an exception indicating an error occurred.

In every case, if an error occurs, a `DatabaseException` will be thrown with the embedded cause (typically a `SQLException`).

Transaction Manager

EJP has a transaction manager class that allows you to do the following:

```
new TransactionManager(dbm)
{
    public void run() throws DatabaseException
    {
        saveObject(new Contact("alincoln", "Abraham", ...));
        // or
        getDatabase().saveObject(new Contact("alincoln", "Abraham", ...));
        ...
    }
}.executeTransaction();
```

This class simply wraps the transaction functionality `beginTransaction()`, `rollback()`, and `endTransaction()` along with the `saveObject()` and `deleteObject()` functionality from `ejp.Database`.

To use the transaction manager simply override the `run()` method with your implementation and call `executeTransaction()`. If the transaction runs error free `executeTransaction()` will return normally. However, if there is an error the transaction

will be rolled back and the exception will be rethrown (wrapped in DatabaseException) from executeTransaction().

The transaction manager uses a single `ejp.Database` instance that is either passed in or obtained from an `ejp.DatabaseManager` instance that is passed in. Therefore, if using `ejp.Database` for database updates within the run method as part of the same transaction, it is important to use `getDatabase()` to ensure you're using the same instance.

EJP also permits the use of `commit()`, `setSavepoint()`, and `rollback()` at anytime within a transaction manager.

Note: It is not possible to safely use `DatabaseManager`'s `saveObject()` and `deleteObject()` in a larger transaction (non-automatic). The reason for this is due to the database manager retrieving a database with each call:

```
Database db = databaseManager.getDatabase();
```

which returns an available database (and connection), but not necessarily the same one with each call.

Always use the `TransactionManager`'s `getDatabase()` or `saveObject()` and `deleteObject()`.

Thread Safety

JDBC is not thread-safe, and as a result EJP is not thread safe. Therefore, an `ejp.Database` instance should not be shared with multiple threads unless locking (synchronization) is taking place. The best practice for thread safety is to make sure your methods are reentrant, and that you have a connection (`ejp.Database`) per thread via a `DataSource` that supports connection pooling. Getting an `ejp.Database` from the database manager is synchronized and so is returning the `ejp.Database` to the database manager. However, operations on the database are not protected (JDBC is not thread-safe).

Basically, if your application is multi-threaded and the methods are reentrant and you're working with a data source that supports connection pooling, then getting the `ejp.Database` and closing the `ejp.Database` as soon as possible within the same thread is all you'll have to worry about. The fact that you're working with connection pooling, and have a connection (`ejp.Database`) per thread will keep you thread-safe. But, it's very important to release the connection as quickly as possible, because it is likely being shared.

Data Types and Large Objects (Blobs/Clobs)

EJP supports all types supported by JDBC ResultSet and PreparedStatement. The only exception is that EJP requires wrappers for InputStreams and Readers. This is due to the fact that there needs to be a way to tell JDBC the length of the stream being stored (it's required by PreparedStatement and many JDBC drivers). There also needs to be a way to tell EJP which method is being used with InputStreams (Ascii, Binary or Character). The three wrapper classes are:

AsciiStream
BinaryStream
CharacterStream

All three have adapter classes:

AsciiStreamAdapter
BinaryStreamAdapter
CharacterStreamAdapter

AsciiStream relates to the PreparedStatement and ResultSet classes setAsciiStream and getAsciiStream (respectively), and the same is true for BinaryStream (setBinaryStream/getBinaryStream), and CharacterStream (setCharacterStream/getCharacterStream).

AsciiStream is used for handling large amounts of Ascii character data (LONGVARCHAR). BinaryStream is used for large amounts of binary data (LONGVARBINARY). And CharacterStream is used for large amounts of UNICODE character data (also LONGVARCHAR).

Data types defined by your class define the ResultSet and PreparedStatement methods that will be called when retrieving and storing data from and to the database. So, if you have Clob defined for your getMethod(), then PreparedStatement.setClob() will be used when storing the data. If you have Object as a parameter to your setMethod, then ResultSet.getObject() will be used when retrieving data.

Data types are database/driver specific. If a data type isn't supported by your database, there isn't much you can do other than to use a different type. Blobs and Clobs are the best example of a data type with varying support among vendors (they're relatively new).

Note: A useful debugging tool to use when you're not quite sure what the type is that's being returned for the table column, is to use Object as the type. Then ResultSet.getObject() is called and an appropriate object (as determined by the JDBC driver) is returned. You can then use Object.getClass() to figure out what is actually being returned.

Databases Lacking Support for Auto-Generated Keys

Note: This only applies to databases that are not currently supported.

Some databases lack support for returning auto-generated keys in JDBC. This means that during `saveObject()` calls, the object can't rely on a primary key for an object whose primary key is auto-generated and can't be retrieved. EJP will throw the following exception:

```
Could not reload object following save. Cannot make this object persistent. Your
database reports that it does not support retrieving auto-generated keys.
Therefore, you can't make objects relying on auto-generated keys persistent
following an insert. In this case, only loaded objects can be persistent. You can
either implement GeneratedKeys or set
PersistentClassManager.setReloadAfterSave(false).
```

The best scenario is to implement `GeneratedKeys`. However, you can also use `PersistentClassManager.setReloadAfterSave(false)`, which allows turning off object reloading following saves while still allowing the object to be persistent. While setting this feature to false greatly increases speed (no object reloading), use of this method could be a problem if the database provides any data manipulation during saves, as the object might not end up equaling what's in the database.

Metadata Clashes

Some databases return too much metadata and even metadata that belongs to other schemas. If you have five schemas with the same tables in them or tables with the same name, a database may return the first table found among the five schemas, and not necessarily the one belonging to your schema. We call this a metadata clash.

You can either set permissions on the schemas to ensure the user of a particular schema doesn't see tables in other schemas, and/or you can set metadata limits with the `databases.xml` file, the `DatabaseManager` constructor/factory methods, and with:

```
setMetaDataLimits(String catalogPattern, String schemaPattern, ...);
```

All subsequent calls will be limited to the catalog and schema that are defined. Either catalog and/or schema can be null. This method can be called anytime and set to other values at anytime. So you could use this combination to load and manage multiple tables with the same name from different catalogs/schemas, at will.

Retrieving a DatabaseManager Via XML Definitions

The database manager has several constructors that define the database connection and/or source. EJP can also pull this information from a file in the classpath named 'databases.xml' which has the simple format of:

```
<databases>
  <database name="" useJndi="" url="" [poolSize=""] [catalogPattern=""]
    [schemaPattern=""] [user=""] [password=""] />
  <database name="" driver="" url="" [poolSize=""] [catalogPattern=""]
```

```
        [schemaPattern="" ] [user="" ] [password="" ] />
</databases>
```

So retrieving an EJP database manager is as simple as defining:

```
<databases>
  <database name="jndiDb" useJndi="true" url="jdbc/mydb" poolSize="10" />
</databases>
```

and calling the method for XML defined databases:

```
DatabaseManager.getDatabaseManager("jndiDb");
```

There are also several constructors/methods to retrieve an `ejp.DatabaseManager` without using the `databases.xml` file.

The `ejp.Database` Class

Since the database class wraps the JDBC Connection class, EJP provides every need you may have when accessing your data. You can work with prepared statements, PL/SQL with callable statements, and simple SQL based queries. In fact, you can use SQL interchangeably at any time. You could say:

```
db.queryObject(new Contact("tnug%"));
```

or:

```
db.executeQuery("select * from Contact where contact_id like 'tnug%'");
```

EJP also allows you to interact with JDBC classes directly. You can, at all times, obtain the underlying JDBC Connection instance with:

```
db.getConnection();
```

When working with very large result sets, it is always better to use `ejp.Database` functionality in place of `ejp.DatabaseManager` object methods, as all `ejp.DatabaseManager` object routines (`loadObject()`, `loadObjects()`, `saveObject()`, and `deleteObject()`) simply wrap `ejp.Database` routines.

For example, the `ejp.DatabaseManager` method `saveObject(object)` is actually:

```
Database db = getDatabase();

try
{
  return db.saveObject(object, externalClauses);
}
finally
```

```
{
  db.close();
}
```

Most every ability available in JDBC is also available in `ejp.Database` (refer to the javadoc for more detail) and `ejp.Result`

Closing `ejp.Database`

It's important to always close an `ejp.Database` instance, especially if it is a shared resource. The `close()` method also closes any `ejp.Result` instances created while using the `ejp.Database` instance (unless they are already closed).

Also, always close an `ejp.Database` instance in a try-finally block. For example:

```
Database db = dbm.getDatabase();

try
{
  ...
}
finally
{
  db.close();
}
```

or in page rendering with:

```
#{db.close}
```

The `ejp.Result` Class

All `ejp.Database` queries return `ejp.Result`, which wraps the functionality of a JDBC `ResultSet` and adds object mapping abilities. `ejp.Result` is a `ListIterator` and can be traversed in both directions (if your database supports it). `ejp.Result` is also iterable, which allows it to be used in a `foreach` construct. For example:

```
for (Customer customer : result)
{
  System.out.println(customer.firstName);
}
```

A result instance is created automatically with a call to any of the `ejp.Database` query methods. A `Result` instance can also be created with one of the constructors:

```
Result(Database db, ResultSet resultSet)
```

```
Result(Database db, ResultSet resultSet, Class<T> cs)
```

If the first constructor is used, then the `ejp.Result` can be used to load any object, but no class is associated with the result, so the methods `next()` and `previous()` will not be able to return objects loaded with data. However, you can use `setClass()` to associate a class to a result, and then you can use the `next()` and `previous()` methods to obtain object instances loaded with the current row's data. You can also use `loadObject()` at anytime to load an object with the current row's data.

If the second constructor is used in the form of:

```
Result<T> r = new Result<T>(Database db, ResultSet resultSet, Class<T> cs)
```

then the result is associated with a specific class, and it is iterable. However, you can also be non-specific:

```
Result r = new Result(Database db, ResultSet resultSet, Class cs)
```

and, then it's the same as the first constructor minus the call to `setClass()`.

Closing a Result

Closing a result is done with:

```
result.close();
```

Since a JDBC `ResultSet` is also a database cursor, it follows that `ejp.Result` is also a cursor. And, since cursors are database resources and usually have a limit, it can be important to close `ejp.Result` instances as soon as possible. However, if that is not a worry and/or you are well below the limit, then you don't have to close the result each time and closing an `ejp.Database` will also close all results that were created with it.

ListIterator Related Functionality

`ejp.Result` implements `ListIterator` and `Iterable`, so a class specific instance is also iterable and can be used in `foreach` constructs, such as:

```
for (Customer customer : result)
{
    System.out.println(customer.firstName);
}
```

or used like an iterator:

```
while (result.hasNext())
{
    Customer customer = result.next();
    System.out.println(customer.firstName);
}
```

ListIterator is bidirectional, so you can use `ejp.Result` in the following manner:

```
result.setFetchDirection(ResultSet.FETCH_REVERSE);

while (result.hasPrevious())
{
    Customer customer = result.previous();
    System.out.println(customer.firstName);
}
```

Being able to traverse in reverse or bidirectional is completely database specific and many databases only support forward traversal.

Appendix A: Troubleshooting

Database Support

The areas most likely to vary from vendor to vendor are the vendor's support for Large Objects (Blobs/Clobs), auto-generated keys, ResultSet holdability, ResultSet type, ResultSet concurrency, etc.

Large Objects are relatively new to JDBC and databases are lacking unified support. Just about every vendor has a different level of support for large objects, but most support large objects in some form. Most allow retrieving Clob/Blob types from the database. Most implementations of Clob/Blob are read only (implementing only the read methods and not the write methods), and cannot be used to store data (see Serial[B/C]lob for storing). Most databases allow retrieving/storing large objects with InputStreams, Readers, Strings, and byte arrays. You'll find that working with large objects is very simple with EJP.

The remaining areas: auto-generated keys, ResultSet holdability, ResultSet type, ResultSet concurrency, etc., will in most cases throw an exception at the JDBC level, indicating the lack of support for a particular mode and/or method. As EJP wraps the Connection, Statement, and ResultSet classes, EJP will return many of the same errors that you may encounter with the Connection, Statement, and ResultSet classes.

EJP is Always Usable

Regardless of the level of support provided by your database and/or JDBC driver, EJP can always be used to queryObject(), saveObject(), loadObject(), deleteObject(), and most other functionality.

The only place where a problem could exist through a lack of JDBC/database support is with retrieval of auto-generated keys. And this can only keep an object from being able to persist following an insert. But, EJP will throw an exception telling you the database doesn't support auto-generated keys. Objects can always persist after being loaded from the database.

Auto-Generated Keys and the JDBC Specification

The JDBC specification defines the ability to retrieve values following an insert with the method getGeneratedKeys() which returns a ResultSet object containing the auto-generated keys. The spec defines multiple methods for telling JDBC to return auto-generated keys. Of interest is the version that allows defining the keys to return. Unfortunately, vendors have differing support for this feature, and the spec isn't clear.

To make things worse, the spec is flawed in the sense that any value could be auto-generated, not just Ids. Id columns can be auto-generated, and with databases that

support triggers, any column can be auto-generated. Furthermore, any column can be generated with a default value.

Therefore, the JDBC spec should add a method for obtaining auto-generated values rather than just ids. And following an insert or an update, JDBC should be able to return any columns requested. This would be similar to Oracle PL/SQL's returning clause, which is available in both inserts and updates. Until these issues are dealt with, true persistence simply isn't possible. EJP and any other database persistence API has to reload the object via a query following the insert/update to make sure the object values match what's in the database. Or, in the worst case (Hibernate/JPA), all generated data must be controlled in the software and not in the database.

One more problem with the JDBC spec is the fact that the auto-generated ids that are returned don't actually have any defined way to match up to a column, they are simply a ResultSet of ids. It's presumed they are in the order you asked for them, but if you ask for a column that isn't an id, some vendors simply don't return a value for the column, and you end up with more or less values than you asked for. Of course, this is due to the assumption that the database only allows one field to be an auto-generated id. However, as described above, any field can be auto-generated. Therefore, the return value should be a ResultSet similar to the ResultSet returned from a query with column/value pairs.

So what does all this mean? It means that support for generated keys is database specific and needs to be coded, at some level, for each database. If your database is not in the supported databases list, then you may have to implement the interface GeneratedKeys and handle generated keys on your own. This won't be an issue for most databases (or for long), and we will work on support for databases that aren't in the supported databases list, as a priority.