PriDE 3 User Manual

Last update: 27.03.2019

For version 3.4

Table of Contents

About PriDE

PriDE is the Java world's smallest object-relational mapper for SQL databases. O/R mapping is the wide-spread approach to map records of a relation SQL database to objects of an object-oriented application. The application should operate on its persistent entities as object-oriented as possible, not regarding that some of them come from a database or must be saved in one. PriDE provides functionality to

- Describe the mapping of database tables to Java classes
- Read and write data records without accessing the complicated JDBC interface, and - as far as possible - don't write any SQL at all
- Simplify the assembly of complicated SQL expressions and selection conditions

While O/R mapping is usually based on single-object operations, PriDE also supports efficient database mass processing within Java. The goal is to avoid shifting application logic into procedures within the database as far as possible and not to break the DRY principle. However, if stored procedures and functions are required sometimes, PriDE provides a convenient way to call them.

PriDE was designed for usage in JSE and JEE environments and is used identically everywhere except some initialization operations and the transaction management. The framework follows a very pragmatic approach to provide basic development support

quickly and easily. It does not claim to conform with established persistence management standards but follows common design patterns and proved to be suitable in mission-critical projects over many years. The detailed feature list may help to figure out whether PriDE meets the requirements of individual development projects, and allows to roughly compare this toolkit with existing well-known O/R mapping products and standards like JPA, JOOQ or MyBatis.

PriDE is so small that it can actually be understood in any single line of its code, providing the developer full control over how data is exchanged between an SQL database and a Java application. The runtime library is less than 200 kByte in size without any dependencies beside the JDBC driver library of the database in use.

Did you ever wonder how to conveniently access your SQLite database in a mobile application which has to keep its footprint small? Well, here is your answer :-)

Or did you ever worked in a multi-million lines of code project and got the feeling that JPA magic causes more loss of control than convenience? Guess what the alternative may be.

The chapter PriDE design principles gives an overview about the concepts which the framework is based on. However, before diving into more theory, it is recommended to walk through the quick start tutorial and get into touch with the real world.

About this manual

This manual gives you a complete overview about PriDE's standard functionality, design patterns and design principles. It is not mandatory to work through all the details unless you want to become an expert. However, it is strongly recommended to begin with the quick start tutorial as many code fragments in other chapters come back to the quick start example. The chapters Find and Query and Insert, Update, and Delete describe the core functionality you should become familiar with. All other chapters describe special aspects which you can dive into, when the time has come.

Many code fragments in this manual refer to existing example code which is available in an appropriate repository on GitHub. The repository includes a pom.xml file to run all examples on a local SQLite database. To have all source code at your fingertips at any time it is recommended to

- clone the repository (`git clone https://github.com/j-pride/manual-example-code.git`) and
- build the project using Maven (`mvn clean compile`)

At March 2019, the PriDE 3 manual is still work in progress. Beside the core chapters above there are chapters coming soon for the following aspects

Calling Stored Procedures

PriDE Design Principles

A good information source for features which are not yet covered by this manual are PriDE's unit tests which you can find on GitHub. You may also consult the manual from PriDE version 2 which is available on SourceForge.

Quick Start Tutorial

This short tutorial gives an introduction into the general working principles of PriDE, based on a simple example. It takes less than half an hour to set up a simple PriDE application which allows to perform basic operations on a single database table. The package quickstart of the PriDE manual example code repository on GitHub contains the complete source code for the tutorial example.

Setting up an application includes the following steps:

- Preparing the development project
- Database table design
- Writing or generating entity classes
- Writing application classes
- Running the application

I.e. there's only a few minutes time for each step now, so let's hurry up ;-)

Preparing the development project

Working with PriDE requires to add the library pride-x-y-z.jar into the CLASSPATH of the working environment, as well as the JDBC driver of the database to access. E.g. in case of a MySQL 6 database this is the library mysql-connector-java-6.y.z.jar, for Oracle 11 the library ojdbc8.jar, for HSQL 2.x the library hsqldb-2.y.z.jar, and for SQLite 3 the library sqlite-jdbc-3.y.z.jar. The ultra light database SQLite in server-less mode is the best choice for first experiments. You may set up a playground project by cloning PriDE's manual examples source code repository on GitHub and compile its sources with Maven, using the included pom.xml. However, as PriDE and SQLite do not depend on any other libraries, you can easily download the PriDE and SQLite JAR files from Maven central and create a project with any technique you like.

Driver class, database URL, database user, and password are supposed to be provided as system properties in this tutorial examples. For SQLite and a local example database, user and password can be omitted and the properties look like this:

```
pride.dbtype=sqlite
pride.driver=org.sqlite.JDBC
pride.db=jdbc:sqlite:pride.examples.db
pride.logfile=sql.log
```

Providing the DB type is recommended, to keep PriDE from making a wrong guess, and logging all SQL operations is usually a good idea - especially for beginners.

Database table design

PriDE follows a database-first approach, so in the next step, the required database table must be designed. PriDE does not provide its own tool for that but assumes one being included in your database installation. If nothing appropriate is around, their are lots of tools available for that, e.g. the free DB Designer online tool which supports various common databases. The tutorial examples uses a database table according to the following definition:

```
create table CUSTOMER (
    id integer not null primary key,
    name varchar(20),
    first_name varchar(30)
);
```

Add this table now to your SQLite database, using SQLite's command shell or by running class CreateCustomerTable included in the PriDE manual example code:

```
java
-Dpride.dbtype=sqlite
-Dpride.driver=org.sqlite.JDBC
-Dpride.db=jdbc:sqlite:pride.examples.db
-Dpride.logfile=sql.log
util.CreateCustomerTable
```

Writing or generating entity classes

Accessing a table via PriDE requires a corresponding entity class (usually a simple Java Bean) and a mapping descriptor object. 1:1 mappings of a database table to a Java class can be generated with a code generator provided with PriDE. For the table CUSTOMER above, the source code for a corresponding entity class Customer and an incorporated descriptor can be generated by the following call:

```
java
-Dpride.dbtype=sqlite
-Dpride.driver=org.sqlite.JDBC
-Dpride.db=jdbc:sqlite:pride.examples.db
-Dpride.logfile=sql.log
pm.pride.util.generator.EntityGenerator CUSTOMER quickstart.Customer >
Customer.java
```

The generator writes its output to the console, so you can either redirect the output to file as you see above or create the class in the IDE of your choice and copy the output from the console to your class editor. Note that you may also generate the descriptive parts in a *separate* class to keep the entity bean class free from database aspects. For the tutorial example we generate a hybrid class which looks like this:

```
public class Customer extends MappedObject {
    public static final String TABLE = "CUSTOMER";
    public static final String COL_ID = "id";
    public static final String COL_NAME = "name";
    public static final String COL_FIRST_NAME = "first_name";

    protected static final RecordDescriptor red =
      new RecordDescriptor(Customer.class, TABLE, null)
        .row(COL_ID, "getId", "setId")
        .row(COL_NAME, "getName", "setName")
        .row(COL_FIRST_NAME, "getFirstName", "setFirstName")
        .key(COL_ID);

    public RecordDescriptor getDescriptor() { return red; }

    private long id;
    private String name;
    private String firstName;

    // Read access functions
    public long getId()   { return id; }
    public String getName()   { return name; }
    public String getFirstName()   { return firstName; }

    // Write access functions
    public void setId(long id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setFirstName(String firstName) { this.firstName = firstName;
}



    // Reconstructor
    public Customer(long id) throws SQLException {
        setId(id);
        findXE();
    }

    public Customer() {}

}
```

Without going into details now, you can see an important design principle of PriDE: the mapping descriptor is code. You won't find any descriptive languages included in PriDE - neither XML nor property nor JSON files. Everything in PriDE is Java code and can be examined with a debugger if necessary.

Writing application classes

Based on the entity classes, you can design the actual application. First of all the PriDE runtime library must be initialized by a so-called "resource accessor". A JSE application requires only a single line of code for an initialization based on system properties:

```
ResourceAccessorJSE.fromSystemProperties();
```

The database operations are performed by invoking corresponding member functions of the entity classes, e.g.

```
public void create(int id, String name, String firstName)
    throws SQLException {
    Customer c = new Customer(id, name, firstName);
    c.create();
}

public void update(int id, String name, String firstName)
    throws SQLException {
    Customer c = new Customer(id, name, firstName);
    c.update();
}

public void queryByName( String name )
    throws SQLException {
    Customer c = new Customer(0, name, null);
    ResultIterator ri = c.query(COL_NAME);
    if (ri != null) {
        do {
            System.out.println(
                c.getId() + ": " +
                c.getName() + "," +
                c.getFirstName());
        } while(ri.next());
    }
}
```

Running the application

The tutorial example on GitHub includes the class CustomerClient, providing an interactive test client. Calling the client with its system property based initialization looks like this:

```
java
-Dpride.dbtype=sqlite
-Dpride.driver=org.sqlite.JDBC
-Dpride.db=jdbc:sqlite:pride.examples.db
-Dpride.logfile=sql.log
quickstart.CustomerClient
```

Play around with the client and then check the working directory. You will find a file sql.log created by PriDE which logs all the SQL statements that resulted from your persistence

operation calls. The log file is plain SQL, so if you encounter any unexpected persistence behavior in you application, you can copy the commands from the log and run them from you database's SQL shell. This is a big advantage over the command logging of most other persistence frameworks.

Actually PriDE is working with plain SQL by default rather than with so-called "bind variables". So what you see in the log file is exactly what PriDE executes against the database resp. its JDBC driver. If you like to change to bind variables as default, you can add the configuration property `-Dpride.bindvars=on`. Restart the CustomerClient with this system property set, run some commands and check the log file again. You still see plain SQL but the values of insert and select statements are now preceded by a ? indicating that the value was passed to the database via a bind variable. You can still copy every command from the log to an SQL shell - you only have to remove its question marks before running it interactively. You won't find many persistence managers with a comfortable SQL logging like that.

A meaningful usage of bind variables becomes a relevant issue for heavily accessed databases and will be discussed in the prepared operations chapter.

That's it!

The tutorial example already introduces the most important basic elements of PriDE. To understand what's going on behind the scenes of the 5 steps above, you will find all aspects explained in detail in the manual.

Before you go ahead…

Before you go on you should simplify the configuration in a way that you don't have to provide system properties every time you call a client. Although PriDE does not *require* any descriptive languages, they are sometimes quite helpful. To simplify the playing-around with examples, all client programs included in the PriDE manual examples use the class util.ResourceAccessorExampleConfig for initialization. It allows to assemble the configuration from two sources: system properties as it was introduced so far and a property file config/pride.examples.config.properties. As the configuration properties probably stay unchanged through all your experiments, you should transfer all your command line system properties to file (without the leading "-D" of course) and start your client programs without passing any system properties at all. To run the entity generator with the file-based configuration call the wrapper class util.EntityGeneratorWithExampleConfig which is also included in the examples.

Entity, Adapter, and Descriptor

The concept of O/R mapping requires three basic building blocks:

- Entity classes which representing data in SQL tables - in the most common usage one entity object represents one record in one SQL table
- Descriptors, describing how the entity classes map to the database

- An adapter which reads data from the database to entities (select) and writes data from entities to the database (insert, update, delete)

There are very different approaches around how to express the descriptor. JPA uses annotations on entity classes, MyBatis uses XML files, and PriDE follows a different approach as you may have seen already from the quick start tutorial. The descriptor is an instance of class pm.pride.RecordDescriptor, i.e. it is code itself. This concept has a few advantages over other approaches.

- It does not clutter the entity classes with database details, so entities can be passed around in the application without violating the information hiding principle. If you are familiar with JPA you may have experienced the problem that mapping annotations can pile up to an annoying amount.
- Its not written in a different language which is always hard to keep in sync with the Java code. This becomes a serious problem when applications grow over time. If you are only working with three database tables, you won't have this problem, of course.
- If it is Java code, it can be tied to any other related Java code by using shared constants for table named and row names and so on. This allows you to easily keep track of dependencies in the code. E.g. if you remove a column from a database table you will remove the appropriate constant in the code and every mentionable IDE will immediately lead you to all the places in the code that do not compile any more. This will include the descriptor as well as all the database queries in the code that refer to that column.
- Descriptors may also be assembled dynamically at runtime. You hopefully will not often run into situations where you need that, but its good to know that there is no limit on that.

A coded descriptor needs to go somewhere in your code, of course. PriDE provides two default patterns for the descriptor placement which are obvious when you think of the building blocks mentioned above: descriptors within adapter classes or descriptors within entity classes.

Descriptors in entities is what you know already from the quick start tutorial. It cases the entities to become their own adapters having their own persistence methods. This is a compact pattern which is suitable for small applications. Therefore you will find it spread over most examples provided with PriDE. The disadvantage is the same one mentioned above with JPA: the entity classes spread knowledge about database mapping information all over the code. Combined with persistence capabilities directly incorporated in entity classes, this is a questionable concept in bigger architectures.

Let's have a look on the more sophisticated pattern of separate adapter classes. You can have a look on the general structure by generating separate classes for the quick start example table. The pure entity class can be generated by the following command:

```
java
-D... see quick start tutorial
pm.pride.util.generator.EntityGenerator CUSTOMER adapter.CustomerEntity -b >
CustomerEntity.java
```

The parameter -b tells the generator to create only an entity class without descriptor. The result is an ordinary Java bean or POJO class:

```
package adapter;

public class CustomerEntity implements Cloneable, java.io.Serializable {
    private long id;
    private String name;
    private String firstName;

    public long getId()   { return id; }
    public String getName()   { return name; }
    public String getFirstName()   { return firstName; }

    public void setId(long id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setFirstName(String firstName) { this.firstName = firstName;
}

    // re-constructor
    public CustomerEntity(long id) {
        setId(id);
    }

    public CustomerEntity() {}
}
```

The “re-constructor” is an additional constructor getting passed a value for all the attributes making up the entity’s primary key. This of interest for find operations.

Generating the corresponding adapter class looks like this:

```
java
-D... see quick start tutorial
pm.pride.util.generator.EntityGenerator CUSTOMER adapter.CustomerAdapter
adapter.CustomerEntity > CustomerAdapter.java
```

The first parameter after the table name specifies the class to generate - in this case a class called CustomerAdapter in package adapter. The second parameter is the name of a entity class the adapter should refer to. The result looks like this:

```
package adapter;

public class CustomerAdapter extends ObjectAdapter {
    public static final String TABLE = "CUSTOMER";
    public static final String COL_ID = "id";
    public static final String COL_NAME = "name";
    public static final String COL_FIRST_NAME = "first_name";

    protected static final RecordDescriptor red =
      new RecordDescriptor(CustomerEntity.class, TABLE, null)
```

```
            .row(COL_ID, "getId", "setId")
            .row(COL_NAME, "getName", "setName")
            .row(COL_FIRST_NAME, "getFirstName", "setFirstName")
            .key(COL_ID);

    public RecordDescriptor getDescriptor() { return red; }

    CustomerAdapter(CustomerEntity entity) { super(entity); }
}
```

All what the adapter class has to provide is a RecordDescriptor and an optional list of column names making up the entity's primary key. Based on that, the class inherits all entity-related persistence capabilities from class pm.pride.ObjectAdapter. Adapters always operate on an instance of the entity class which must be passed in the adapter's constructor. Finding a customer by its primary ID looks like this when using separate adapter classes:

```
// Create a customer entity, initialized with a primary key value of 1
CustomerEntity customer = new CustomerEntity(1);

// Create an adapter based on the entity
CustomerAdapter adapter = new CustomerAdapter(customer);

// Call the adapter's find method to find a customer by primary key 1.
// The primary key value is read from the entity passed in the adapter's
constructor
// The result (if any) is written to the same entity
adapter.find();
```

As you see, every persistence operation now requires one additional line of code to create the adapter. Especially when you design a multi-threaded application, it is important to know that adapter and entity instances are not supposed to be shared among multiple threads. So creating new instances in every operation is the prefered technique and is usually not a considerable code complication.

If you want to minimize the amount of code, you are free to invent your own adapter concept. Have a look on the base classes pm.pride.ObjectAdapter for the adapter above and pm.pride.MappedObject for the hybrid variant from the quick start tutorial. Both are minimalistic implementations of the mix-in pm.pride.DatabaseAdapterMixin which is the actual provider for all entity-related persistence operations. It is in turn based on the static methods of the class pm.pride.DatabaseAdapter. Using this class or the mix-in you could easily produce a generic adapter being responsible for multiple entity types similar to JPA's EntityManager interface.

One note concerning packages: When you actually use the pattern of separate adapters in a sophisticated architecture, you should consider generating entity and adapter classes in different packages. Only the entity classes should be part of the interface for dependent code while the adapter classes should completely be hidden behind facade components as proposed in the wide-spread repository pattern.

Descriptor structure

The examples for descriptors you have seen so far should already clarify most of the descriptor structure. You will see more complicated examples in following chapters of this manual. A descriptor is assembled from the following information:

- The name of the entity class and the name of the database table which the entity class is mapped to. Preferably the table name is not specified as a string-literal but as a reference to a constant representing the table name. If you have a look on the outcome of PriDE's code generator, there are appropriate constants generated and used.
- A reference to the descriptor of a base class. This is of interest when you build up an inheritance hierarchy between entity classes as explained in chapter Entity Inheritance.
- A table-row to attribute mapping by adding calls of the row() method for every row of interest. The row() method returns the descriptor object, making up a fluent API. Every row/attribute mapping consists of
  - The name of the database column (similar to table names: avoid using string-literals here)
  - The name of the getter method for the corresponding attribute in the entity class
  - The name of the setter method

  The methods are the ones which the adapter is supposed to use for transporting entity attributes to the database via JDBC and vice versa. The getter methods' return type implies which methods the adapter uses to access JDBC statements and result sets and how to translate the values to SQL syntax. Getters are mandatory whereas setters may be null in case of entity types that are never supposed to be written to the database. A typical example for this case are entity classes representing the result of SQL joins (see chapter Joins).
- An optional primary key definition by adding a call of the key() method with a list of column names making up the primary key. Alternatively you can use rowPK() instead of row() for the row mappings.

The RecordDescriptor class has a few more constructors concerned with joins and accessing multiple databases, but that's not important for now. The basic structure described above is what you work with most of the time.

Attribute Type Mapping

The following table illustrates the mapping of Java object attribute types to SQL database field types as they are supported by PriDE. The row *'JDBC type'* determines the type being used for the specified attribute type to access results from a JDBC ResultSet or to pass inputs to a JDBC PreparedStatement. The *'SQL type'* specifies the actual SQL row types, the attributes can usually be mapped to. Not all SQL databases support all the mentioned type identifiers and it may also depend on the JDBC driver's capabilities which mappings are

supported. Primitive attribute types should of course only be used, if the corresponding row must not be NULL. Otherwise an exception will be thrown at runtime when attempting to process NULL values.

Java attribute type	JDBC type	SQL type
String	String	VARCHAR, VARCHAR2, NVARCHAR2, CHAR
java.util.Date	java.sql.Date	DATE, DATETIME, TIMESTAMP, TIME
java.sql.Date	java.sql.Date	DATE, DATETIME, TIMESTAMP, TIME
java.sql.Timestamp	java.sql.Timestamp	DATETIME, TIMESTAMP, TIME
int / Integer	Integer	INTEGER
float / Float	Float	DECIMAL, REAL
double / Double	Double	DECIMAL, REAL
Any enum	String	VARCHAR, VARCHAR2, NVARCHAR2, CHAR
boolean / Boolean	Boolean	BOOLEAN, INTEGER, SMALLINT, TINYINT, CHAR
BigDecimal	BigDecimal	DECIMAL, NUMBER
long / Long	Long	INTEGER, DECIMAL, NUMBER, BIGINT
short / Short	Short	INTEGER, SMALLINT, TINYINT, DECIMAL
byte / Byte	Byte	TINYINT
byte[]	byte[]	BLOB, LONGVARBINARY, VARBINARY
java.sql.Blob	java.sql.Blob	BLOB, LONGVARBINARY, VARBINARY
java.sql.Clob	java.sql.Clob	CLOB, LONGVARCHAR
java.sql.SQLXML / String	java.sql.SQLXML	java.sql.SQLXML

Clobs and Blobs can only be used through PreparedStatements, i.e. you either have to access the entities with Clob / Blob attributes with PriDE's prepared operations or you configure PriDE to use bind variables by default (see quick start tutorial).

The precision of dates and time stamps in the database vary significantly between different database vendors. E.g. although date rows were originally intended to represent dates without time portions in SQL databases, Oracle allows seconds precision instead and so does PriDE for Oracle. When using PriDE with plain SQL, dates and timestamps are rendered by appropriate database-specific formating functions like `to_date` or `to_timestamp` in Oracle, preserving the same precision as it applies to prepared statements.

The interface `pm.pride.ResourceAccessor` provides the constant *SYSTIME_DEFAULT*. In update and insert operations this values will be translated to an expression which addresses the current database server time like *CURRENT_TIMESTAMP* in MySQL or *SYSDATE* in Oracle. This translation is only applied in plain SQL.

You can tell PriDE to map a java.util.Date attribute to an SQL time stamp by providing the JDBC type in the row definition of the record descriptor as an additional parameter like that:

```
.row(<columnname>, <getter>, <setter>, java.sql.Timestamp.class)
```

There are a few more type conversions which can be expressed that way. E.g. Enums can be represented by their ordinals in the database by providing `java.lang.Integer.class` as additional parameter for the mapping of the corresponding attributes.

The general pattern for arbitrary type conversion is to provide appropriate additional getter/setter pairs which encapsulate the conversion. To make clear, that these getters / setters are for internal use only, it is common practice to give the method names a leading underscore. Assume you have an enumeration type for coins with their value in cent like that:

```
public enum Coin {
    FIVE_CENT(5), FIFTY_CENT(50), ONE_EURO(100);

    private int valueInCent;
    Coin(int valueInCent) { this.valueInCent = valueInCent; }
    int value() { return valueInCent; }
}
```

If you map an attribute of this type to the database, PriDE expects an SQL row of type VARCHAR or a similar type to store values like 'FIVE_CENT' etc. If you want to represent the coins by their value in the database, you provide a type-converting getter-setter-pair for the corresponding attribute in the entity class:

```
class MyEntity {
    private Coin myAttr;

    public int _getMyAttr() {
        return myAttr.value();
    }

    public void _setMyAttr(int v) {
        for (Coin coin: Coin.values()) {
            if (coin.value() == v) {
                myAttr = coin;
                return;
            }
        }
        throw new IllegalArgumentException();
    }
}
```

Now you can use this getter setter pair to map the myAttr attribute to a DECIMAL table row, holding the coins' values:

```
.row("MYATTR", "_getMyAttr", "_setMyCoin")
```

Find and Query

The terms “find” and “query” for data retrieval are used in the same sense in PriDE as you may know it from other persistence concepts. Finding means to select data with the expectation to retrieve 1 result or none treating the presence of multiple results as an exception case. The most common example is a selection by primary key.

A query means to select data with an unpredictable number of result. PriDE is designed to take “unpredictable” literally and allows to process even millions of results in an efficient way in Java.

Find

Examples for finding a record with PriDE were already part of the quick start tutorial and the chapter about entity, adapter, and descriptor. But let’s go into some details here for a deeper understanding. The important things to know:

- No matter if you are working with hybrid objects or a separation of entity and adapter - PriDE avoids creating entities by itself but expects you to provide them.
- Find operations work like a query-by-example. You provide an entity with all the primary key fields initialized and call the find() or findXE() method without parameters. This is a method of the entity class itself when using hybrid entities, otherwise it is a method of the corresponding adapter class.
- The result of the find operation is placed in the same entity which you provided the key fields by. The boolean return value of the find() method tells the caller if there was actually a matching record found. The findXE() method reports a missing match by an exception which is of interest for situations where a missing result is a unexpected case. Think of typical navigation like retrieving the customer who placed an order. You usually don’t expect the customer not being present in the database.
- The alternative method findRC() can be used to provide the resulting record in a *copy* of the original object. The copy is preferably created by cloning the original object in a clone() method with public visibility. This keeps the responsibility for object creation in the hands of the application code. Otherwise the entity class must provide a copy constructor or a default constructor.

When you are working with a generated hybrid entity, a find operation by primary key fields is a single line of code like that:

```
Customer customer = new Customer(1);
```

PriDE’s generator produces a so-called “re-constructor” if the referred database table has a primary key. The re-constructor expects a parameter for all attributes making up the primary key, initializes the entity accordingly and calls the entity’s findXE() method. I.e. if the retrieval by primary key fails, the re-constructor throws a pm.pride.FindException. The FindException is derived from java.sql.SQLException which must be handled anyway.

When you are working with separate adapter classes, the same operation takes two lines of code:

```
CustomerEntity customer = new CustomerEntity(1);
new CustomerAdapter(customer).findXE();
```

Query

Whenever selecting multiple records from the database, PriDE returns a pm.pride.ResultIterator to iterate through the results. The ResultIterator encapsulates a java.sql.ResultSet, i.e. it is an open database cursor which is suitable for any amount of results. Taking up the example from the quick start tutorial, you can select all customers from the CUSTOMER table by the following lines of code:

```
Customer customer = new Customer();
ResultIterator ri = customer.queryAll();
```

To allow the processing of large amounts of records, the ResultIterator works slightly different from what you may be used to.

- To step through the results, you have to call the iterators next() method which returns false if there are no more results available.
- Instead of creating a new entity with every step, the iterator provides all results in the entity which the query was initiated from. I.e. every call of next() overwrites the data from the step before. No matter how many results you have, you will not run into memory problems when you directly process the results within the iteration loop.
- The first result from the query gets initially written to the entity, so the iteration process usually requires a do-while-loop rather than a while-loop.
- If there are no results at all, the query functions returns a ResultIterator which returns true from its isNull() method.

Pulling all this together, an iteration for direct result processing looks like that, e.g. if we would like to print all customers to the console:

```
Customer customer = new Customer();
ResultIterator ri = customer.queryAll();
if (!ri.isNull()) {
    do {
        System.out.println(customer);
    }
    while(ri.next());
}
else {
    System.out.println("No customers found");
}
```

A ResultIterator must be closed when the iteration is over, because it holds an open java.sql.ResultSet inside which in turn holds an open database connection. For convenience, the ResultIterator closes its ResultSet automatically when your code iterates to the end or if there occurs a database exception while fetching results. So usually you don't have to care about closing the iterator. For special cases, call the close() method.

The direct iteration is a highly efficient option on the one hand (PriDE is a lot faster with this approach than any JPA implementation), but on the other hand it is not the typical case. Usually the amount of results is small and they don't need to be processed on such a low layer of the application. Instead you may want to pass them as a list or array to a higher application layer where the business logic resides in. In this case, you can call appropriate functions on the ResultIterator:

```
// Extract the customers as list
List<Customer> allCustomers = ri.toList(Customer.class);

// Extract the customers as list with a limitation for the amount of results
List<Customer> allCustomers = ri.toList(Customer.class, 100);

// Extract the customers as array
Customer[] allCustomers = ri.toArray(Customer.class);
```

PriDE produces the entities in the lists and arrays in the same way as mentioned earlier for the findRC() method: by cloning the original entity with a public clone() method, by a copy constructor, getting passed the original entity as a parameter, or by a default constructor. All entity types generated by PriDE have a clone() method with public visibility implemented, based on Java's protected default implementation. This is a single-line implementation, so it's very simple to provide even if you are not using the generator.

In general it is strongly recommended to define a base class for all entity types where all those standard capabilities are encapsulated. Not only a clone() implementation but also a reasonable default (reflection-based) toString() method and maybe even a set of standard attributes as explained in chapter Entity Inheritance.

Streaming

Another alternative form of result processing are the ResultIterator's stream methods. There are two different methods available.

The method `stream(Class)` provides the results as a "real" stream with a new result instance for every result. The resulting stream is suitable for any kind of Java stream operations but should be used with care when selecting a very large number of results combined with stream operations which have to keep all the results (e.g. sort and collect operations). This may cause serious memory problems.

The method `streamOE(Class)` provides all results in the original entity just as it is the case in direct iterating and processing demonstrated above. This kind of stream is suitable for any amount of results but can only be used for a limited set of stream operations. Especially operations that rely on object identity will usually not work. Operations for direct processing like forEach() or count() won't cause any problems. The direct processing example from the beginning of the query section would look like this when using streams:

```
Customer customer = new Customer();
customer.queryAll().streamOE()
    .forEach(c -> System.out.println(c));
```

Examples for find and query code can by found in the class QueryClient in the package query of the PriDE manual source code repository on GitHub.

Selection criteria

PriDE has a few different features to assemble SQL where-clauses for queries:

- Query by example
- The builder class WhereCondition
- Completely self-defined conditions

Query-by-example is something you already come across in the section about finder methods. It is addressed by the method queryByExample(String… dbfields) which is available in every adapter and every hybrid entity class. The where-clause is assembled from an equality expression for all the database columns being passed to the function call where the values are taken from the corresponding attributes of the entity. Of course, the entity must be initialized accordingly first. Taking up the Customer entity from the quick start tutorial, the following query-by-example would allow to find all customers with first name “Peter”:

```
Customer customer = new Customer();
customer.setFirstName("Peter");
ResultIterator ri = customer.queryByExample(Customer.COL_FIRST_NAME);
```

The query takes all the specified columns into account, considering also Null-values in appropriate attributes. E.g. passing Customer.COL_FIRST_NAME, Customer.COL_NAME to the query method without setting a name value in the customer entity, the resulting SQL query will look like this:

```
select id,name,first_name from CUSTOMER where ( first_name = 'Peter' AND name
IS NULL )
```

Remember that yo can always check the SQL log file to find out what SQL statements have been assembled by PriDE. And always remember to use constants for the column names as you can see above rather than string literals. This allows you to keep track of which code depends on which aspects of your data model.

WhereCondition

Query-by-example is easy to use but limited to equality expressions. A more sophisticated tool in PriDE is the WhereCondition class. It allows to assemble more complicated queries with a fluent API as a compromise between syntax and type safety on the one hand and code readability and simplicity on the other hand. Lets start with a simple example of a where condition, producing the same query as above for an empty name and the first name “Paddy”:

```
WhereCondition byFirstNameAndEmptyName =
    new WhereCondition(Customer.COL_FIRST_NAME, "Paddy")
    .and(Customer.COL_NAME, null);
ResultIterator ri = new Customer().query(byFirstNameAndEmptyName);
```

The basic principle of the class is straight-forward:

- The function and() creates a sub-condition which is is AND concatenated with the condition being assembled so far. The same applies to the corresponding or() method.
- The and() function returns the WhereCondition itself which causes the class and its methods to become a fluent API.
- The and() function which only gets passed a field name/value pair produces an equality expression.
- The name/value pair passed to the constructor makes up the initial condition. To keep from mixing real constructor parameters with the first condition fragment, it is recommended to use the following form instead:

```
WhereCondition byFirstNameAndEmptyName = new WhereCondition()
    .and(Customer.COL_FIRST_NAME, "Paddy")
    .and(Customer.COL_NAME, null);
ResultIterator ri = new Customer().query(byFirstNameAndEmptyName);
```

It doesn't make a difference if you start with the or() method or the and() method at the top. Both methods are available in the more flexible variant `xxx(String field, String operator, Object... value)`. The parameter *operator* is an SQL operator with the commonly known ones listed in the *Operator* interface within the WhereCondition class. Using String rather than a type-safe Enum keeps the API open for future extensions and vendor-specific operators. Multiple values can be passed for the operators WhereCondition.Operator.BETWEEN and WhereCondition.Operator.IN. E.g. selecting customers with first name "Paddy" or "Mary" can be expressed by

```
.and(COL_FIRST_NAME, IN, "Paddy", "Mary");
```

The variants `xxxNotNull(...)` will only add the sub-condition if the (first) field value differs from Null. This is of interest for the assembly of conditions from interactive search criteria input. An empty criterion usually means 'do not consider' rather than 'must be empty'.

The variant without parameters opens up a sub-condition which must be completed by function bracketClose(). The following condition looks for early customers (id less than 1000) that registered with a suspicious name "Mickey Mouse":

```
WhereCondition byMickeyMouse = new WhereCondition()
    .and(COL_ID, LESS, 1000)
    .and()
        .or(COL_FIRST_NAME, IN, "Mickey", "Mouse")
        .or(COL_NAME, IN, "Mickey", "Mouse")
    .bracketClose();
```

What of you are interested in other suspicious cases where name and first name are equal. In this case, the value is a field name itself and you have to bypass the value formatting. This is achieved by passing pre-formatted SQL values like that:

```
.and(COL_FIRST_NAME, SQL.pre(COL_NAME))
```

Finally the WhereCondition can be extended by ordering and grouping clauses. E.g. the following condition selects all customers ordered by name and first name:

```
new WhereCondition().orderBy(COL_NAME).orderBy(COL_FIRST_NAME)
```

Especially when you select data *in order*, the ResultIterator provides the additional methods spoolToList() and spoolToArray() to read results in chunks. This allows to run multiple co-ordinated selects in parallel as an alternative to joins when selecting along 1:N relationships. Joins cause a duplication of transfer data in such a case which may cause mentionable latency in very large selections.

And finally finally the methods bindvars…() in the WhereCondition class give you fine-grained control over which parts of the expression should use bind variables and which ones should be plain SQL. Bind variables become a relevant issue for databases on heavy duty and therefore are also discussed in the prepared operations chapter.

Arbitrary Criteria

When the going gets tough there is a method `query(String where)` available in all adapters and hybrid entities. The function gets passed a fully assembled where-clause without the leading where keyword. It takes any limitations away but also a lot of convenience and safety. Assembly of complicated SQL expressions may not only become an issue in where-clauses but in any multi-record operation likes joins, merge statements, or mass updates. PriDE can help you assembling these expressions with the class pm.pride.SQLExpressionBuilder resp. the function pride.pm.SQL.build(). It preserves the native readability of complicated SQL on the one hand and allows you to work with table name and column name constants on the other hand to preserve code dependency tracking. In the preceding chapters you learned already that the PriDE principles heavily emphasize this central aspect for robust application design. The expression builder is addressed in a separate chapter. However, to give you a first impression, here is an example how to build the most complicated expression above - the Mickey Mouse case - using the expression builder:

```
String byMickeyMouse = SQL.build(
    "@ID < 1000 AND (" +
    "  @FIRST_NAME IN ('Mickey', 'Mouse') OR " +
    "  @NAME IN ('Mickey', 'Mouse')" +
    ")",
    COL_ID, COL_FIRST_NAME, COL_NAME);
ResultIterator ri = new Customer().query(byMickeyMouse);
```

This is a very limited example but you may already recognize the advantage over using the convenient WhereCondition fluent API: The SQL code is plain to see in nearly its native structure and notation although the actual assembly still makes use of the table and column name constants. The expression string may contain ? characters for bind variables. In this case, the variable values must be appended to the expression in the query() method call. The following example uses that feature for the ID threshold of 1000, making up the same selection as above:

```
String byMickeyMouse = SQL.build(
    "@ID < ? AND (" +
    "  @FIRST_NAME IN ('Mickey', 'Mouse') OR " +
    "  @NAME IN ('Mickey', 'Mouse')" +
    ")",
    COL_ID, COL_FIRST_NAME, COL_NAME);
ResultIterator ri = new Customer().query(byMickeyMouse, 1000);
```

Using bind variables is the easiest way for you as a developer to overcome the problem of value formating, which may become a bit tricky for complex data types like dates and timestamps. However, this should of course not be the main reason to use bind variables. See chapter Prepared Operations for purposeful usage. As long as you are working with small databases, it is OK just to utilize the formatting side effect.

If you want to learn more about the expression builder right now, read the chapter SQL Expression Builder.

Insert, Update, and Delete

The basic functionality for inserting, updating and deleting data is very simple. In addition to the basics, this chapter also explains how to manage transactions which is of course a very important issue when you manipulate the data. For most code snippets in this chapter you can find example code in package modify in the PriDE manual source code repository on GitHub.

Insert

To insert a record in a database table, you create an instance of the corresponding entity class, set all its attributes and call its create() method:

```
Customer customer = new Customer();
customer.setId(57);
customer.setName("Fingal");
customer.setFirstnae("Paddy");
customer.create();
```

When you are working with separate adapters instead, create() is a method of the adapter and the code looks like that:

```
CustomerEntity customer = new CustomerEntity();
customer.setId(57);
customer.setName("Fingal");
customer.setFirstnae("Paddy");
new CustomerAdater(customer).create();
```

You can insert multiple records successively using the same entity (and adapter) by changing the entity's data and repeatedly call create(). This is OK for small amounts of inserts. If you have to insert thousands or hundreds of thousands records, you better work with the class pm.pride.PreparedInsert as explained in chapter Prepared Operations.

If the addressed database table has auto-increment rows, you can specify these rows in the descriptor by a call of method auto() with a list of column names. In this case you leave the appropriate attributes uninitialized, and after creation PriDE will set them according to the values generated by the database. Expressing auto-incrementation in a database table definition is always a bit vendor-specific as well as the supported generation features in general. Supposed you are still experimenting with the SQLite database from the Quick Start Tutorial, you could create a modification of the CUSTOMER table as follows to make the ID and auto-increment row:

```
create table AUTOINCCUSTOMER (
    id integer not null primary key AUTOINCREMENT,
    name varchar(20),
    first_name varchar(30)
);
```

An appropriate hybrid entity class looks exactly like the one from the Quick Start Tutorial only extended by the following line at the end of the descriptor definition:

```
protected static final RecordDescriptor red =
    //...
    .auto(COL_ID);
```

Based on that, the following loop creates 10 unique test customers in a row and prints out the auto-generated ID of each of them:

```
Customer customer = new Customer();
for (int i = 0; i < 10; i++) {
    customer.setName("Fingal-" + i);
    customer.setFirstName("Paddy");
    customer.create();
    System.out.println(customer.getId());
}
```

You find an example for a customer class with auto-increment ID in package modify in the PriDE manual source code repository on GitHub.

Transactions

Try to write a loop as above without anything else and you will recognize that it does not produce any rows at all in you CUSTOMER table. Most SQL databases are fully transaction-saved by default and thus require the application to properly commit its work. The foundation for transaction management with commit and rollback is the ACID principle, which every developer must be well aware of as it is deep-seated in JDBC and every JDBC-based persistence manager.

In a JSE environment you will by default lose all your database work when the application terminates and you forgot to explicitly run a commit operation. In enterprise environments like standard JEE or Spring the application is usually not responsible for ending transactions by programmatic operations. Most applications use so-called container-managed transactions which are implicitly controlled by method annotations and exception handling. This is a very convenient and recommendable technique and it is also the key for

composing higher-level methods from calls of lower-level methods in an elegant way according to the Single Level of Abstraction Principle. No method has to worry about whether it is the very top-level of the (potentially still growing) composition tree.

PriDE does not really manage transactions by its own but relies on the transaction management of the environment it is used in. The link between PriDE and its environment is the ResourceAccessor interface and you have to install one somewhere in your application. The chapter JSE, JEE, and ResourceAccessor will explain that in detail. Except in this chapter, all other examples in this manual are working on a simple JSE environments and use the class ResourceAccessorJSE. So here is how this resource accessor works concerning its simple connection management and the resulting transaction behavior:

- As soon as you access the database for the first time, the accessor opens a database connection and binds it to the current thread. I.e. all succeeding database operations within the same thread are performed on the same database connection.
- The ResourceAccessorJSE has no connection pooling, i.e. the number of concurrently allocated database connections corresponds to the number of threads requiring access to the database. This is not a suitable model for server applications, but if you implement server components with PriDE, you will hopefully work in a JEE environment.
- The current database transaction can be committed by the call `DatabaseFactory.getDatabase().commit()`. This addresses only the connection being bound to the current thread. After committing, the connection is kept open and related to the thread so that subsequent databases operations run without warming up a new connection.
- Alternatively the commit can be initiated from any adapter resp. hybrid entity as well. This is especially of interest when working with multiple databases because the commit then refers to the database which the entity resides in whereas DatabaseFactory.getDatabase() refers to the database which was addressed at last. This will be explained in detail in chapter Multiple Databases. When you are working with a single database, these operations are equivalent.
- Instead of committing you may rollback your work by calling the rollback() method either on the database object or an adapter resp. a hybrid entity.
- The resource accessor explicitly turns off auto-commit for every allocated connection.

Coming back to the example for customer creation above, the code must be completed as follows:

```
Customer customer = new Customer();
for (int i = 0; i < 10; i++) {
    // see above
}
customer.commit();
```

As a result the code will either successfully create all 10 customers or non at all because the whole work of the loop is committed at once at the end. If any of the 10 insert operations fails with an exception, the commit call would be skipped. This causes the application to terminate without any commit which in turn causes and implicit rollback. If you write an

application which runs for a long time and is supposed to survive severe exceptions (i.e. a UI client), you should pay some attention on making the application robust against accidentally unterminated transactions. A recommended Java feature for an appropriate safety net is the UncaughtExceptionHandler interface. You can install a handler to every thread which preventively performs a rollback call.

Update

Updating a record is performed by calling the update() method of the adapter resp. the hybrid entity. All fields listed in the record descriptor's key() method call are used to identify the record, and all other fields are updated. The code

```
Customer paddy = new Customer(57);
paddy.setFirstName("Paddy");
paddy.update();
paddy.commit();
```

results in the following SQL statements as you can see from the log file:

```
select id,name,first_name from CUSTOMER where ( id = 57 )
update CUSTOMER set name = 'Fingal',first_name = 'Paddy' where id = 57
```

All update calls return the number of affected rows which should be 0 or 1 in case of an update by primary key. It's up to you if you check the result. PriDE has no detection which attributes actually changed since an entity has been loaded, so it simply updates *all* attributes which are not part of the primary key. As PriDE has no instance and change management, updates always have to be explicitly performed by the application. If you are familiar with JPA, you may recognize that the concepts are very different concerning this aspect. The chapter PriDE Design Principles explains why the much simpler approach of PriDE is not a loss.

Updating single rows by update() calls is OK for a limited number of operations per transaction. If you have to update thousands of records instead you should consider working with the class pm.pride.PreparedUpdate as explained in chapter Prepared Operations.

There are a few variants of the update() method available which allow to update multiple records at once. E.g. the method `update(WhereCondition where, String... updatefields)` can address the records of interest by a where condition. In these cases there are usually only particular fields requiring an update. The following example demonstrates how to change all first names from "Paddy" to "Patrick":

```
Customer customer = new Customer();
customer.setFirstName("Patrick");
customer.update(new WhereCondition(COL_FIRST_NAME, "Paddy"), COL_FIRST_NAME);
customer.commit();
```

The code above makes clear that multi-record updates are not necessarily best to understand when they are expressed by entity operations. Have a look on the resulting update statement which is pretty clear to understand:

```
update CUSTOMER set first_name = 'Patrick' where ( first_name = 'Paddy' )
```

If the entity layer is not appropriate, you have a lower level at hand using the class pm.pride.Database. You get access to the current database by the call `DatabaseFactory.getDatebase()`. Here is how the renaming update looks like with a combination of the Database class and the SQLExpressionBuilder:

```
Database database = DatabaseFactory.getDatabase();
String operation = SQL.build(
    "update @CUSTOMER set @first_name = 'Patrick' where ( @first_name =
'Paddy' )",
    TABLE, COL_FIRST_NAME);
database.sqlUpdate(operation);
database.commit();
```

In this example, the SQL code is almost present in its native, well recognizable form but it is still based on the entity's table and column name constants. The DRY principle and dependency tracking is kept up properly. The commit is performed on the Database instance as explained in section Transactions above.

Delete

To delete a record you have to call the delete() method of the adapter resp. the hybrid entity. All fields listed in the record descriptor's key() method call are used to identify the record. All other fields are ignored, so whether they are initialized or not is irrelevant. The following code deletes the customer with ID 57:

```
Customer c57 = new Customer();
c57.setId(57);
paddy.delete();
paddy.commit();
```

Note that the code above is based on a hybrid entity and does *not* use the re-constructor which would immediately initiate a find operation. Entities don't have to be loaded before deletion. Just like updates, every deletion returns the number of affected rows and it's up to you if you check the result.

There is a deleteByExample() method available which allows to specify a different set of key attributes and usually deletes multiple records at once. More complicated multi-record deletions can be performed by the sqlUpdate() method of the Database class similar to the example section Update.

Entity Inheritance

From a technical point of view, entity inheritance is of interest to encapsulate basic design concepts in a base class which should apply to various entity types in the same way. The chapter Find and Query already mentioned a few examples like a default public clone() method and a toString() method. Another typical reason is a stereotype set of table rows

which should be present in each table like an auto-incremented technical ID, a creation time and a last modification time, or a lock counter for concurrency control by optimistic locking.

As a simple example for inheritance, you can split up the Customer entity in a way that the ID is encapsulated in a separate entity class IdentifiedEntity which the Customer entity is derived from. This is based on the assumption that all entity classes should have a unique ID row which is a wide-spread concept.

Inheritance in PriDE is a bit inconvenient as you have to maintain more than one inheritance hierarchy. However, new tables and entities don't shoot like mushrooms out of the ground, so there's no reason to bother. Beside the entities, you also have to relate both entities' descriptors, and if you are working with separate adapters, the adapter classes have to be derived from each other too. As long as you are using 1:1 mappings, you can let PriDE's entity generator do most of the job. So here is how to generate the little inheritance hierarchy which you can find in package inheritance in the PriDE manual examples source code repository on GitHub. You start with generating the base class. As the generator is based on table structures in a database and there is no such concept like "base classes" in SQL, you must ensure that the CUSTOMER table is already present (resp. any other table following the same pattern with a technical ID) . To generate a class which does not map all the columns, you specify the columns of interest as comma-separated list along with the table name when calling the generator:

```
java
util.EntityGeneratorWithExampleConfig
CUSTOMER(ID) inherit.AbstractHybrid > AbstractHybrid.java
```

The call above uses the class EntityGeneratorWithExampleConfig mentioned at the end of the Quick Start Tutorial to simplify the passing of configuration parameters. Adding column names to a table name, tells the generator, that this is only a partial mapping resulting in an abstract class like this:

```
abstract public class AbstractHybrid extends MappedObject implements
Cloneable, java.io.Serializable {
    public static final String COL_ID = "id";

    protected static final RecordDescriptor red =
      new RecordDescriptor(AbstractHybrid.class, null, null)
            .row( COL_ID, "getId", "setId" )
            .key( COL_ID );

    public RecordDescriptor getDescriptor() { return red; }

    private int id;

    public int getId()   { return id; }
    public void setId(int id) { this.id = id; }

    // Re-constructor
    public AbstractHybrid(int id) throws SQLException {
        setId(id);
```

```
        findXE();
    }

    public AbstractHybrid() {}

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}
```

Note that there was no table name constant generated and no table name is specified in the RecordDescriptor. All this doesn't make sense for an abstract base class. The derived Customer class is then generated by specifying the base class in the generator call:

```
java
util.EntityGeneratorWithExampleConfig
CUSTOMER inherit.DerivedCustomer
-h inherit.AbstractHybrid > DerivedCustomer.java
```

It is important to know that the call requires the base class to be compiled first. The generator will determine the remaining columns to map from reading the meta data of the CUSTOMER table from the database and the mapping information from the base class *as byte code*. It is therefore mandatory to specify the base class as fully qualified name even if it resides in the same package. The result looks like this:

```
public class DerivedCustomer extends inherit.AbstractHybrid {
    public static final String TABLE = "CUSTOMER";
    public static final String COL_NAME = "name";
    public static final String COL_FIRST_NAME = "first_name";

    protected static final RecordDescriptor red =
      new RecordDescriptor(DerivedCustomer.class, TABLE,
inherit.AbstractHybrid.red)
        .row(COL_NAME, "getName", "setName")
        .row(COL_FIRST_NAME, "getFirstName", "setFirstName")
        .key( COL_ID );

    public RecordDescriptor getDescriptor() { return red; }

    private String name;
    private String firstName;

    public String getName()   { return name; }
    public String getFirstName()   { return firstName; }

    public void setName(String name) { this.name = name; }
    public void setFirstName(String firstName) { this.firstName = firstName;
}

    // Re-constructor
```

```
    public DerivedCustomer(int id) throws SQLException {
        super(id);
    }

    public DerivedCustomer() {}

}
```

Note the following details:

- The class is not abstract and is derived from AbstractEntity
- It contains only the attributes and corresponding getters and setters for the name and first name column.
- The RecordDescriptor contains only mappings for these attributes and refers to the RecordDescriptor from the base class. The complete mapping for the CUSTOMER table is assembled from both descriptors.
- The primary key is re-defined, just in case the derived class adds additional key columns over the inherited ones (which is not the case here).
- The re-constructor doesn't call the findXE() method but the super re-constructor from AbstractEntity instead which already does the find job. The base class will consider *all* the mappings because the derived class overrides the method getDescriptor().

The resulting DerivedCustomer class behaves exactly like the Customer class from the Quick Start Tutorial. You can check that by running the CustomerClient from the quick start tutorial in parallel with the equivalent DerivedCustomerClient from the package inherit. Both have the same functionality and operate on the same table but work with the two different entity representations. This reveals an important fact about PriDE's concept how inheritance is mapped to SQL where you don't find such a concept. In terms of JPA, PriDE follows the table-per-class strategy. For every non-abstract class in the hierarchy there must exist a database table with columns for *all* mapped attributes of the class itself and *all* its super classes.

Inheritance with separate adapters

When you are working with separate adapters, you need a derivation for both, entity class and adapter class. The generator calls look like that:

```
# Entity base class
java util.EntityGeneratorWithExampleConfig
CUSTOMER(id) inherit.AbstractEntity -b

# Adapter base class
java util.EntityGeneratorWithExampleConfig
CUSTOMER(id) inherit.AbstractAdapter inherit.AbstractEntity

# Derived Customer entity class
java util.EntityGeneratorWithExampleConfig
CUSTOMER inherit.DerivedCustomerEntity -b inherit.AbstractAdapter
```

```
# Derived Customer adapter class
java util.EntityGeneratorWithExampleConfig
CUSTOMER inherit.DerivedCustomerAdapter inherit.DerivedCustomerEntity
inherit.AbstractAdapter
```

An important detail is that generating the derived bean class requires to specify the base *adapter* class, not the base *entity* class in the generator call. In fact the generator needs to know about both, but the entity class can be determined from the adapter class' record descriptor. The pure base entity class however doesn't know about its mapping - that's lastly the goal of the separation ;-)

The output of these generator calls is a very straight-forward separation of the hybrid code above. There is nothing tricky to know about. You can find the outcome in the package inherit from the PriDE manual examples source code repository on GitHub.

Entity inheritance hierarchies are of course not limited in their depth. E.g. if it were a typical pattern that entities have names, try this command and have a look on the output:

```
java util.EntityGeneratorWithExampleConfig
CUSTOMER(name) inherit.AbstractNamedHybrid -h inherit.AbstractHybrid
```

What PriDE does not support are queries based on abstract base entities which automatically consider the tables of the derived non-abstract entities. You can find features like that in JPA, but they require a highly complicated, obscure SQL query assembly - in combination with the table-per-class strategy resulting in SQL union expressions. This is something, which doesn't happen too often and should always remain in the developer's responsibility to stay on control of your SQL.

SQL Expression Builder

If you walked through the preceding chapters of this manual, you already came across some simple examples for building SQL expressions with PriDE's expression builder. As this helpful little utility will be used more intensively in the following chapters, it is worth to understand its idea. It does not really depend on SQL but can be used for any string assembly where things become too confusing when making use of Java's built-in capabilities like string concatenation, StringBuilder oder String.format(). Actually it is just a small extension of String.format().

Elaborated SQL vs. Java

Let's take up the good old CUSTOMER table from the Quick Start Tutorial and let's suppose you want to implement a batch application querying for suspicious new customer registrations, which the system will initially block from order placement until the customers have verified their identity (somehow). We are looking for customers in a certain ID range with

- The summarized length of name and first name is less than 7 letters or
- The name consist only of a single letter or
- Name and first name are identical

SQL is a very powerful and highly expressive and compact language for things like that. The appropriate where-clause would look like that, where only the boundaries of the ID range differ from one call to the next:

```
id between <lowest> and <highest> and (
    ( length(name) + length(first_name) < 7 ) or
    ( length(name) < 2 ) or
    ( name = first_name )
)
```

No matter which assembly API your Java persistence manager provides - JPA's criteria API, PriDE's WhereCondition, or JOOQ's DSL API - it will cost a lot more Java code than SQL code to assemble the expression, and it will become hard to tell from the Java code what the resulting SQL may look like. So the recommendation is: for the sake of SQL maintainability, integrate the SQL code in your Java code *as is*. Of course, Java won't accept SQL syntax, so "integration as is" means integration as a String after having verified syntactical correctness in a suitable SQL tool. Unfortunately this would raise another maintenance problem: the String literal is a big, big magic number composite and conflicts with the DRY principle. Although it contains various column name references you won't be able to safely detect that the query might be affected e.g. when the NAME column requires a size change. You hopefully don't try a full text search for the term "name" ;-)

As a first step towards a solution, PriDE's entity generator generates constants for table and column names, and it is strongly recommended to use these columns for SQL expression assembly. However concatenating String fragments and constants won't result in better readability:

```
COL_ID + " between " + lowest + " and " + highest + " and ( " +
"( length( + COL_NAME + ") + length( " + COL_FIRST_NAME + ") < 7 ) or" +
...
```

String.format() is designed to keep the structure of the result string recognizable, but in this case it won't help too much:

```
String.format(
"%s between %d and %d and (" +
"    ( length(%s) + length(%s) < 7 ) or" +
"    ( length(%4$s) < 2 ) or" +
"    ( %4$s = %5$s )" +
")",
COL_ID, lowest, highest, COL_NAME, COL_FIRST_NAME);
```

Elaborated SQL with SQLExpressionBuilder

PriDE's expression builder extends String.format() in a way, the you can use identifiers rather than just % and position numbers as variables. The builder is address by the static function `build(String formatString, Object... args)` in class pm.pride.SQL. Identifiers in the format string that require replacement by any of the following arguments begin with an @ character and end with the first character that is neither a letter nor an underscore. Based on that, the SQL can be represented almost natively:

```
SQL.build(
"@id between %d and %d and (" +
"    ( length(@name) + length(@first_name) < 7 ) or" +
"    ( length(@name) < 2 ) or" +
"    ( @name = @first_name )" +
")",
COL_ID, lowest, highest, COL_NAME, COL_FIRST_NAME);
```

As you can see, the identifier feature can be combined with Java's standard replacement feature addressed by % characters. Arguments are assigned to identifiers in order of occurrence in the format string. Repeated occurrences of an identifier are replaced by the argument which was assigned to the identifier on its first occurrence.

By default, the identifiers and the assigned argument values don't have to be identical, so the identifiers may be abbreviations or - vice versa - more descriptive forms of the actual table or column names passed as arguments. The possible risk is a hidden miss-assignment which still leads to syntactically valid SQL but to a wrong business logic. Referring to the example, swap the constants COL_NAME and COL_FIRST_NAME in the argument list and it results only in a minimal subtle miss behavior. If you don't have fine-grained test suite to reveal such a bug, you may use the expression builder in a more restrictive way. If you call SQL.buildx() instead of SQL.build() the builder will throw an InvalidArgumentException if the variable identifiers don't match the values of the assigned arguments based on a case-insensitive string comparison. E.g. the following SQL assembly would fail as the argument value "name" would be assigned to the variable identifier "first_name":

```
SQL.buildx("@first_name is null", "name")
```

This variant implies that you use the % notation where name conformity doesn't make sense, e.g. for a column *value* instead of a column *name* like the ID range boundary values in the examples above. Additional validation options are available when you use the class SQLExpressionBuilder and its constructors directly. They allow to specify if the identifier comparison should be performed case sensitive or case insensitive and if the builder should actually throw an exception in case of miss-matches or just print out a warning on Stderr. Furthermore you may change the validation behavior of SQL.build() by setting SQLExpressionBuilder's static member `validationDefault`.

If you need lots of arguments, it is helpful to split the argument list in multiple lines like the format string. Each argument line contains only the arguments which are (first) assigned to the identifiers of the corresponding line from the format string. Applied to the example above it looks like that:

```
SQL.build(
"@id between %d and %d and (" +
"    ( length(@name) + length(@first_name) < 7 ) or" +
"    ( length(@name) < 2 ) or" +
"    ( @name = @first_name )" +
")",
COL_ID, lowest, highest,
COL_NAME, COL_FIRST_NAME);
```

A small expression as the one above doesn't need those tricks, but e.g. a complex SQL merge statement may require 20 arguments and more. Alternatively you may combine identifiers with position numbers as known from String.format(), so that you can check the identifier/argument matching by counting:

```
SQL.build(
"@1$id between %2$d and %3$d and (" +
"    ( length(@4$name) + length(@5$first_name) < 7 ) or" +
"    ( length(@name) < 2 ) or" +
"    ( @name = @first_name )" +
")",
COL_ID, lowest, highest, COL_NAME, COL_FIRST_NAME);
```

Only one occurrence of an identifier needs to be accompanied by a position specification, while all the others automatically inherit the argument assignment. You may add the position number to all occurrences but then they have to be identical. Re-positioning is not allowed.

Building and Formatting

All identifiers are replaced by the string representation of its assigned argument. The expression builder is *not* concerned with SQL value formatting. If you do not only pass column, table, and alias names as arguments but also *values*, you must ensure proper SQL formatting. There are different approaches to achieve that.

- Add formatting characters to the SQL string. This works well for simple data types like string and integer values, but you must be aware of the SQL injection risk if the values come from an untrustworthy source like a consumer website.
- Format the value before passing it to the argument list. The value formatter for the current database is available through `DatabaseFactory.getDatabase().formatValue(Object value)`. The result is a fully formatted SQL value string. E.g. passing a string HELLO results in an SQL formatted String 'HELLO'.
- Just place a ? character in the SQL string and pass the value argument to the function that consumes the SQL rather than the builder. You can see an example at the end of chapter Find and Query in section Arbitrary Criteria.

Joins

PriDE provides different techniques to express joins, depending on the purpose resp. the type of outcome of the join:

- An entity being a 1:1 mapping of a table because the join is only required for complicated selection conditions that take related tables into account.
- An extended entity consisting of a 1:1 mapping of a table, extended by a few attributes from a related table.
- An entity composition, i.e. an entity for a 1:1 mapping of a table, extended by references to related entities that also represent 1:1 table mappings.

- A composition from table fragments and/or computations, making up a new type of entity with its own specific meaning.

The different variants are explained with the CUSTOMER table being used in all other examples so far, and an additional table ADDRESS like this:

```
create table ADDRESS (
    customer_id integer not null,
    street varchar(30),
    city varchar(30)
);
```

Each customer optionally has an address attached and the column customer_id is the foreign key to reference a customer from an address. You find a corresponding Address entity and a table creation class in the package joins of PriDE's manual examples source code repository on GitHub.

Bringing customers and addresses together in a query requires a join which may look like that to express an inner join:

```
select ... from
CUSTOMER cst
join ADDRESS addr
on addr.customer_id = cst.id
```

Joins are combinations of tables and therefore require a descriptor like tables do. Although simple join cases can be expressed with the RecordDescriptor class explained in chapter Entity, Adapter, and Descriptor, you will usually work with the derived class `pm.pride.JoinRecordDescriptor`. All following examples make use of that class.

Joining Table Fragments

Just because it gives the best view on how the join structure appears in PriDE code, let's start with the most open variant: pulling fragments from different tables together, making up a new entity type. This is of interest when selecting huge amounts of records or when the result does not actually represent a primary entity like a customer or an address but is a kind of chimera. E.g. you may need a customer ID pair plus some address data in a query for duplicates, retrieving customers with same names and same addresses. To keep things simple for the beginning, this section's example selects the ID and name from the CUSTOMER table and the city from the associated address. The pure entity part of this chimera looks like this:

```
public class CustomerNameAndCity {
    private int id;
    private String name;
    private String city;

    // Standard getters and setters as usual
    // ...
}
```

The entity generator cannot produce combined types, so you must assemble it by your own. However, the mechanical work is not really complicated and mature join conditions should remain in the developer's responsibility anyway. You can make a hybrid entity from the type above by adding only a few details:

```
public class CustomerNameAndCity extends MappedObject {
    // attributes, getters, and setters like above

    protected static final RecordDescriptor red;

    public RecordDescriptor getDescriptor() { return red; }
}
```

You have seen this structure in many examples of this manual before. You may of course separate entity and adapter class as usual, but learning is easier with hybrid types. The essential detail is the assembly of the record descriptor. Using PriDE's JoinRecordDescriptor type, the join is assembled as follows

```
protected static final RecordDescriptor red =
    new JoinRecordDescriptor(CustomerNameAndCity.class,
"CUSTOMER", "cst")
.join("ADDRESS", "addr",
"addr.customer_id = cst.id")
```

The funny line indention above is just there to point out that all the parts of an SQL join are plain to see in the core of the descriptor definition. Compare that to the join at the beginning of this chapter. What needs to be added is the mapping of table columns to attributes. This is accomplished by the row() method which you already know from earlier descriptor examples. You have to add them after each part describing a table. So for the name and city example the complete descriptor looks like that:

```
protected static final RecordDescriptor red =
  new JoinRecordDescriptor
    (CustomerNameAndCity.class, "CUSTOMER", "cst")
          .row("id", "getId", "setId")
          .row("name", "getName", "setName")
      .join("ADDRESS", "addr", "addr.customer_id = cst.id")
          .row("city", "getCity", "setCity");
```

Of course it is strongly recommended to substitute the string literal by references to appropriate constants. Usually you should have the (generated) primary entity types for CUSTOMER and ADDRESS available, including constants for tables and columns. Constants for the alias names "cst" and "addr" must be added by the developer, and the join condition should be assembles with the SQL expression builder. For the example above the clean-up results in something like that:

```
public static final String CUSTOMER_ALIAS = "cst";
public static final String ADDRESS_ALIAS = "addr";
public static final String CUSTOMER_ADDRESS_JOIN_CONDITION =
  SQL.build("@addr.@customer_id = @cst.@id",
    ADDRESS_ALIAS, Address.COL_CUSTOMER_ID,
```

```
    CUSTOMER_ALIAS, Customer.COL_ID);

protected static final RecordDescriptor red =
  new JoinRecordDescriptor(
    CustomerNameAndCity.class,
    Customer.TABLE,
    CUSTOMER_ALIAS)
        .row(Customer.COL_ID, "getId", "setId")
        .row(Customer.COL_NAME, "getName", "setName")
    .join(
      Address.TABLE,
      ADDRESS_ALIAS,
      CUSTOMER_ADDRESS_JOIN_CONDITION)
        .row(Address.COL_CITY, "getCity", "setCity");
```

At the first sight this may look less clear than the example based on string literals. However, the price you pay for the constant-based form is usually worth it when your application grows. Joins are a very powerful concept from SQL but may violate module boundaries in a vertically well-structured Java architecture. if you allow the violation for powerful joins you should at least make the module dependencies trackable by using constants across the boundaries. If you change something in any of the tables and their corresponding primary entity mappings, the depending join descriptors and joined entity types should automatically change as well or lead the developer to broken code through compile time errors. Hidden dependencies by intense use of magic numbers compromises refactoring of the application code, and sophisticated persistence operations should not be an excuse.

If you want to perform a left outer rather than an inner join, you have to exchange the call of method join() by a call of leftJoin(). A typical problem when designing outer join types is the fact that all data from the joined tables is *optional* data. The corresponding attributes which this data is mapped to, must therefore accept null values. I.e. primitive attributes types like `int` or `long` are not suitable.

The complete hybrid entity type CustomerNameAndCity (based on constants) can be found in PriDE's manual examples source code repository on GitHub.

Joining Entities with Fragments

Another typical join variant is to extend the complete content of a record from one table by fragments from other associated tables. Applied to the customer/address example, something like that for a full customer and the city from the associated address:

```
select cst.*, addr.city from
CUSTOMER cst
join ADDRESS addr
on addr.customer_id = cst.id
```

The term "extend" already leads to the technical solution: create a new type derived from the Customer class which contains the attribute *city* and appropriate mappings. You can use a different constructor for JoinRecordDescriptor here which refers to the base class' descriptor to minimize the additional descriptive work to do.

```
public class CustomerWithCity extends Customer {

  protected static final RecordDescriptor red =
    new JoinRecordDescriptor(
        CustomerWithCity.class,
        Customer.red,
        CUSTOMER_ALIAS)
      .join(
        Address.TABLE,
        ADDRESS_ALIAS,
        CUSTOMER_ADDRESS_JOIN_CONDITION)
      .row(Address.COL_CITY, "getCity", "setCity");

    public RecordDescriptor getDescriptor() { return red; }

    private String city;

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }
}
```

The complete class CustomerWithCity can be found in PriDE's manual examples source code repository on GitHub. It uses the same join condition and table aliases as the example from section Joining Table Fragments. Writing queries for such a class may require to put the table alias in front of column names if the same name appears in more than one of the joined tables. As long as the names are unique, you can work with plain column names. E.g. customers living in London can be found by

```
CustomerWithCity cwc = new CustomerWithCity();
cwc.setCity("London");
cwc.queryByExample(COL_CITY);
```

If the extended type inherits query methods from the base class, you can still use all methods that don't refer to columns which became ambiguous by the extension. E.g. CustomerWithCity can be equipped by a re-constructor that delegates to the Customer class' re-constructor:

```
public CustomerWithCity(int id) throws SQLException {
    super(id);
}
```

Entity Composition

A very simple join case is a complete composition of existing entities. This variant requires to derive a composite type from one entity type with members for the associated entity types. For a customer and its address, such a join may look like this:

```
public class CustomerWithAddress extends Customer {
    Address address;

    public Address getAddress() { return address; }
```

```
    public void setAddress(Address a) { this.address = a; }

    public static RecordDescriptor red =
      new JoinRecordDescriptor(Customer.red, CUSTOMER_ALIAS)
        .join(Address.red, ADDRESS_ALIAS,
            "address", CUSTOMER_ADDRESS_JOIN_CONDITION);

    public RecordDescriptor getDescriptor() { return red; }
}
```

The example above makes use of a join() method that gets passed the descriptor for the contained entity and the name of the member. PriDE will not access the member directly but use the appropriate getter and setter method. Why using the setter? Well, here is the only exception to the rule that PriDE doesn't create entities. If a contained entity is null when being required to receive data from the database, PriDE will create an instance and associate it to the composite type by calling the appropriate setter. The contained entity type must provide a constructor without parameters or a constructor getting passed the containing entity.

As mentioned earlier, you may change from inner to left outer join by calling leftJoin() rather than join() in the descriptor assembly. When iterating through the results, PriDE will set the member to null for every record which has no joined record associated and re-create the member when needed.

Ad-hoc Joins

Sometimes a join is only needed to express query conditions which span multiple related tables but the results are entities of a type which is already present. For these cases, it is not desirable to create a new entity type just to provide a place for the join descriptor. As an alternative a record descriptor may be defined where ever an when ever and passed to special query methods provided by every hybrid entity and every adapter class.

Let's come back to the example of retrieving customers living in London but without actually selecting any address data. The following code snippet demonstrates an example:

```
Customer c = new Customer();

RecordDescriptor customerJoinedWithAddress =
  new JoinRecordDescriptor(c.getDescriptor(), CUSTOMER_ALIAS)
    .join(Address.TABLE, ADDRESS_ALIAS,
          CUSTOMER_ADDRESS_JOIN_CONDITION);

WhereCondition onlyLondon =
  new WhereCondition().and("city", "London");

c.joinQuery(customerJoinedWithAddress, onlyLondon);
```

The method joinQuery() accepts every record descriptor which is compatible with the entity's own descriptor in the sense that it maps to the same entity type.

You have seen a lot of different ways now to express table joins in PriDE. Finally it is important to mentioned that the class JoinRecordDescriptor is not restricted in the number of tables to join. You may chain the calls of join() and leftJoin() as ofter as needed. So happy joining :-)

Optimistic/Pessimistic Locking

Locking for concurrency control is not directly addressed by PriDE but can be achieved by simple patterns. Usually, the object locking strategy of an application is a more general design decision and nothing you decide individually for every single table / entity. The examples in the section therefore demonstrate pattern which can be encapsulated in base classes and need to be implemented only once.

This manual will not go into details about what kind of locking to prefer over the other for which kind of business requirements. There lots of general introductions on concurrency control on the Internet, e.g. on Wikipedia.

Optimistic Locking

Optimistic Locking is a typical concept for the management of concurrent update access from multiple applications on the same record in a database. An update of an existing record is only performed if it has not been modified by someone else since the current application has read the record of interest from the DB the last time. If it was modified, the caller is informed about a concurrent access conflict. The concept requires a version counter in the table. Every update operation increments this version counter and performs the actual update only if the entity's version counter value in memory is still the same as in the database.

Optimistic locking only makes sense for single-record updates by primary key, so the essential aspect of the pattern is to override the update() method of an adapter class resp. a hybrid entity.

The CUSTOMER table being used in almost all the manual examples, can be equipped for optimistic locking, by adding an appropriate counter:

```
create table LOCKABLECUSTOMER (
    id integer not null primary key,
    name varchar(20),
    first_name varchar(30),
    version integer not null
);
```

The resulting entity class gets an appropriate attribute int version; and the entity's update() method must be overridden like that:

```
@Override
public int update() throws SQLException {
    version++;
    int numRows = update(where().and(COL_VERSION, version-1));
```

```
    if (numRows == 0) {
        version--;
        throw new SQLException("optimistic lock error");
    }
    return numRows;
}
```

The update() method without parameters is the update by primary primary key, and the override above works as follows:

- The version number is incremented by 1 in the entity before actually updating the DB. So if the update goes through, the version number will also be incremented in the database.
- The where() call without parameters assembles a WhereCondition from the entity's primary key attributes.
- The appended and() call extends the condition by a constraint that - beside the primary key - also the version number in the DB must match the former value.
- The method then calls an update() method with the extended WhereCondition and checks the number of affected rows.
- If the version number was already incremented in the database, the number of affected rows is 0 and the method reports an error. The version number is restored, just in case the application can handle the exception and works on with the entity. If your application is actually this robust, you should consider defining your own optimistic lock exception type to make this case easy to distinguish from other DB problems.

The code may be well encapsulated in a base class for all entity types which require optimistic locking. You can find an appropriate base class and derived OptimisticCustomer class in package locks of the PriDE manual source code repository on GitHub.

Pessimistic Locking

In SQL databases, pessimistic locking is usually achieved by selecting-for-update operations. I.e. instead of overriding the update() method, pessimistic locking requires to override the find() method without parameters. Like the update() method without parameters, find() addresses the entity's primary key and therefore is a single-record operation. As a difference to optimistic locking, it does not require any additional columns to organize the locking, However, it requires the database to actually support select-for-update which is sometimes not the case for server-less databases. SQLite e.g. doesn't support select-for-update as any manipulative DB operates always locks the whole database.

The following override will do the job:

```
@Override
public boolean find() throws SQLException {
    return find(where().forUpdate());
}
```

As you already from the optimistic locking example, the method where() assembles a selection criterion based on the entity's primary key attributes. The appended call of forUpdate() adds the required "… FOR UPDATE" to the constraint which then is used to call the find() method accepting a WhereCondition.

Keep in mind that there are a few more find() methods which you may have to override. Method findXE() is based on find(), so it doesn't need an extra override, but findRC() e.g. has its own implementation. The overrides can be well encapsulated in a base class for all entity types which require pessimistic locking. You can find an appropriate base class and derived PessimisticCustomer class in package locks of the PriDE manual source code repository on GitHub.

JSE, JEE, and ResourceAccessor

Resources accessors are the link between PriDE and JDBC, Java's foundation API for accessing SQL databases. Its main job is to provide JDBC connections for the database that PriDE is supposed to operate on. The accessor must be instantiated before any persistence operation is performed in the code. So usually accessor instanciation takes place somewhere in the application's bootstrap. In an enterprise application which properly encapsulates database access in repository components resp. data access objects, the accessor may also be lazy-initialized in these components' lifecycle methods.

PriDE provides two standard implementations, one for JSE and one for JEE environments, which are both derived from the base class `pm.pride.AbstractResourceAccessor`.

JSE

The class `pm.pride.ResourceAccessorJSE` is used in all examples throughout this manual. It is based on JDBC's driver manager interface to establish a connection to a database which requires the following information:

- A JDBC driver class, provided by a third-party library on the class path
- Usually the name and password of a database user
- A database URL

Driver class and user/password are passed to the constructor of the resource accessor. As there are a few more optional configuration parameters available, the parameters are passed as a java.util.Properties object to keep the constructor signature simple. How the application assembles these properties is up to you. In the quick start tutorial you already learned about two possible variants - system properties and property files.

The database URL is not part of the resource accessor but is used to register the particular database and its resource accessor in a *database context* in PriDE's database factory. So in fact you may access multiple databases of the same type by one resource accessor as you will see in chapter Multiple Databases. The following code snippet demonstrates a minimal bootstrap for a JSE application with a single SQLite database.

```
Properties props = new Properties();
props.setProperty(
    ResourceAccessor.Config.DBTYPE, "sqlite");
props.setProperty(
    ResourceAccessor.Config.DRIVER, "org.sqlite.JDBC");

ResourceAccessor re = new ResourceAccessorJSE(props);

DatabaseFactory.setDatabaseName(
    "jdbc:sqlite:pride.examples.db");

DatabaseFactory.setResourceAccessor(re);
```

Databases with default support in PriDE are listed in the Interface ResourceAccessor.DBType, so instead of the string literal "sqlite" you should rather use ResourceAccessor.DBType.SQLITE. There are some database types listet which PriDE is known to work on but which are not continuously tested. The permanently tested ones are visible on PriDE's continuous integration page at Travis CI.

JEE

The class `pm.pride.ResourceAccessorJEE` is suitable for enterprise environments like JEE application servers. It is based on a JNDI lookup of data sources. The database name is not used to specify a database URL but its JNDI lookup name. Driver class, user name, and password are not required, so the minimal bootstrap looks like that:

```
Properties props = new Properties();
props.setProperty(
    ResourceAccessor.Config.DBTYPE,
    ResourceAccessor.DBType.ORACLE);
ResourceAccessor re = new ResourceAccessorJEE(props);
DatabaseFactory.setDatabaseName("java:global/myapp/mydb");
DatabaseFactory.setResourceAccessor(re);
```

You may consider placing the database type into the JNDI context as well to make the application's bootstrap code completely independent from that issue. Maybe you want to use an HSQL database in test environments which can be killed and re-initialized within seconds while the productive environment is based on an Oracle database.

Accessor Configuration

Beside the basic things like database name and user name there are some more configurations parameters available for the pre-defined resource accessors classes provided with the PriDE distribution. Most of them are concerned with two other aspects which resource accessors are also responsible for: logging and SQL syntax. Every resource accessor has to implement the interface `pm.pride.SQL.Formatter` which is used by PriDE to produce well-formed SQL value and operator representations. E.g. the method formatValue() called with a string argument is supposed to put single quotes around the string and escape any single quotes and other SQL key characters within the string. If you need to implement your own special resource accessor for some reason, you should usually

derive it from class AbstractResourceAccessor which provides reasonable default functionality for these formating issues.

The configuration parameters available for the default implementations are listed in interface ResourceAccessor.Config. The details of all options are documented by their Javadocs. Here is only an excerpt of options which help to understand general aspects:

- `pride.logfile = sql.log`

  If a log file is configured, PriDE will log every database operation to the file. In the quick start tutorial you already learned how the log entries look like. The log file will be re-written with every application start and after exceeding the maximum length of 100 kilobyte. You may change the maximum file size by parameter `pride.logmax`. If you prefer a different logging mechanism. e.g. logging by log4j, you have to provide a record descriptor with an alternative implementation of the methods sqlLog() and sqlLogError().

- `pride.bindvars = on|off`

  Specifies if PriDE is supposed to use bind variables by default. Without this configuration parameter, PriDE talks plain SQL if not explicitly coded differently by the application. Prepared operations always use bind variables, and the WhereCondition class (see chapter Find and Query) provides the method bindvarsOn() to overrule the default. If you are working with binary large objects attributes (BLOBs), you must switch bind variables on as there is no plain SQL representation for these attributes.

- `pride.systime = 1000230`

  Specifies a UNIX milliseconds time value which is used to represent the current database server time. Where ever the application uses this value in a database operation it is replaced by an appropriate server expression to address the current time. E.g. in Oracle databases this is SYSDATE, in SQLite it is a call of the strftime() server function. The application can access this special value for the current database by calling SQL.systime(). The default value is January 1. of year 0 which should hopefully not clash with any "real" date value which the application's business logic works with. So usually you don't have to change the value :-)

Multiple Databases

You may access multiple databases in an application, using PriDE's concept of a *database context*. As long as you don't explicitly address particular contexts, you are implicitly working with a default context. If you recall the chapter about resource accessors and the the application bootstrap, there appeared two calls which initialized the default context:

```
DatabaseFactory.setDatabaseName(...)
DatabaseFactory.setResourceAccessor(...);
```

If you want to work with multiple databases, you have to initialize multiple contexts in the bootstrap in the same manner. The contexts are addressed by name and you perform a

context switch by calling the DatabaseFactory's static method setContext(). Switching to a context which doesn't yet exist, causes the context to be created.

```
// Initialize the default database context
DatabaseFactory.setDatabaseName(...)
DatabaseFactory.setResourceAccessor(...);

// Create a new context by switching
DatabaseFactory.setContext("other-db");

// Initialize the new context
DatabaseFactory.setDatabaseName(...)
DatabaseFactory.setResourceAccessor(...);

// Switch back to the default context
DatabaseFactory.setContext(DatabaseFactory.DEFAULT_CONTEXT);
```

Alternatively you can add a context without switching back and forth by calling method addContext().

There are two ways how to work with these contexts:

- By explicit context switching as demonstrated above which will cause subsequent database operations to operate on the current context. As context switching is a global configuration change, this is only suitable for applications without concurrent access by multiple users, e.g. fat clients or batch applications. You should never use that in a server application.

- By associating entity types to different contexts. This is achieved by a call of the context() method when constructing the appropriate RecordDescriptor

  ```
  new RecordDescriptor(...)
      .context("other-db");
  ```

  All database operations that are based on that record descriptor will operate on the associated context. A global context switch is not required for that, so this concept will also work in server applications. Usually this is also the more convenient variant. However, if you make use of this, make sure that your application either doesn't call context-dependent static methods of the DatabaseFactory class or that it is using them safely. Earlier chapters of this manual introduced the method call DatabaseFactory.getDatabase() to get access to a lower operation level. This call returns a Database object form the *current context* which of course may be the wrong one if you don't switch it. In a multi-database server application you should rather call the alternative getDatabase(String) method which gets passed a context name.

Multi-Database Transactions

Although SQL database usually provide very good transaction safety for their own, the transaction management across *multiple* independent databases is a separate challenge. If your are working in a JEE environment, the application server should provide a safe

transaction coordination for the involved databases, e.g. a 2 phase commit protocol. In a JSE environment you either have to integration a transaction manager like Atomikos or follow design patterns that minimize the risk of data inconsistencies.

A recommended design pattern is the so-called *best efforts 1 phase commit*. If you have to perform modifications in multiple databases within one transaction then do the actual work in all involved databases first and at the very end, run the commit calls for all databases. I.e. inconsistencies can only occur if any of the commit calls should fail which is a very rare situation. The pattern is suitable not only for databases but can be applied to any combined usage of transactional resources. In that case you should start the sequence of commit calls with the resource with the highest failure risk. E.g. you may assume that committing a JMS queue or a Kafka topic has a higher failure risk than committing a transaction on an Oracle database server based on decades of experience and hardening.

Prepared Operations

PriDE is designed to keep as much persistence logic as possible in Java code instead of writing stored procedures that reside in the database. This has several reasons:

- Stored procedures must be written in a highly vendor-specific language like Oracle's PL/SQL, so the code is not compatible between different database systems. On the other hand, Java and PriDE-based persistence logic is widely compatible, allowing things like unit testing on a light-weight HSQL while working with an Oracle DB in production.
- As stored procedures are written in a different language that forces you to violate the DRY principle which is strongly emphasized by PriDE as one of the most valuable design principles for long-term code maintainability. E.g the stored procedure's code can't refer to constants defined in Java. They will often break at runtime not at compile time, and refactoring becomes a fragile job.
- You need an additional development environment and appropriate knowledge which can make things very complicated.

Although there may still be good reasons to write stored procedures, you should at least know about PriDE's capabilities to efficiently run database mass operations within Java. Some of the mass operation features have already been mentioned in earlier chapters:

- Query results are by default provided by ResultIterators, allowing to iterate through large amounts of results using a single entity. You can process millions of records without being afraid of memory shortage. See chapter Find and Query for details.
- Methods spoolToArray() and spoolToList() of class ResultIterator allow to run multiple co-ordinates selects as an alternative to Joins when accessing data along 1:N relationships. This allows to minimize the actual data being transfered from the database to your application.
- Updates and deletes may get passed key field lists and WhereConditions to address multiple records with a single operation. See chapter Insert, Update, and Delete for details.

- The class pm.pride.Database provides low-level API functions to assemble multi-record operations of arbitrary SQL structure. An example based on method sqlUpdate() can be found in chapter Insert, Update, and Delete. Additional methods are sqlQuery() and sqlExecute().

Prepared operations are a thin convenience layer combining JDBC's concept of prepared statements with PriDE's concept of O/R mapping based on record descriptors. PriDE provides the following prepared operation types:

- pm.pride.PreparedInsert
- pm.pride.PreparedUpdate
- pm.pride.PreparedSelect

When you turn on the usage of bind variables (see configuration parameter pride.bindvars in the quick start tutorial) PriDE will also use these classes internally to run database operations based on prepared statements. However, using the classes explicitly in your application code allows to use the same prepared statements for many operations without reinitializing it with every call. You can also use them in batch mode which allows to run thousands of operations per second.

Consider the CUSTOMER table and the appropriate Customer entity class which is used all over this manual. If you need to insert thousands of customers in one turn, you should use the PreparedInsert class in a way like that:

```
Customer c = new Customer();
PreparedInsert insert = new PreparedInsert(c.getDescriptor());
for (int i = 0; i < 100000; i++) {
    c.setId(i);
    c.setFirstName("Paddy-" + i);
    c.setName("Fingal");
    insert.execute(c);
}
c.commit();
```

Running this example on an SQLite databases takes less than a second to insert 100.000 customers. On a local Oracle XE with a usual 4 core laptop it takes about 30 seconds, i.e. more than 3.000 inserts per second. But you can even speed up Oracle to SQLite's performance by using batched operations instead:

```
Customer c = new Customer();
PreparedInsert insert = new PreparedInsert(c.getDescriptor());
for (int i = 0; i < 100000; i++) {
    c.setId(i);
    c.setFirstName("Paddy-" + i);
    c.setName("Fingal");
    insert.addBatch(c);
}
insert.executeBatch();
c.commit();
```

With 100 thousand inserts per seconds, PriDE is a lot faster than all other O/R mapping tools for Java, even if you consider maximum tuning options. And you may do a lot of things with this performance before you have to consider using stored procedures. A key aspect for high performance is the network latency between the Java application and the database server. Mass operations are usually performed by batch programs rather than interactive clients or web sites. You should therefore keep the network distance between the Java batch programs and the database as short as possible. Best performance is achieved by co-locating both on the same server machine using the so-called loopback device and localhost addresses.

When using prepared operations, you should keep in mind that they have an open JDBC prepared statement inside. So that you must not forget to close the operations when the job is done. Prepared operations implement the AutoCloseable interface, so you may use try-with-resources:

```
Customer c = new Customer();
try (PreparedInsert insert = new PreparedInsert(c.getDescriptor())) {
    //... do the job ...
}
c.commit();
```

An examples for a mass insertion can by found in the class MassInsertClient in the package mass of the PriDE manual source code repository on GitHub.

As you can see from the example, you don't necessarily need a new entity for each call of execute() or addBatch(). So as you already know from the ResultIterator class the whole iteration may operate on a single entity. Tying both sides together, you can design fast and memory efficient ETL procedures in Java by reading records from a result iterator into a single entity, transforming it in place, and directly passing the result to a prepared operation.

The class PreparedUpdate allows to run mass updates instead of insertions. By default, the class uses the definitions from the record descriptor passed in the constructor to tell which fields make up the key to identify a record and which other fields need to by updated. However, the class provides a few alternative constructors to define key fields and update fields separately.

The class PreparedSelect is probably not useful for application code. It is used internally by PriDE.