

# The jSyncManager:

## jConduit Development by Example

*Version 0.81*

*Written by Brad BARCLAY <bbarclay@jsyncmanager.org>.*

### Introduction

With the recent Open Source release of The jSyncManager (<http://www.jsyncmanager.org>), organizations now have a platform-neutral Java-based tool with which to manage and synchronize data between traditional applications and databases, and handheld computers based on the Palm Computing platform. The jSyncManager's object-oriented jConduit APIs make it easy to create plug-in modules which can manage the synchronization of a limitless variety of data sources, including applications, database management systems, and network-based data sources.

This guide will discuss the process of developing your own Java-based jConduits to link The jSyncManager to your own handheld and traditional data sources. It is intended for those who need platform independent data synchronization capabilities, and who have experience in developing software classes in Sun Microsystem's Java language.

**Note:** it is worth being aware that there are two different ways in which conduits can be used in The jSyncManager. The first is through the The jSyncManager GUI client, which typically runs on an end-user workstation and synchronizes one handheld at a time. The second way in which they can be used is with the jSyncManager MultiPort Server, where dozens -- and potentially hundreds -- of simultaneous synchronization sessions can be run at once. Both types of conduits share the same development APIs and classes, and are thus interchangeable, however conduits intended for use with the MultiPort server need to adhere to strict thread-safety and data-locking issues to prevent incorrect or undesirable results when two or more copies of a jConduit are active at once. From time to time, notes such as this one will appear in this document to mention strategies that can be used for developing such conduits.

You will probably find it extremely helpful to have available along with this document a copy of the The jSyncManager JavaDoc Class documentation. An archive of this documentation is available from the The jSyncManager website.

## What is a jConduit?

A jConduit is a piece of plug-in code that defines the specific tasks that should be taken to synchronize data between one or more handheld databases, with one or more external data sources (applications, databases, network accessible data, etc.). jConduits can perform a one-way synchronization (i.e.: they can read data from the handheld and write it to a host, or take data from a host and write it to the handheld), however it's much more common to perform a two-way data synchronization (where data from both the handheld and host can be interchanged in both directions).

The jSyncManager provides a number of facilities to jConduit developers. Its primary duties include:

**The Protocol Stack:** The jSyncManager includes a robust, all-Java implementation of the communications protocols required to send and retrieve data to and from the handheld system, freeing developers from having to write their own,

**High-Level jConduit Interfaces:** The jSyncManager also includes a number of high-level interfaces for performing synchronization tasks, such as opening and closing databases, reading and writing records, determining what records need synchronization, and general data management tasks for the handheld device,

**Common Data Abstraction Classes:** The jSyncManager also provides a number of classes for abstracting both generic and specific handheld data types, such as records and resources. Classes for abstracting records from the six standard handheld databases (AddressBook, DateBook, ToDo List, Mail, MemoPad and Expense) as typical Java data objects are provided, freeing developers needing access to these standard record types from needing to know the details of the handheld storage of these data types,

**jConduit Management:** The jSyncManager also handles all of the necessary tasks to manage installed jConduits. It provides facilities to allow multiple jConduit plug-ins to be installed simultaneously, and at synchronization times it takes care of the task of scheduling each in turn to run their synchronization tasks,

**Graphical User Interface:** along with all of the above tasks, The jSyncManager also provides all of the necessary facilities to present a Graphical User Interface to the user for device and jConduit management. jConduits can take advantage of the GUI services provided by The

jSyncManager to present jConduit specific configuration options to the end-user prior to synchronization,

**Data Persistence:** Lastly, The jSyncManager provides data persistence for jConduits, without any special intervention required on the part of the jConduit developer. jConduits are serialized and deserialized at run and exit time to provide field persistence for things such as configuration settings, without requiring the jConduit developer to code their own setting storage solutions.

jConduits must individually take care of the task of opening and closing the required handheld and host databases, determining what data needs to be synchronized, and managing both the order of synchronization, and how to resolve conflicts between data on the host, and data on the handheld. In general, these tasks are highly application-specific: it is normally left up to the developer to determine the best implementation strategies for their jConduit, by either developing their conduit to use the best possible strategy, or to provide multiple strategies and allow users to select between them.

### **Getting Started: the AbstractConduit Class**

The jSyncManager takes a highly object-oriented approach to jConduit development. To this end, each and every jConduit developed for The jSyncManager is a descendent of the abstract class *org.jSyncManager.API.Conduit.AbstractConduit*. This class defines all of the interfaces a jConduit must implement in order to interface with The jSyncManager synchronization engine. Some of these interfaces have a default implementation (which can be overridden if the j Conduit developer desires), however others are left as abstract, and require an implementation be given to them by the jConduit developer. Thus the first step in developing any jConduit is to subclass *AbstractConduit*, and implement the desired and necessary methods.

To aid in internationalization and globalization of jConduits, the *AbstractConduit* class also defines some methods which need to return static string values to retrieve them from an jConduit developer supplied *ResourceBundle*. This is an easy and flexible approach to supplying much of the data a jConduit needs to display information to the user.

The *AbstractConduit* class has the following methods which *must* be implemented by direct subclasses:

getResourceBundleName	Returns a String object containing the full package and base class name of the resource set to be used with this jConduit. This resource bundle should include information such as the name of the conduit, its description, who developed it, and any text required by the jConduits settings panel.
startSync	This is where the primary data synchronization code will reside. This method is called by The jSyncManager when it is time for a given jConduit to perform its synchronization tasks.
constructConfigPanel	<p>This method needs to return an instance of <i>javax.swing.JPanel</i>. This JPanel object should contain all the controls you require for configuring your jConduit. It will be used by the jSyncManager when the user selects to configure your jConduit. You should provide whatever event handling your jConduit requires.</p> <p>It is not considered an error to return <b>null</b> for this call. If your jConduit requires no configuration, returning <b>null</b> is acceptable, as applications that accept jConduit plug-ins are expected to construct a default empty panel to handle this case.</p>

The *AbstractConduit* class also has the following methods which the jConduit developer can optionally override:

Constructor	<i>AbstractConduit</i> provides a default constructor with no parameters. The jSyncManager will only create new jConduit instances by a zero-parameter constructor. If your conduit needs to do any first-time initialization, it can provide its own implementation. Note however that as all jConduits are serialized and deserialized between application sessions, this constructor will only be called when the jConduit is first installed by the user.
getPriority	This method returns the priority byte for this jConduit. The priority byte can be used by a jConduit developer to influence (although not absolutely specify) when their jConduit is run at sync time. It is a value between 127 and -128, with 127 being the highest priority level, and -128 the lowest. At sync time, the installed conduits are sorted based on their priority levels before they are run. jConduits with the same priority level are run in an undefined order. This value can be especially useful for suites of jConduits that must be run in a particular order, and is very useful when used in conjunction

with the inter-jConduit communications system built into the ***ConduitHandler*** class. In general, jConduits that need to be run early in the synchronization should have a priority >0, and those that need to be run late in the synchronization should have a priority <0. Resist the temptation to use excessively high or excessively low priority values, as these are typically reserved for core jConduits bundled with the jSyncManager to handle data backup and restoration. For most jConduits, use the default implementation which uses a priority value of 0.

One final note on the ***AbstractConduit*** class. Subclasses of this class must be contained in a subpackage of the ***org.jSyncManager.Conduit*** package. This must be done in order to allow the The jSyncManager client to find your jConduit when a user enters its name in the "Add Conduit" dialog. This will minimize the possibility that classes you develop will conflict with those developed by the The jSyncManager team, or by other parties. Other classes you develop to supplement your jConduit may either be placed inside this package, or within another package of your choosing. You also have the option of putting your jConduits resource bundle class into a subpackage of the ***org.jSyncManager.Conduit.Resources*** package, however this is not strictly required. Your jConduit class itself *must* have the same name as its parent package in order for it to be discoverable by the jSyncManager.

### **A sample jConduit: the CSV Export jConduit**

Throughout this document, to supplement the discussion of the APIs used in jConduit development, we'll develop one of our own. The CSV Export jConduit will be a simple jConduit which can export data from the six standard Palm handheld databases (Address, Datebook, Expense, Mail, Memo, ToDo List) to a comma separated value file. We'll develop this jConduit in a step-by-step manner after introducing each new API set.

To begin, we need to create the skeleton of our jConduit class. We'll fill in some of the methods that provide static pieces of information, and will create our resources class as well. Full source for this jConduit is available for download from the jSyncManager website. To begin, we'll define the classes and packages we need to import, and the package declaration. Don't worry if some of these statements deal with packages not yet discussed in this text; they'll be covered later.

```
import org.jSyncManager.API.Conduit.AbstractConduit;
import org.jSyncManager.API.Conduit.ConduitHandler;
import org.jSyncManager.API.Protocol.Util.StdApps.*;
package org.jSyncManager.Conduit.CVSConduit;
```

Next, we'll define our class name, and extend the ***AbstractConduit*** class. Note that we're making this class public, in order to allow the jSyncManager to load it.

```
public class CVSConduit extends AbstractConduit {
```

Next, we'll probably want to define the fields used inside this class. We haven't discussed what our sample will need yet, so for now we'll leave this blank. However, it's worth noting that as the `jSyncManager` serializes and deserializes all `jConduits` at initial load and exit times that all fields you define will have their information persistently stored unless you mark them as *transient*. Fields marked as being *transient* should generally define a default value. This value will depend on your `jConduits` needs, however for reference types **null** is an obvious option. In fact, as will be discussed in the next section below, GUI parts used in configuration panels should generally be marked as transient with a default value of null, as it's faster to recreate the GUI at runtime than it is to deserialize the GUI from storage.

Lastly, we need implement the methods we want. Three of these are required for all non-abstract subclasses of *AbstractConduit*, the rest are to satisfy our design needs. For now, we'll just define our `getResourceBundleName` implementation; we'll define the other methods we'll want in the sections below. We'll explore the contents of this class in a later section.

```
public String getResourceBundleName() {  
    return  
        "org.jSyncManager.Conduit.Resources.Text.CVSResources";  
}
```

## The jConduit Configuration Panel

Non-trivial `jConduits` will generally need a way of interacting with the user prior to synchronization time in order to determine the users preferences, and any settings the `jConduit` may require. The `jSyncManager` uses standard Java Swing components to present the user with it's graphical user interface, and it provides `jConduits` the ability to specify their own Swing-based configuration panel. Each `jConduit` is granted one panel, however developers requiring more may either use this one integrated panel as a launch-point for other frames, or can make use of a layout manager such as *CardLayout* which allows multiple panels to inhabit one panel region, with only one visible at a time. Note that if the `jConduit` developer chooses this option, it is up to them to provide the necessary user interface elements to allow the user to "flip" through the panels they've defined.

The GUI parts are defined through the `constructConfigPanel` method. It should return a handle to a fully constructed and configured *JPanel* instance or subclass which has all of the developer desired GUI components on it. There are two ways you can approach this: for simple `jConduits`, you can define all of the parts as transient fields within your `jConduit` class, and construct them in the `constructConfigPanel` method; alternatively, for more

complex jConduits you may want to simply subclass *JPanel*, and simply construct and return an instance of it.

For our CVS jConduit, we'll want a configuration panel in order to allow the user to choose which type of CVS synchronization to run for each database at runtime (import, export or do nothing), and which files to use for each operation for each database. As this configuration panel will have a large number of choices, we'll put it in our jConduits package, in it's own class which we'll call *CVSConfigPanel*, which will be a direct subclass of *JPanel*. This class will also handle it's own event handling, however there is no reason why the jConduit class itself can't do so (we simply choose this design to minimize the complexity of our example by ridding it of code not specific to jConduit development). As a discussion of Swing development is outside the scope of this document, we won't explore the development of this class, however the source is included with the CVS jConduit sources on the jSyncManager website.

For our example, we will now define our implementation of `constructConfigPanel`. Note that *AbstractConduit* provides you with a *JPanel* field in which to store a handle to your configuration panel, called `configurationPanel`. Note that your `constructConfigPanel` method shouldn't assign the *JPanel* it constructs to this field directly -- this will be handled automatically by the jConduit through *AbstractConduits* public `getConfigurationPanel` method.

```
public JPanel constructConfigPanel() {
    return new CVSConfigPanel();
}
```

### jConduit Priority Levels

As mentioned above, jConduits can also define a synchronization priority level. This level should not be confused with the priority level of the thread that your jConduit is run in; it is solely used to determine in what order jConduits should be run when the user initiates a synchronization.

For most conduits, you should allow *AbstractConduit* to use its default value of zero. In particular, if your application simply synchronizes its data from one database to one data source, and collision or co-operation with other jConduits isn't a concern, use the default. Some jConduits, however, may need to be run before or after other jConduits, at which point using the priority levels becomes very important. An example of a jConduit that requires it be run after other jConduits is the backup jConduit included with the jSyncManager. In this case, we want to make sure that we backup the data as it is *after* the other jConduits synchronize, so as to backup the results of this synchronization.

Note that unlike some other data synchronization solutions, the jSyncManager doesn't impose any special limitations on what jConduits can sync with what databases, or how many

databases a single jConduit can synchronize with. This provides increased flexibility to the developer, however it comes at a cost: the developer has to be aware that they may not be the only jConduit to access a given database on the handheld. Correct usage of the jConduit priority levels can help you maintain the integrity of your data by allowing you to control when your jConduits are run.

Our CVS Export jConduit should run after most other jConduits. As its only duty is to export data in CVS format to individual files, it should run after most other jConduits have had a chance to synchronize in order to ensure that it picks up on the changes these other jConduits cause. However, we need to be mindful that we don't interfere with the backup, installer, and deleter jConduits, which we generally like to see run last. To this end, we want to define a priority level less than the default priority level of zero.

The *AbstractConduit* class provides three pre-defined priority levels you can use: HIGH\_PRIORITY, NORMAL\_PRIORITY, and LOW\_PRIORITY. You may use these priority levels as-is, or may return modified versions of them through standard mathematical manipulations.

For example, with our jConduit, let's assume we want to sync somewhere between the normal and low priority levels. We can implement our `getPriority` method like this:

```
public int getPriority() {
    return (NORMAL_PRIORITY + HIGH_PRIORITY) / 2;
}
```

It's worth noting that this value doesn't need to be statically specified. In some situations, you may want to give the user an option through your jConduits configuration panel as to when to run. This option may be given in a variety of ways, such as through a set of three radio buttons (for High, Medium and Low priority, using the pre-defined priority levels mentioned above, for example), or perhaps through a slider that can allow the any of the values between +126 and -127 (note that you should avoid allowing the user to select +127 and -128, as these values are reserved for the jSyncManager core jConduits, such as the Default jConduit, and may cause problems if your jConduit synchronizes before or after the core jConduits do). You may also want to dynamically determine the priority level via an offset against another jConduit. If you want to guarantee that one jConduit always runs just before another jConduit which uses a dynamic priority setting, implement your jConduits `getPriority` method such that it just adds one to the value returned by `getPriority` in the other jConduit.

You should also note that jConduits are sorted at the beginning of each and every synchronization session based on their reported priority levels. Any dynamic changes will only be reflected at the beginning of the next synchronization. Once a synchronization has begun, the order cannot be influenced. Two or more jConduits may also have the same priority level; in this situation, the jConduits will be run one after another only after all other jConduits with a

higher priority have had a chance to run, but before any with a lower priority have run. The order in which jConduits with the same priority level are run is not specifically defined, and thus they may be run in any order. If you need to guarantee that your jConduit runs before or after one or more other specific jConduits, set a suitable priority level.

Lastly, it's worth noting here that any given jConduit can only have one priority level. You can't create a single jConduit that synchronizes both before *and* after other jConduits. The jSyncManager only allows one instance of any given jConduit to be active at once, so if you have a situation where you need to perform some operations both before and after other specific jConduits have run, you will need to break up the desired logic into two distinct jConduits, one with a relatively high priority, and the other with a lower one. The jSyncManager provides a mechanism for inter-jConduit communications to better synchronize efforts between two or more jConduits that you can use in this situation. This facility will be discussed in the section below on the *ConduitHandler* class.

## jConduit Resources

The jConduit resources class is assumed to be something that is loadable via the standard java `getBundle` methods available in *java.util.ResourceBundle*. All of the conduits provided with the jSyncManager use instances of *PropertyResourceBundle*, however you may elect to use any subclass of *ResourceBundle* (or *ResourceBundle* itself), including *ListResourceBundle* and *PropertiesResourceBundle*.

Your jConduits resource class can contain whatever resources your jConduit requires. However, there are some standard resource keys which need to be provided in order for the method implementations provided by *AbstractConduit* to work correctly. These keys are:

<i>conduit.name</i>	The name of your jConduit. This should be a String of not more than 64 characters.
<i>conduit.description</i>	A description for your jConduit. This is a String.
<i>conduit.developer.name</i>	The name of the jConduit developer, or company that developed the jConduit. This should be a String of not more than 64 characters.
<i>conduit.developer.email</i>	An e-mail address that can be used to reach the jConduit developer. This should be a String with a valid e-mail address.
<i>conduit.copyright</i>	The copyright statement for this jConduit. This is a String.

Here is part of the implementation of the *CVSXResources* class to satisfy our jConduits current needs. We'll add to this as necessary, to make our jConduit easily internationalizable in the future.

```
import java.util.ListResourceBundle;
```

```

package org.jSyncManager.Conduit.Resources.Text;

public class CVSXResources extends ListResourceBundle {

    static final Object[][] contents = {
        {"conduit.name", "CVS Export jConduit"},
        {"conduit.description", "This jConduit will
            export data from the standard Palm application
            to comma separated value files."},
        {"conduit.developer.name", "Brad BARCLAY"},
        {"conduit.developer.email",
            "bbarclay@jsyncmanager.org"},
        {"conduit.copyright", "Copyright (C) 2002 Brad
            BARCLAY."},
    };

    protected Object[][] getContents() {
        return contents;
    }
}

```

### The ConduitHandler Class

*ConduitHandler* is the workhorse class used by jConduits to communicate with the handheld device during synchronization. It provides all the methods you'll need to query, read, and write data to and from the handheld system.

The *ConduitHandler*'s function is relatively simple. An instance of *ConduitHandler* is created at the start of each and every synchronization session. It is attached on the back end to the primary interface to the jSyncManager protocol stack (specifically, to the class *org.jSyncManager.API.Protocol.JHotSync*). When it's time for each jConduit to synchronize, the the jSyncManager synchronization engine will call the *startSync* method for the jConduit whose turn it is to sync, passing it two pieces of information: the handle to the current *ConduitHandler*, and the handle to a class that provides information on the current user (namely an instance of *DLPUserInfo*, which will be discussed later). Inside the *startSync* method, the jConduit developer will make calls to the methods inside the current instance of *ConduitHandler* in order to communicate with the handheld.

*ConduitHandler* has over fifty methods which you can use to open databases, query their contents, and read and write records, resources, application blocks and sort blocks. It also contains methods to allow your jConduit to leave and retrieve temporary messages for other jConduits, and to interface with the *JFrame* control of the main jSyncManager GUI (useful for displaying dialogs at synchronization time that need a parent handle). It also provides methods for maintaining the connection to the handheld system during long segments of processing. Discussing each and every method in this class is outside the scope of this document, however

these methods are all discussed in detail in the `jSyncManager` class documentation for the *ConduitHandler* class (available for viewing and download from the `jSyncManager` website).

There are some important methods that will prove both necessary and useful for nearly all `jConduits`. A sampling of these methods are described here.

#### openDatabase, closeDatabase:

All methods in *ConduitHandler* that access or manipulate information in a database on the handheld require a handle to the open database you want to access. Obtaining that handle is the job of the *openDatabase* method.

*openDatabase* requires two parameters: the name of the database (as a `String`), and a byte representing the open mode flags. These flags are stored in the *DLPDatabase* object, and are **READ\_MODE**, **WRITE\_MODE**, **EXCLUSIVE\_MODE**, and **SHOW\_SECRET**. To open using multiple flags, simply use the logical OR operator (`|`). For example, to open the Address Book in read and write mode with secret records visible, you would call:

```
int dbHandle = conduitHandler.openDatabase("AddressDB",
                                           DLPDatabase.READ_MODE |
                                           DLPDatabase.WRITE_MODE |
                                           DLPDatabase.SHOW_SECRET);
```

Conversely, the *closeDatabase* method takes the specified open database handle, and closes it, preventing the database from being accessed again until it is reopened.

#### readRecordByID, readRecordByIndex:

These two methods provide the primary means for reading data from a given database. They both require that the database to access be opened with the read attribute set.

The two methods differ only in how they reference records. The first, *readRecordByID* reads the record with the specified 32 bit record ID number. Every record in every database on the handheld has a unique 32 bit ID number associated with it. It is an error to request a record by ID number if the record with that ID number doesn't exist in the specified database. In the second case, *readRecordByIndex*, records are referenced by an ordinal value based on their order in the database (ie: the first record, regardless of its 32 bit record ID, has an index of '1', the second an index of '2', etc.).

#### readNextModifiedRecord, readNextModifiedRecordInCategory, readNextRecordInCategory:

These three methods provide a more selective mechanisms for reading records from an open database than *readRecordByID* and *readRecordByIndex*. *readNextModifiedRecord* allows

you to iteratively cycle through a list of records in the specified database that have their modified flag set. The modified flag is set by the handheld when the user adds, updates or deletes a record on the handheld. *readNextModifiedRecordInCategory* is even more selective; iterative calls to this method return only those modified records that have the specified category value. Likewise, *readNextRecordInCategory* also returns the next record in the specified category, however it does so irrespective to the state of the modified flag.

All three of these methods can be used in conjunction with the *resetRecordIndex* method, which allows you to reset the next record pointer back to the first one in the database.

#### writeRecord:

*writeRecord* refers to two overloaded methods of the same name. Both are used to write a record to the specified database (which should be opened in write mode). In the first case, *writeRecord* takes a ***DLPRecord*** object (or subclass) and a set of flags (one or more of **ARCHIVED, DELETED, DIRTY, and SECRET** from ***DLPRecord***), and writes them to the handheld. In the second case, *writeRecord* provides a more primitive form that allows you to write a record as an array of bytes.

#### startTickles, stopTickles:

These two simple methods take no parameters, and simply allow an jConduit to start and stop "tickle" packet transmissions. Tickle packets are used by the handheld to retain the communications link by resetting the connection timeout counter on both ends, during lengthy operations on either end. jConduits that expect to do any operations that will last more than a second (that don't involve reading or writing data to/from the handheld) should first make a call to *startTickles*, and once they are ready to resume operations to call *stopTickles*.

Note that it is not illegal to call *startTickles* multiple times in a row without calling *stopTickles*. Note, however, that the protocol stack ignores all calls to *startTickles* after the first time it has been called, until a call to *stopTickles* is made. Conversely, no matter how many times you call *startTick*

#### storeProperty, getProperty:

These two methods are useful for storing information for access between two or more related jConduits. They provide access to a standard Java HashTable (***java.util.HashMap***) that the ***ConduitHandler*** maintains on behalf of all jConduits. You may define whatever keys you so desire when you store your data, however it is suggested that you use a String key based on your jConduits package name, plus whatever identifier you wish to use. Following this guide will reduce the possibility that jConduits from other developers will overwrite your stored objects with their own using the same key.

Note that the data stored within this *HashTable* is only valid during the context of the current synchronization. It will not be available at the beginning of the next synchronization, as the *ConduitHandler* instance is regenerated at the beginning of each synchronization. Thus, if you wish to store data persistently between synchronizations, you will need to do so elsewhere.

#### cleanupDatabase:

Calling this method is generally a good idea at the end of your jConduits synchronization logic. Calls to this method force the database being cleaned up to flush all deleted records from the handhelds memory, and will reset all the modified flags. Failure to call this method at the end of synchronization will cause deleted records to remain in the handhelds memory, and modified records to continue to show up as modified during the next synchronization.

### **Data Abstraction Objects**

#### **Conduit Helper Classes**

Of special interest to some developers will be the standard abstract subclasses of *AbstractConduit*. These classes are used to make developing certain types of jConduits easier. Currently, there is only one such jConduit in the core jSyncManager API, called *AbstractInstaller*. The details of this helper jConduit are described in the section below.

#### AbstractInstaller:

The jSyncManager package contains two useful ready-to-run jConduits: *Installer* and *NetInstaller*. Both of these jConduits provide the ability to install standard Palm PRC and PDB databases to the Palm -- in the first case, by allowing the user to load the desired databases via a GUI interface, and in the latter case loading them from a list of databases, by URL, retrieved from a list on a filesystem or webserver. Both of these jConduits build upon the *AbstractInstaller*.

*AbstractInstaller* provides everything needed to create you own database installation jConduit. Subclasses don't need to implement the `startSync` method -- instead, they implement a `getDatabaseList` method that returns an array of *DLPDatabase* objects to be installed, and the *AbstractInstallers* `startSync` method will take care of the actual input/output details. The abstract methods that need to be implemented are as follows:

<code>getDatabaseList</code>	
<code>isDatabaseInstallAllowed</code>	
<code>clearDatabaseList</code>	
<code>installingDatabase</code>	

skippingDatabase	
exceptionInstallingDatabase	

Note that *AbstractInstaller* doesn't implement most of *AbstractConduits* abstract methods. Thus, your *AbstractInstaller* subclass will still need to implement methods that are jConduit-specific, such as `getResourceBundleName`.