5

Exploring ManifoldCF's Core Services

This chapter covers

- Global ID generation
- Synchronization services
- Cache management services
- Database services

You have now reached the part of this book that is meant to help people learn how to write their own connectors. In this chapter, you will become familiar with how to use the most important ManifoldCF core services, such as synchronization, caching, and database access. These services underlie the ManifoldCF framework, and are an essential part of the environment under which ManifoldCF connectors operate. Any connectors you write will thus be using these services, either directly or indirectly.

As a way of learning the proper techniques, we will be writing example code that uses these services in a simple command-line environment to allow us to explore their functionality in isolation, without undue complication. Our goal is to become familiar with the way one writes ManifoldCF code, while we learn about services you will need to be familiar with in order to design smart connectors. By the time we're done, your ManifoldCF bag of tricks should be pretty well filled out.

5.1 ManifoldCF coding fundamentals

The next time you run into Bert, he's been making great progress with ManifoldCF. His search engine has plenty of properly secured content to search, and his boss is happy because he has a big red light on his desk that Bert's automated software knows when to light. Bert's reputation has definitely improved, and his expertise in integrating ManifoldCF with other enterprise systems is considered sublime.

But there is one thing that has Bert worried. His corporate arch-nemesis, Frank, has been given the go-ahead to produce a home-grown content management system, into which a good chunk of the company's content will eventually migrate. Frank is apparently hoping that his efforts will derail Bert's sterling reputation once and for all. So Bert needs to learn how to write his own connectors – not just repository and authority connectors, but also output connectors, because it has occurred to Bert that he just might be able to use a custom output connector to inject enough existing content into Frank's new system to make its likely flaws readily apparent to all that use it.

You assure Bert that writing ManifoldCF connectors is straightforward enough, if you understand the rules. But, you also point out, there's plenty of stuff you really want to know **before** you design any connector. Most of the work of writing a connector takes place before the first line of code is ever written, and involves thinking through exactly how the connector is going to work. In order to do that effectively, you need to know what services and capabilities are available for your use within ManifoldCF. That's what we're going to do in this chapter. We're going to start with the easy stuff and gradually progress to the more advanced. But first, we need to understand the basic rules of how to write code in the ManifoldCF world, including what kinds of threads there are, how to work with them, how to throw exceptions, and how to write to the log.

5.1.1 Thread types and thread rules

Every ManifoldCF process has its own unique thread structure. The command-line utilities in general make do with a single main thread. An Agents Process starts with a single main thread, but when it fires up the Pull Agent crawler it creates hundreds of helper threads, which persist for the lifetime of the process. For the web applications, there is some number of request threads, which are created by the Java application server under which all the ManifoldCF web applications must operate.

All of the threads I've described so far have one unifying feature: they are meant to be cleanly shut down, rather than abandoned. Although these threads may technically be created as standard Java daemon threads, which do not block process shutdown, ManifoldCF nevertheless treats these threads very specially. They are called *ManifoldCF persistent threads*.

> **Definition** A *ManifoldCF persistent thread* is any thread whose continued execution prevents the process in which it is running from terminating itself.

On the other hand, ManifoldCF also makes use of threads that come into existence as needed, usually very briefly, and never interfere in any way with process shutdown. These threads are essential, as we will see. I'll call these *transient threads*.

> **Definition** A *ManifoldCF transient thread* is any thread that is safe to terminate at any point, should the process it is running in be terminated.

Both of these permitted flavors have their own sets of rules, which must be obeyed rigorously, or ManifoldCF's processes will not shut down cleanly. Let's examine each type of thread in turn.

PERSISTENT THREADS

Most ManifoldCF threads we will be working with in this chapter are of the persistent variety. To end one of these threads, they must be explicitly terminated. The way they are terminated is by being signaled by way of a `Thread.interrupt()` method invocation. It is then the persistent thread's responsibility to shut itself down cleanly.

This turns out to be a challenge in versions of Java beyond 1.4. In Java 1.4, a thread that was waiting on a socket or on other I/O could be made to wake up and throw an `java.io.InterruptedIOException` if another thread called `interrupt()` on the associated `Thread` object. This was convenient and would allow a clean shutdown to be possible no matter what the thread was doing. Third party Java libraries, however, were not necessarily good citizens in this regard – they could (and did) sometimes silently "eat" the exception, which blocked shutdown from proceeding.

The situation got even worse in Java 1.5. The contract for socket waits changed, so that `interrupt()` no longer had any effect whatsoever on threads that were waiting in this way. It became no longer possible to reliably shut down any thread that might be doing socket-related activities.

The result of all this is that persistent threads are not permitted to undertake any socket activities. More broadly, persistent threads cannot undertake any activities which cannot be interrupted.

But why does ManifoldCF need persistent threads at all? The answer has to do with how ManifoldCF manages thread and process synchronization. Later in this chapter, you will learn that one of the core services that is used by nearly all ManifoldCF processes is the synchronization service. Other core services also use the synchronization service, either directly or indirectly, including the caching service and the database services.

And this is the reason that persistent threads are necessary. In a multi-process deployment, in one deployment model the synchronization service uses the file system to implement its synchronization methods. Synchronization via the file system means that all state changes that are reflected in the state of the file system as a result of synchronization must be reversed on process exit, without exception. Otherwise, since you cannot shut

down any single process cleanly, you would be forced instead to shut down *all* processes whenever you shut down one, and also clean up the synchronization file area afterwards.

ManifoldCF uses persistent threads to avoid this kind of problem. Since persistent threads are never abandoned, code that executes in one of these threads is guaranteed to complete the instructions written by the programmer. This makes it possible to reliably clean things up. As a result, the code executed by persistent threads has the full use of all the ManifoldCF core services, such as synchronization and caching, so long as the persistent thread code takes care to use these services properly, and cleans up properly even when exceptions are thrown.

> **Remember** When you create a connector, and implement your own connector class, it must be coded to operate properly in ManifoldCF's multi-threaded environment. Each of your connector methods must safely be callable by a wide variety of ManifoldCF persistent threads, obeying a specific threading model. The model's details will be discussed in Chapter 6.

ManifoldCF also allows multiprocess synchronization using Apache ZooKeeper. ZooKeeper synchronization is more robust than file system synchronization – and in the simple case of a process shutting down suddenly, ZooKeeper may well be able to fully recover. But the details of the synchronization implementation hidden from the rest of the ManifoldCF software – and so, the contract must be obeyed for code to be considered correct.

TRANSIENT THREADS

We've learned why ManifoldCF persistent threads are necessary. But we also have learned that they are prohibited from communicating via sockets, or indeed using any extended-time I/O operation. How do we resolve this conundrum?

We resolve it by using transient threads to perform all socket-related activities. These threads I've called "transient" for a reason – because they usually come into existence for a very limited time, do the job they were created for, and immediately thereafter go away. Since they are always created as Java daemon threads, their existence will not block the shutdown of the process. Indeed, ManifoldCF counts on this behavior, because exiting the process is the only way to interrupt socket communications in Java now.

It should be clear that transient threads must not use any of the ManifoldCF core services. Violating that constraint would potentially lead to a corrupt set of synchronization files when a ManifoldCF process is shut down. This is one reason that the lifetime of such threads is typically so short. Figure 5.1 shows the typical relationship between a persistent thread and some transient threads.

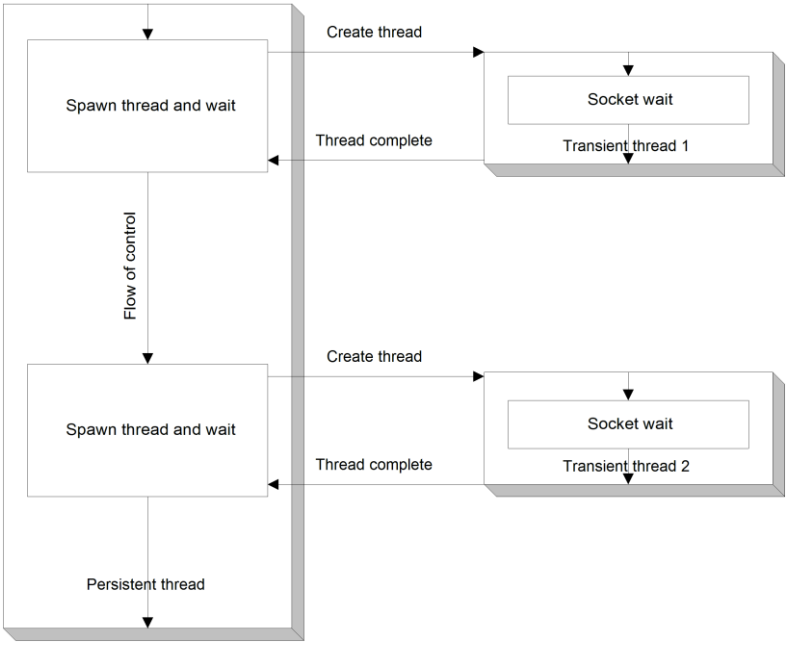

Figure 5.1 Flow of control demonstrating proper handoff of control from a persistent thread to two transient threads, each of which does some socket-related activity before completing.

It isn't just straight socket usage that might be a problem. Sometimes a library or API includes methods that need to use sockets inside. Also, it is the case that some third-party java libraries do not use sockets but nevertheless are poorly behaved in the event of a `Thread.interrupt()` request. In either case, it is necessary for the involved ManifoldCF code or connector code to insulate the library or API by creating transient threads for every API method that may be problematic. This can result in a fair bit of coding work, but there is no avoiding it.

> **Note** Even persistent threads in ManifoldCF are usually created as Java daemon threads. The reason for this is to make it possible to easily kill the corresponding ManifoldCF process, even if this is the wrong way to shut it down.

5.1.2 Thread contexts

To save work, ManifoldCF provides a mechanism for keeping data around that is local to a persistent thread. This thread-local storage mechanism is called a *thread context*. You will find that many of the classes and methods in ManifoldCF require a thread context as an

argument. The type of the argument is `org.apache.manifoldcf.core.interfaces.IThreadContext`.

The methods supported by a thread context are relatively simple. If you look at the interface above, you will see that there is a `save()` method and a `get()` method, for saving some keyed piece of data, and later retrieving it.

A thread context is usually created shortly after a persistent thread begins executing. In the case of a main thread or request thread, ManifoldCF creates the thread context as soon as ManifoldCF gets control. The way to create a thread context is as follows:

```
IThreadContext context = ThreadContextFactory.make();
```

This seems pretty straightforward. But that's not all there is to it. While it is sometimes acceptable for there to be more than one thread context per actual thread, it is **never** acceptable to trade thread contexts across threads. Most of the time, when you are writing connectors in ManifoldCF, there is rarely any need to worry about this situation, because most objects are created and used by a single thread only. But, occasionally, people need to create shared objects. When this happens, these people are occasionally tempted to store thread contexts, or (more typically) other objects or handles that were created using a specific thread context, in objects which may be used by more than one ManifoldCF persistent thread. Doing this is a great way to build software that fails in all sorts of non-obvious ways, and this practice should be avoided at all costs.

For example, listing 5.1 should immediately raise red flags in your mind.

Listing 5.1 A very bad way to construct a shared object

```
public class MyObjectFactory
{
  static Integer lock = new Integer(0);
  static MyObject multithreadReference;

  public static MyObject get(IThreadContext tc)
  {
    synchronized (lock)                                                    #A
    {
      if (multithreadReference == null)                                    #B
        multithreadReference = new MyObject(tc);                           #B
      return multithreadReference;
    }
  }

}

public static class MyObject
{
  IDBInterface database;

  public MyObject(IThreadContext tc)
  {
    database = DBInterfaceFactory.make(tc,"mydatabasename",                #C
      "myusername","mypassword");                                          #C
  }
```

```
  ...                                                                    #D
}
```

#A Only one thread at a time
#B If object not yet created, create it
#C Create member variable "database"
#D Do stuff that needs "database"

In this example, the programmer attempts to use a shared object in multiple threads. The intent is that a thread uses the `MyObjectFactory.get()` method to get a reference to the shared object. The shared object (of class `MyObject`) is, however, initialized in a manner that requires a thread context! Even worse, looking inside the `MyObject` class, you can see it is using the thread context to create a secondary object, which is then being saved as a member variable, so someone looking at only the `MyObject` member variables might not even be able to figure this out. This is definitely not going to work in real life.

So what is the proper way to construct code like this? Actually, there are two answers. One answer involves passing the thread context into each method of the `MyObject` class, so that each method will create the database handle. See listing 5.2.

Listing 5.2 One solution to the problem of cross-thread shared objects

```
public class MyObjectFactory
{
  static Integer lock = new Integer(0);
  static MyObject multithreadReference;

  public static MyObject get()
  {
    synchronized (lock)
    {
      if (multithreadReference == null)
      {
        multithreadReference = new MyObject();                           #A
      }
      return multithreadReference;
    }
  }

}

public static class MyObject
{
  public MyObject()
  {
  }

  protected IDBInterface makeDatabaseHandle(IThreadContext tc)           #B
  {                                                                      #B
    return DBInterfaceFactory.make(tc,"mydatabasename",                  #B
      "myusername","mypassword");                                        #B
  }                                                                      #B
```

```
    ...                                                                    #C
  }
```

#A Construct object without thread context
#B Protected method builds database handle
#C All methods have thread context argument

In this solution, the object is still shared across threads with the same semantics, but care has been taken to be sure that no thread-context-related data is stored as a class member. This, of course, means that every `MyObject` method that needs a database will have to receive a thread context argument.

Is there a better way? There may be – but only if the contract is changed somewhat. You see, as soon as the shared object becomes specific to a thread **in any way**, it can only be used by that one thread until that thread is done with it. So, instead of allowing multiple threads to use the same object simultaneously, we can imagine a shared object that only one thread at a time can use. See listing 5.3.

Listing 5.3 Introducing a setContext() method to allow cross-thread sharing

```
public class MyObjectFactory
{
  static Integer lock = new Integer(0);
  static MyObject multithreadReference = new MyObject();

  public static MyObject grab(IThreadContext tc)                           #A
    throws InterruptedException                                            #A
  {
    while (true)
    {
      synchronized (lock)
      {
        if (multithreadReference == null)
        {
          lock.wait();
          continue;
        }
        MyObject rval = multithreadReference;
        multithreadreference = null;
        rval.setContext(tc);                                               #B
        return multithreadReference;
      }
    }
  }

  public static void release(MyObject myobject)                            #C
  {
    synchronized (lock)
    {
      myobject.setContext(null);                                           #D
      multithreadReference = myobject;
      lock.notifyAll();
    }
  }
}
```

```
public static class MyObject
{
  protected IDBInterface database = null;

  public MyObject()
  {
  }

  protected void setContext(IThreadContext tc)                          #E
  {                                                                     #E
    if (tc == null)                                                     #E
      database = null;                                                  #E
    else                                                                #E
      database = DBInterfaceFactory.make(tc,"mydatabasename",           #E
        "myusername","mypassword");                                     #E
  }                                                                     #E

  ...                                                                   #F
}
```

#A Method to get shared object
#B Initialize with the thread context
#C Method to release shared object
#D Clear the thread context, to prevent problems
#E Context set method initializes "database"
#F Methods use "database"

In this implementation, the caller is meant to use the methods `MyObjectFactory.grab()` and `MyObjectFactory.release()` to obtain the object temporarily, and later release it. As part of obtaining the object, the object is initialized with the current thread context. The object is expected to be used by the same thread that performed the `grab()` operation, or there would be little point in the whole exercise.

Next, we'll look at another aspect of writing code in ManifoldCF: how to throw and handle exceptions.

5.1.3 Throwing and handling exceptions

Exception handling is one of those Java topics that everyone seems to have their own opinion on. But when you are working with ManifoldCF, it's essential to play by ManifoldCF's rules on the matter.

Most of the methods in the services we are describing exclusively throw an exception called `ManifoldCFException`. `ManifoldCFException` is, to some extent, a "catch-all" exception, in that lower level exceptions are often wrapped by `ManifoldCFException`. But, as usual, there's more to it than that.

When a `ManifoldCFException` is thrown, the entity reporting the exception must also characterize the exception as being one of several different types. This is not a mere nicety; much of the code in ManifoldCF will react differently based on what type of exception actually gets thrown. The method `ManifoldCFException.getErrorCode()` can be used to obtain this exception type. Table 5.1 describes the types, and what they mean.

Table 5.1 The ManifoldCFException error codes, and their meanings

Error code	Meaning
`GENERAL_ERROR`	Any generic error, not of the other kinds.
`DATABASE_ERROR`	An error occurred accessing or changing the database that was not related to the communication with the database, e.g. a permanent database error.
`SETUP_ERROR`	An error occurred which is best described as related to improper configuration.
`DATABASE_CONNECTION_ERROR`	An error occurred accessing or changing the database which was a direct result of communication problems with the database.
`REPOSITORY_CONNECTION_ERROR`	This is the error type that a connector should throw when there is a permanent error trying to connect to the repository it is supposed to be accessing, e.g. an authentication failure.
`DATABASE_TRANSACTION_ABORT`	While in a transaction, an error occurred that indicated that the transaction needs to be aborted and retried, usually due to a deadlock situation.
`INTERRUPTED`	The thread noticed that it was interrupted, either because of an explicit check, or because `java.lang.InterruptedException` or `java.io.InterruptedIOException` were thrown at a higher level.

HANDLING EXCEPTIONS

When any code of yours catches a `ManifoldCFException`, it can, of course, choose to re-throw it. But if you write code that needs to do something else, it's essential that it act in accordance with the meaning of the exception type.

For example, your code must never ignore a `ManifoldCFException.INTERRUPTED`-type exception, because if it does it may block the shutdown of a persistent thread. If you catch such an exception, be very careful to re-throw the same exception, or a comparable one. For example, see listing 5.4.

Listing 5.4 Handling an exception, making sure exceptions of type INTERRUPTED are rethrown appropriately

```
try
{
  ...                                                    #A
}
```

```
catch (ManifoldCFException e)
{
  if (e.getErrorCode() == ManifoldCFException.INTERRUPTED)                #B
    throw e;                                                              #B
  ...                                                                     #C
}
```

#A Code that throws ManifoldCFException
#B Make sure to rethrow interruptions
#C Do what you need to process the exception

Another case that may require special handling on your part is a `ManifoldCFException.DATABASE_TRANSACTION_ABORT`-type exception. This type of exception is thrown when the ManifoldCF database layer cannot complete a requested operation due to a deadlock situation. Typical writers of connectors never encounter this exception. However, if your connector has its own database tables, you may well need to catch it so that your database code properly backs off and retries the problematic transaction. We'll talk about this case in more detail in the chapter section on database services.

All of the other error types can be dealt with as you think best. Usually, the right thing to do is to allow all of these exceptions to be re-thrown; the ManifoldCF persistent threads are all perfectly capable of dealing with these cases. But if, instead, you want to just log the exception and continue, you may certainly do that safely, if the exception type is not INTERRUPTED or DATABASE_TRANSACTION_ABORT.

THROWING EXCEPTIONS

Throwing a `ManifoldCFException` is easy, as you might expect. You can throw a GENERAL_ERROR type of exception using the typical form of exception constructor, as follows:

```
throw new ManifoldCFException(message);
```

Or:

```
throw new ManifoldCFException(message,nestedException);
```

To throw any of the other types of exception, you need to include the exception type, e.g.:

```
throw new ManifoldCFException(message,ManifoldCFException.INTERRUPTED);
```

Or:

```
throw new ManifoldCFException(message,nestedException,
  ManifoldCFException.INTERRUPTED);
```

We're almost done now learning about the fundamentals of coding in ManifoldCF. Only one subject remains – how to write to the log.

5.1.4 Logging

ManifoldCF uses the Apache Commons-Logging library for its logging needs. This is a straightforward logging framework which segregates logging output streams into separate bins called *loggers*. Each logger can be controlled by means of a global configuration file, called logging.ini. In this file, you can set the format of individual log streams, direct specific loggers to specific files, and many other things.

The traditional way people use commons-logging is to create a logger per Java class. This has the benefit of very fine-grained control over what gets logged and what doesn't. But it is often very difficult to figure out what classes to select in order to get the big picture of what is going on in your program.

ManifoldCF therefore includes an alternate method of setting up loggers. While you can still use an individual logger per class, there's also a system-wide logger for each subsystem. These are all defined by the ManifoldCF framework, and they are enabled in the ManifoldCF `properties.xml` file, not in `logging.ini`.

The Pull Agent logger instances themselves, should you want to write log messages to them, can be found in the class `org.apache.manifoldcf.crawler.system.Logging`. This class inherits some of the loggers it supports from its parent class, `org.apache.manifoldcf.agents.system.Logging`, which also uses a similar mechanism for core system loggers. For Authority Service loggers, the class in which the loggers are defined is `org.apache.manifoldcf.authorities.system.Logging`.

Table 5.2 summarizes all framework-defined loggers, and also the `properties.xml` properties that enable them.

Table 5.2 Loggers, their purposes, and their associated properties

Logger object	Property name	Purpose
db	org.apache.manifoldcf.db	Database access logging
root	org.apache.manifoldcf.root	Root logger for all ManifoldCF loggers
misc	org.apache.manifoldcf.misc	Miscellaneous log output
lock	org.apache.manifoldcf.lock	Synchronization and lock management logging
cache	org.apache.manifoldcf.cache	Cache management logging
keystore	org.apache.manifoldcf.keystore	Keystore management logging
perf	org.apache.manifoldcf.perf	Performance-related logging
agents	org.apache.manifoldcf.agents	Agents management related logging
ingest	org.apache.manifoldcf.ingest	Target system ingestion logging (used by output connectors)

api	org.apache.manifoldcf.api	API access logging
threads	org.apache.manifoldcf.threads	Pull Agent thread activity logging
jobs	org.apache.manifoldcf.jobs	Job activity logging
connectors	org.apache.manifoldcf.connectors	Repository connector logging
hopcount	org.apache.manifoldcf.hopcount	Hop count calculation logging
scheduling	org.apache.manifoldcf.scheduling	Document priority and scheduling logging
authorityService	org.apache.manifoldcf.authorityservice	Authority service logging
authorityConnectors	org.apache.manifoldcf.authorityconnectors	Authority connectors logging

You can, of course, still choose to create your own loggers, and control them entirely through `logging.ini`. But, in case you would like to use the framework's loggers, this is how.

HOW TO LOG INFORMATION

In order to log information within your code, you need to have some idea what level of importance that information is. The commons-logging infrastructure supports five levels of information, as described in table 5.3.

Table 5.3 Commons-logging levels

Level	Meaning
FATAL	A fatal error, which is usually defined as an error that demands that the process be stopped. Note that it is very unusual for a connector to log errors of this kind.
ERROR	A common error, which is serious enough to affect the activity of the system, but which does not cause the process to be stopped.
WARNING	A situation which may require user attention, but which has little real impact on the operation of the system.
INFO	An informational message, which a user may be interested in, but which carries no implication of required action.

DEBUG	A message which may be helpful for diagnostics, but which is not usually needed by a user.

As you might suspect, there are different logger methods for each of these levels. For example, if you want to log a fatal error from within a repository connector, your code might look like listing 5.5.

Listing 5.5 Logging a fatal error

```
try
{
  ...                                                                    #A
}
catch (MyFatalException e)
{
  Logging.connectors.fatal("A fatal error happened: "+e.getMessage(),e); #B
  throw new ManifoldCFException("MyFatalException: "+e.getMessage(),e);  #B
}
```

#A Code that throws a fatal exception
#B Log and rethrow

Astute readers may note that there is an inconsistency in listing 5.5 between the log level of the error and the type of `ManifoldCFException` that gets thrown. The exception is of type `GENERAL_ERROR`, and is thus more suited to an `ERROR`-type log message.

In fact, there is really only one proper response to a fatal error, and that is to shut down the process or service, usually via a `System.exit()` call. But this is an exceedingly rare thing for even the ManifoldCF framework to be doing, let alone any individual connector! A more likely usage is captured in listing 5.6.

Listing 5.6 Logging a standard error

```
try
{
  ...                                                                    #A
}
catch (MyException e)
{
  Logging.connectors.error("An error happened: "+e.getMessage(),e);      #B
  throw new ManifoldCFException(e.getMessage(),e);                       #B
}
```

#A Code that throws a non-fatal error
#B Log and rethrow

This is better. The error is logged, including the text of the error message and the exception that was caught, and a consistent `ManifoldCFException` is thrown, which includes not only the original error text, but also the exception itself as a nested sub-exception of the new `ManifoldCFException`.

For warnings, the code will likely be quite similar, but may well not re-throw the exception, and instead handle it in some appropriate manner. See listing 5.7.

Listing 5.7 Logging a warning

```
try
{
  ...                                                                    #A
}
catch (MyException e)
{
  Logging.connectors.warn("Document "+docID+" could not be fetched: "+   #B
    e.getMessage()+"; skipping the document",e);                         #B
}
```

#A Code that throws an error we compensate for
#B Log the warning and keep going

If you are going to bother logging a warning at all, you should log enough information that someone looking at the logs can go back and figure out whether there's a serious problem or not. Just logging a generic message is not adequate, and code that works that way is sure to generate irate phone calls from your client base. Given the international nature of the software profession these days, Murphy's Law demands that the irate calls arrive sometime in the dead of night, which means potentially losing a perfectly good night's sleep just because you were a little lazy about what you put in one of your log messages. So, take my advice, and be rigorous about making sure that warnings have enough information that they can be readily interpreted by somebody other than you.

There is, of course, a much better way to record information of this general kind within a repository connector. Repository connector methods have access to the repository connection history, and they can record connector-specific activities and their results. These activities are recorded in a database table, and become available to ManifoldCF end users via the history reports, right in the ManifoldCF crawler UI. We will learn how to do this for various connector types in Chapter 7. My recommendation is to use **both** commons-logging **and** activity logging, where they are both available, since you probably cannot accurately predict in advance whether you will have access to the crawler UI or to the logs.

LOGGING DEBUGGING INFORMATION

Logging debugging information is a little different than logging errors and warnings. First of all, the information may be verbose. And second, the meaning of the information needs to make sense only to somebody who knows the connector code intimately. And, finally, you really don't want the debugging code to slow the whole system down even when debugging is off. Listing 5.8 shows an example which does precisely that.

Listing 5.8 Debugging code we never want to see

```
while (i < documentIdentifiers.length)
{
  String documentIdentifier = documentIdentifiers[i];
  // Log everything there is to know about this document
  Logging.connectors.debug("Document '"+documentIdentifier+              #A
    "' has the following characteristics:");                             #A
```

```
  RepositoryDocumentDescription rdd =                                    #A
    callFabulouslyExpensiveInterrogationMethod(documentIdentifier);      #A
  Logging.connectors.debug(" Document attribute A = "+rdd.getA());       #A
  Logging.connectors.debug(" Document attribute B = "+rdd.getB());       #A

  ...                                                                    #B

  i++;
}
```

#A Log everything about the document
#B Do the rest, which doesn't need rdd

The problem with this code is that it does a lot of work just to get hold of information that won't even be logged if connector logging is not set to `DEBUG` mode. Listing 5.9 shows the proper way to do the equivalent.

Listing 5.9 The right way to include debugging code that might be expensive

```
while (i < documentIdentifiers.length)
{
  String documentIdentifier = documentIdentifiers[i];
  if (Logging.connectors.isDebugEnabled())                               #A
  {
    Logging.connectors.debug("Document '"+documentIdentifier+            #B
      "' has the following characteristics:");                           #B
    RepositoryDocumentDescription rdd =                                  #B
      callFabulouslyExpensiveInterrogationMethod(documentIdentifier);    #B
    Logging.connectors.debug(" Document attribute A = "+rdd.getA());     #B
    Logging.connectors.debug(" Document attribute B = "+rdd.getB());     #B
  }

  ...                                                                    #C

  i++;
}
```

#A Check if debug logging is on
#B Log everything we want
#C Actually do stuff

Since logging is often a critical remote debugging tool, please don't neglect to add debug logging messages that record your connector's progress, and especially all results of critical decisions that your code makes. We'll talk more about this in Chapter 7 also.

ENABLING LOGGING

Now that we know how to log information, we still need to know how to turn on that logging. This is, of course, trivial. All we need to do is provide a `properties.xml` property of the appropriate name, with a value that's either `DEBUG`, `WARN`, `ERROR`, or `FATAL`. For example:

```
<property name="org.apache.manifoldcf.misc" value="DEBUG"/>
```

This will enable the framework's misc logger at the debugging level `DEBUG`, which also will capture info, warning, error, and fatal messages. That's all there is to it! The property will not take effect, of course, until the process you are interested in restarts, so you may need

to shut down ManifoldCF and start it up again. But that's usually only a minor inconvenience.

But, where does the log output actually go? The answer is that it goes wherever you tell it to go. All processes in ManifoldCF must be pointed at the appropriate `properties.xml` equivalent. This is often done through a `-D` switch to java, typically something like this:

```
java -Dorg.apache.manifoldcf.configfile=properties.xml …
```

The ManifoldCF `properties.xml` file, in turn, provides the path for the logging configuration file, usually called `logging.ini`:

```
<property name="org.apache.manifoldcf.logconfigfile"
  value="./logging.ini"/>
```

Finally, the `logging.ini` file actually refers to the log file, and also allows you to control formatting, the default logging level, and much more:

```
log4j.appender.MAIN.File=manifoldcf.log
log4j.rootLogger=WARN, MAIN
log4j.appender.MAIN=org.apache.log4j.RollingFileAppender
log4j.appender.MAIN.layout=org.apache.log4j.PatternLayout
```

You can learn much more about commons-logging at http://commons.apache.org/logging.

5.2 Global unique ID generation

With all the background issues discussed, we are now ready to start exploring the usage of ManifoldCF's core services. We'll start with something pretty simple: global unique identifier generation.

5.2.1 What unique ID generation does

A global unique ID, for ManifoldCF, is just a Java `String`, which contains a valid number that is guaranteed to fit in a Java `long`. While database software can, in theory, generate such identifiers readily, there are many situations in which it is useful to generate such an ID outside of the database. For example, ManifoldCF uses unique ID's as transaction identifiers, and also sometimes as keys for hash tables.

ManifoldCF provides the unique ID generation service because it is a bit more complex than just a simple monotonically-increasing `long` value. Specifically, the ID value returned must be safe against process stops and starts, as well as safe across multiple ManifoldCF processes. This latter feature is why it is called a "global" unique ID generator.

The class method that actually creates a unique identifier is `org.apache.manifoldcf.core.interfaces.IDFactory.make()`, which has the following method signature:

```
public static String make(IThreadContext tc) throws ManifoldCFException;
```

After obtaining an ID in this way, you are free to use it in whatever way you see fit, either as a unique column value in a database table, or in any other manner. You can be assured that the ID value will not be given to any other threads calling the same method in any of the other ManifoldCF processes.

UNIQUE ID GENERATION AND THREAD CONTEXTS

You may have noticed that you will need a thread context in order to use the unique ID service. As we have already discussed, this has some implications. In particular, it implies that you may only create such an identifier within a ManifoldCF persistent thread. But why should that be?

The reason for this restriction is that global ID creation requires synchronization. Synchronization, however, is the one critical activity in ManifoldCF that can never be interrupted or abandoned in the middle. Should this basic restriction be lifted, the potential will exist for dangling locks to be left around, at least on a multi-process ManifoldCF deployment.

> **Single process vs. multi-process deployments**
>
> What is the difference between a ManifoldCF single process, a file-based multi-process, and a ZooKeeper-based multiprocess deployment? The major distinction, as you might have guessed by now, is in how synchronization is performed. The `properties.xml` parameter `org.apache.manifoldcf.lockmanagerclass` determines how lock management is done. For memory-based single-process deployments, no additional parameters are needed. For file-based deployments, the `properties.xml` parameter `org.apache.manifoldcf.synchdirectory` specifies a disk directory, and controls whether (and where) disk-based synchronization will take place. For ZooKeeper deployments, `properties.xml` parameters determine configuration needed to talk to the proper ZooKeeper instances.

PERFORMANCE OF UNIQUE ID GENERATION

With the potential for disk-based synchronization going on, you may be concerned about the performance of unique ID generation in ManifoldCF. But ManifoldCF is clever in that it allocates identifiers in chunks to save time. Thus, file synchronization need take place only once every hundred identifiers that are needed.

As a result, unique ID generation is not considered expensive, at least not for the kinds of usages that it was initially created for, such as providing identifiers for database rows. You'd still probably not want to use it for constructing sorting bins or some other high-volume use, however.

5.2.2 A unique ID generation example

I've put together a simple example that demonstrates unique ID generation. It also happens to demonstrate a lot of what we've been talking about up until now – e.g. thread contexts, exception handling, and logging. This example does not do much – it just fires up two threads which each generate and print unique ID's. Each of these threads runs forever, until they are told to stop by the main thread, which gives them each one minute of execution before it signals them to shut down. Listing 5.10 has annotated highlights. The complete code can be found at

http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/core_example/src/org/apache/manifoldcf/examples/UniqueIDGenerator.java.

Listing 5.10 A unique ID generator example

```
public class UniqueIDGenerator
{
  private UniqueIDGenerator()
  {
  }

  public static void main(String[] argv)
  {
    try
    {
      ManifoldCF.initializeEnvironment(ThreadContextFactory.make());      #1
      Logging.misc.debug("System successfully initialized");

      IDGeneratorThread thread1 = new IDGeneratorThread("first thread");
      IDGeneratorThread thread2 = new IDGeneratorThread("second thread");

      try                                                                  #2
      {
        thread1.start();
        thread2.start();

        Logging.misc.debug("Threads have been started");

        ManifoldCF.sleep(60000L);                                          #3
      }
      finally
      {
        Logging.misc.debug("Shutting threads down");

        while (true)                                                       #4
        {                                                                  #4
          if (!thread1.isAlive() && !thread2.isAlive())                    #4
            break;                                                         #4
          if (thread1.isAlive())                                           #5
            thread1.interrupt();                                           #5
          if (thread2.isAlive())                                           #5
            thread2.interrupt();                                           #5

          Thread.yield();                                                  #6
        }

        Logging.misc.debug("Threads are down");
      }

      Logging.misc.debug("Example run complete");
    }
    catch (ManifoldCFException e)
    {
      e.printStackTrace(System.err);                                       #7
```

```
        System.exit(-1);                                                    #7
      }
      catch (InterruptedException e)
      {
      }
    }

    protected static class IDGeneratorThread extends Thread
    {
      protected String threadName;

      public IDGeneratorThread(String threadName)
      {
        super(threadName);
        setDaemon(true);                                                    #8
        this.threadName = threadName;
      }

      public void run()
      {
        IThreadContext tc = ThreadContextFactory.make();                    #9

        try
        {
          while (true)
          {
            if (Thread.currentThread().isInterrupted())                    #10
              break;                                                       #10

            System.out.println("Identifier: "+threadName+
              ": "+IDFactory.make(tc));

            ManifoldCF.sleep(1000L);
          }
        }
        catch (ManifoldCFException e)
        {
          if (e.getErrorCode() != ManifoldCFException.INTERRUPTED)         #11
          {                                                                #11
            Logging.misc.error("Thread got unexpected exception: "+        #11
              e.getMessage(),e);                                           #11
          }                                                                #11
        }
        catch (InterruptedException e)
        {
        }
      }
    }
  }
```

#1 Initialize process environment
#2 try-finally for threads
#3 Wait for a minute
#4 Wait until both threads done
#5 Interrupt threads if still alive
#6 Yield so threads can finish
#7 Exception on initialization

#8 Make sure thread is daemon
#9 Each thread has thread context
#10 Explicitly check if interruption has been signaled
#11 Handle interruption exception specially

At #1, we initialize the ManifoldCF core services, which must be done once per process. This will read the `properties.xml` file and set up all the core services internal structures. If that should fail, then we catch the error at #7, where we cannot presume the existence of logging or anything else, so we just print the exception and exit. At #2, we have already created the threads, but have not started them yet. We use a `try-finally` block to insure we shut down the threads under all circumstances. At #3, we wait for one minute, using a convenient ManifoldCF primitive which works better than `Thread.sleep()` on Linux systems where it is possible for the clock to be changed during the sleep. At #4, we enter the thread shutdown loop, where we wait for both threads to become inactive before we exit. In that loop, at #5, we send the interruption signal to the threads. Then, at #6, we relinquish thread control so that the threads can actually shut themselves down.

The thread class also has some noteworthy logic. At #8, we make sure to set the thread to be the Java daemon type, so it will not block process shutdown. For each thread, we need to create (at #9) a thread context. At #10, in the main loop we explicitly check for a thread shutdown. At #11, we silently terminate the thread when we see interruption exceptions. This is indeed a general policy.

BUILDING AND RUNNING THE EXAMPLE

You can run the example to see what it does by checking out the following directory and using `ant` to build it and run it. Unpack the zip file, or check it out as follows:

```
svn co
http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/core_example
```

To build the example, change the current directory to `core_example`, and type:

```
ant jar
```

Of course, you can "cut to the chase" and run it by typing:

```
ant run-id-generator
```

You should then see output that includes something like this:

```
run-id-generator:
     [java] Configuration file successfully read
     [java] Identifier: first thread: 1296310522228
     [java] Identifier: second thread: 1296310522227
     [java] Identifier: second thread: 1296310522226
     [java] Identifier: first thread: 1296310522225
     [java] Identifier: first thread: 1296310522224
     [java] Identifier: second thread: 1296310522223
     [java] Identifier: first thread: 1296310522222
     [java] Identifier: second thread: 1296310522221
     [java] Identifier: first thread: 1296310522220
     [java] Identifier: second thread: 1296310522219
     [java] Identifier: first thread: 1296310522218
     [java] Identifier: second thread: 1296310522217
```

You can also look at the log file, which contains the log output from the example. You will find that in `manifoldcf.log`, right alongside the `properties.xml` file and `logging.ini` files I created for this test. The log output should look something like:

```
System successfully initialized
Threads have been started
Shutting threads down
Threads are down
Example run complete
```

Interrupting the example

On most operating systems, it is possible to kill a process outright. Such processes cannot be guaranteed to finish up properly, unless very special care is taken. In our example, if you ^C out of the example run, you will be effectively killing the process in just such a way.

One of the best ways to get around this problem in Java is to register a process shutdown hook. That's indeed what the Quick-Start example does. Such a technique still does not prevent all methods of killing a process abruptly, however. When a ManifoldCF process is shut down in an abrupt manner, and the system is operating in multi-process mode, then it is essential to clean the system locks. To do this, first shut down all ManifoldCF processes. Then, run the Java class `org.apache.manifoldcf.core.LockClean`. Restart the stopped processes, and all locks will have been reset.

Now that we have some idea what we are doing, we can move on to a more advanced topic: synchronization and locking. That's what we'll look into next.

5.3 Synchronization services and locking

Efficient synchronization and coordination of processes and threads is an important component of any multi-process or multi-threaded system, and ManifoldCF is no exception. But in one very important respect, ManifoldCF supports this basic need in a manner that is unusual: it abstracts from the nuts-and-bolts of Java and operating-system-based synchronization, and presents synchronization and locking as an abstraction. In this section, we'll discuss the ManifoldCF abstraction, how to use it, and the different ways it can be configured.

Note You are not absolutely required to use ManifoldCF's synchronization and locking mechanisms when writing your own connectors. Most likely your connector will not need to use any of these mechanisms directly anyway. But if your connector requires explicit locking and synchronization, and you use your own techniques, you may find that either your connector does not function properly in a multi-process setup, or you will find it necessary to add an extra configuration burden for users of your connector. I therefore highly recommend using these services when possible.

5.3.1 The synchronization abstraction

All of ManifoldCF's synchronization logic is encapsulated in one interface. That interface is `org.apache.manifoldcf.core.interfaces.ILockManager`. You create an instance of this object by the following code:

```
ILockManager lockManager = LockManagerFactory.make(threadContext);
```

The `ILockManager` interface contains a number of inter-process and inter-thread communication methods. Indeed, in flavor it's a little bit like a small operating system. I've summarized the broad sets of functionality in table 5.4.

Table 5.4 Synchronization functionality in ILockManager

Functional group	What it does
Services	Keeps track of cross-cluster service registration, so that resources can be properly apportioned
Locks	Cross-cluster locks, which gate thread execution access to sections of code
Critical sections	Inside-process locks, which gate thread execution access to sections of code
Global flags	Boolean flags visible to all processes in the ManifoldCF process family
Global shared data	Chunks of binary data visible to all processes in the ManifoldCF process family

Let's examine each functional group in turn.

5.3.2 Services

A *service* is a description of functionality provided somewhere in the cluster. This part of the abstraction was introduced when ManifoldCF began to support distributed processing.

The service abstraction describes each service as having a type and a name. Within a given service type, the name must uniquely describe the service somewhere on the cluster. Services must be registered with the lock manager in order to be known to it. A service, once registered, can be in either the "active" or "inactive" states. The goals of all this are the following:

- To be able to count all active services of a given type;
- To be able to clean up if a service of a given type is no longer active.

The basic `ILockManager` primitives provided to support the service abstraction are described in table 5.5.

Table 5.5 ILockManager primitives supporting services

Method	Function
`registerServiceBeginServiceActivity()`	Register a service of a given type with a given name, and mark the service as active
`updateServiceData()`	Update the data associated with an active service
`retrieveServiceData()`	Retrieve the data associated with an active service
`scanServiceData()`	Scan the service data for all services of a given type
`cleanupInactiveServices()`	For every inactive service of a specified type found, call a provided cleanup method, and unregister the service
`endServiceActivity()`	Mark a service of a given type with a given name as being inactive
`checkServiceActive()`	Check whether a service of a specified type and name is active or not

Worth mentioning is that data associated with a service is transient, and will go away when the service becomes inactive. Thus, service-associated data is a great place to keep statistics and other information that other services of the same type may need to know.

It is pretty unusual for a connector to need to define services of its own. The only common exception to this observation is when a connector needs to do throttling. Throttling is common enough that throttling support has been built into the ManifoldCF core classes as of release 1.5. We'll talk about this secondary throttling abstraction later in this chapter.

5.3.3 Locks

A *lock* is a cross-cluster restriction on a thread's access to a specific block of code. You might be thinking, "oh, that's just a critical section", and that's not too far from the truth. But, as always, ManifoldCF's model is somewhat more nuanced than you might at first think.

In this case, the elegance comes from the fact that ManifoldCF classifies access to the protected body of code by an abstract concept involving resources and actions. Each lock has a name, which in essence is the name of some abstract "resource". Each body of code is viewed as acting upon that shared resource. There are three different model ways in which it can act. These are described in table 5.6.

Table 5.6 Locking model code actions and their descriptions

Model action	Description	Excludes

READ	Abstract resource is accessed but not changed	Any other WRITEs, NON-EX WRITEs
NON-EXCLUSIVE WRITE	Abstract resource is written to, but not read, and thus does not have to remain self-consistent	Any other READs, WRITEs
WRITE	Abstract resource is written to and must remain consistent	Any other READs, NON-EX WRITEs, WRITEs

For example, if one thread in one process entered lock "foo" for the activity READ, no other thread in any other ManifoldCF process could enter lock "foo" for the activity WRITE or NON-EXCLUSIVE WRITE. However, other threads could well enter the same "foo" lock with READ being their stated activity.

> **Note** This locking model is just that – it's a model. There may or may not be a correspondence with an actual resource or set of resources in the system. It's best viewed as a tool to permit only the minimum necessary locking restrictions.

LOCK METHODS

The methods used to obtain and release locks come in two distinct flavors. The first flavor has all required lock waiting built into the method. The second, the `NoWait` version of the method, throws a `LockException` if the lock cannot be immediately obtained. It is thus possible to explicitly fail back out of multiple levels of locking and retry all of them, which is a good way to avoid deadlocks. We'll talk about that in more depth in a moment.

Table 5.7 summarizes the locking methods available in `ILockManager`.

Table 5.7 ILockManager lock methods, and their meanings

Method	Meaning
`enterReadLock()`	Enter a read lock for a named resource, and wait until it is available
`enterReadLockNoWait()`	Enter a read lock for a named resource, but do not wait if not available
`leaveReadLock()`	Leave a read lock for a named resource
`enterNonExWriteLock()`	Enter a non-exclusive write lock for a named resource, and wait until it is available
`enterNonExWriteLockNoWait()`	Enter a non-exclusive write lock for a named resource, but do

	not wait if not available
`leaveNonExWriteLock()`	Leave a non-exclusive write lock for a named resource
`enterWriteLock()`	Enter an exclusive write lock for a named resource, and wait until it is available
`enterWriteLockNoWait()`	Enter a write lock for a named resource, but do not wait if not available
`leaveWriteLock()`	Leave a write lock for a named resource
`enterLocks()`	Enter read, non-exclusive write, and exclusive write locks for a specified set of named resources, and wait until they are all available
`enterLocksNoWait()`	Enter read, non-exclusive write, and exclusive write locks for a specified set of named resources, but do not wait until they are available
`leaveLocks()`	Leave read, non-exclusive write, and exclusive write locks for a specified set of named resources
`timedWait()`	Wait a specified amount of time, as part of the back-off process for a `NoWait`-style lock failure

USAGE

Using the lock methods is straightforward. However, it is **critical** to insure that, whenever a lock is thrown, it cannot fail to be released. The best mechanism in Java for doing that is the `try-finally` block. See listing 5.11.

Listing 5.11 Proper usage of implicit-wait lock methods

```
String lockName = "foo";
lockManager.enterReadLock(lockName);                    #A
try                                                     #B
{
  ...                                                   #C
}
finally                                                 #D
{                                                       #D
  lockManager.leaveReadLock(lockName);                  #D
}                                                       #D
```

#A Enter lock
#B Use try-finally to insure cleanup
#C Do stuff that reads "foo"
#D Exit lock in finally clause

Neophyte Java programmers often fail to recognize that, in addition to exceptions that are derived from `java.lang.Exception`, which must be explicitly declared in method

signatures and which thus are explicitly tracked by the Java compiler, there are other derivations of `java.lang.Throwable` possible. This includes the entire hierarchy of `java.lang.Error` exceptions and `java.lang.RuntimeException` exceptions, which the compiler ignores. It is therefore inherently dangerous to presume you've caught all the exceptions that could possibly be thrown in any given block of code, because the compiler will not help you some of these places you may have screwed up. But by using the `try-finally` structure, that risk is removed.

The canonical form for the `NoWait` version of the same lock-protected code can be found in listing 5.12.

Listing 5.12 Proper use of explicit-wait lock methods

```
String lockName = "foo";
while (true)
{
  try
  {
    lockManager.enterReadLockNoWait(lockName);                    #A
    try                                                           #B
    {
      ...                                                         #C
    }
    finally                                                       #D
    {                                                             #D
      lockManager.leaveReadLock(lockName);                        #D
    }                                                             #D
    break;
  }
  catch (LockException e)
  {
    lockManager.timedWait(Random.nextInt(0,60000));               #E
  }
}
```

#A Try to enter the lock
#B If we succeed, use try-finally
#C Do stuff that reads "foo"
#D Exit lock in finally clause
#E If we fail to get the lock, retry later

This is pretty straightforward. All that I've done is make the retry wait be explicit, as opposed to being hidden and implicit. The reasons for wanting to do something like this have mainly to do with deadlock recovery, which we'll talk about in a moment.

There are a couple of other interesting restrictions I should take the time to point out before we move on. First, like Java synchronizers, a thread that already has a specific lock cannot fail to obtain the same lock again. Therefore, code like this will work, and will not deadlock:

```
lockManager.enterReadLock("foo");
try
```

```
{
  lockManager.enterReadLock("foo");
  try
  {
    ...
```

Second, you cannot upgrade a lock by trying to enter it again with a new, more restrictive action. It is illegal to do so; an exception will be thrown if you try it. Thus, you cannot be within a READ lock-protected section for a resource, and attempt to enter a WRITE or NON-EX WRITE section for the same resource.

DEALING WITH DEADLOCK

One cannot discuss locking without discussing deadlock. A deadlock may occur when two or more threads obtain the same locks but in different orders. If both such threads happen to be grabbing locks at the same time, then they may well wind up both holding a subset of the necessary locks, while the other thread holds the other subset, and neither thread can proceed.

Detecting deadlock when it occurs involves obtaining a JVM thread dump. This is a stack trace of all threads currently in existence within the JVM, dumped to the Java process's standard output stream. How you get one of these depends on the operating system you are using; for Linux, you use the command `kill -QUIT` <*process_id*>, while for Windows you must type a CTRL-BREAK in the MSDOS box where the process is running. If you find threads waiting on each other in a cycle, you've got a deadlock problem.

There are, in general, two ways to deal with deadlock. The first way is to avoid having it occur at all. The best way to avoid deadlock is to guarantee that **locks are always obtained in the same order**, no matter which thread is obtaining the locks.

Indeed, `ILockManager` has a pair of lock methods which guarantee precisely that. The `enterLocks()` and `enterLocksNoWait()` methods each can throw multiple locks of multiple kinds, all in a well-defined order. The order happens to be the order of the lock resource names after they have been sorted, within each category. Thus, all WRITE locks are thrown first, in sorted order, then all WRITE NON-EX locks, also in sorted order, then all READ locks, once again in sorted order. Using these methods when multiple locks need to be obtained will always prevent deadlock.

Sometimes, however, it is not possible to obtain all the necessary locks at the same time. It can legitimately happen that one set of locks are needed before it is even possible to know what a second set of locks might be. In such cases, **deadlock cannot be avoided**, it can only be managed.

How can you manage deadlock? The only way to do it is to recognize when it occurs, and when it does, release **all** locks, wait for a while, and try over again. In the case of ManifoldCF's `ILockManager`, that is the main purpose behind the `NoWait` variants of all the locking methods – they allow backoff and retry for complex locking scenarios that cannot be handled through `enterLocks()` alone. Listing 5.13 shows some example code for handling deadlock in this way.

Listing 5.13 Handling deadlock by backoff and retry

```
String outerLockName = "foo";
while (true)                                                          #A
{
  try
  {
    lockManager.enterReadLockNoWait(outerLockName);                   #B
    try
    {
      ...                                                             #C
      String innerLockName = ...                                      #C

      lockManager.enterReadLockNoWait(innerLockName);                 #D
      try
      {
        ...                                                           #E
      }
      finally
      {
        lockManager.leaveReadLock(innerLockName);
      }
    }
    finally
    {
      lockManager.leaveReadLock(outerLockName);
    }
    break;                                                            #F
  }
  catch (LockException e)                                             #G
  {                                                                   #G
    lockManager.timedWait(Random.nextInt(0,60000));                   #G
  }                                                                   #G
}
```

#A Loop until success
#B Enter outer lock
#C Figure out what inner lock we need
#D Attempt to grab inner lock
#E Read both "foo" and inner resource
#F Break out of loop on success
#G Retry randomly if failure

5.3.4 Critical sections

Critical sections are much like locks, except that they always work only within the current process. That is, entering a critical section cannot affect other ManifoldCF processes in any way.

Otherwise, ManifoldCF critical sections and locks function in an utterly identical manner, even down to the models they use. Critical sections do not, however, have `NoWait` variants of each method; this is because heretofore these have been unneeded.

Table 5.8 summarizes the critical section methods and their functions.

Table 5.8 ILockManager critical section methods, and their meanings

Method	Meaning
`enterReadCriticalSection()`	Enter a read critical section for a named resource, and wait until it is available
`leaveReadCriticalSection()`	Leave a read critical section for a named resource
`enterNonExWriteCriticalSection()`	Enter a non-exclusive write critical section for a named resource, and wait until it is available
`leaveNonExWriteCriticalSection()`	Leave a non-exclusive write critical section for a named resource
`enterWriteCriticalSection()`	Enter an exclusive write critical section for a named resource, and wait until it is available
`leaveWriteCriticalSection()`	Leave a write critical section for a named resource
`enterCriticalSections()`	Enter read, non-exclusive write, and exclusive write critical sections for a specified set of named resources, and wait until they are all available
`leaveCriticalSections()`	Leave read, non-exclusive write, and exclusive write critical sections for a specified set of named resources

Usage of critical section methods is similar to usage of lock methods. But since, as of this writing, there are no `NoWait` critical section variants, all critical section waits are implicit.

> **Note** It is important to remember that there is no relationship whatsoever between a lock resource of a specific name, and a critical section resource of the same name. They are entirely distinct from one another.

5.3.5 Global flags

The `ILockManager` abstraction also supports global flags. These are simple Booleans, which have arbitrary global names, whose states can be either `true` or `false`, depending on how they are set. Initially, or after the entire ManifoldCF synchronization system is reset, each such flag is considered to have a value of `false`.

The `ILockManager` methods that support global flags are summarized in table 5.9.

Table 5.9 ILockManager methods that implement global flags

Method	Meaning
`setGlobalFlag()`	Set the state of the specified global flag to be `true`

`clearGlobalFlag()`	Set the state of the specified global flag to be `false`
`checkGlobalFlag()`	Retrieve the state of the specified global flag

USAGE

The usage of global flags is straightforward. In all respects, they behave like Boolean values. They are typically used to pass signals from one process to another. For example, the Agents process is usually shut down with a signal that is sent via a global flag. A typical usage is therefore something like this:

```
lockManager.setGlobalFlag("foo");
```

The process or thread receiving the signal usually receives the signal, and then sets the flag status back to allow the signal to be sent again, as follows:

```
while (true)
{
  if (lockManager.checkGlobalFlag("foo"))
    break;
  ManifoldCF.sleep(1000L);
}
lockManager.clearGlobalFlag("foo");
```

This, of course, means that it is possible to lose a signal between the time it is sent and the time it is reset by the receiving thread. It's possible to use a second global flag to fully synchronize the signal, if that is important for your application. However, I will leave that as an exercise for the student.

5.3.6 Global shared data

Global shared data allows the communication of small amounts of binary data between ManifoldCF processes and threads. Data is named, and each chunk of named data can be set or retrieved.

This mechanism is used primarily by the Cache Manager to describe the expiration time of a cached piece of data. The methods in `ILockManager` that support this kind of synchronization are described in table 5.10.

Table 5.10 ILockManager methods supporting global shared data

Method	Meaning
`readData()`	Read a named chunk of data from the pool of global shared data
`writeData()`	Write a named chunk of data to the pool of global shared data

This functionality is largely used in ManifoldCF to support cache expiration. Other uses may occur to you, but frankly its utility is low where writing connectors is involved.

With the description of global shared data, we've listed all of the functionality that exists in the `ILockManager` abstraction. Next, let's try an example to put it all together, and demonstrate how to use `ILockManager` in real life.

5.3.7 Synchronization example

Our synchronization example is not terribly more advanced than our unique ID generation example, other than it exercises locks and does not generate IDs. However, I hope you will find it useful as an exercise in writing proper synchronization code.

The actual example has a number of reader threads, and a number of writer threads, each throwing locks. The reader threads perform a short sequence of actions on a resource, which only will make sense if the resource is self-consistent. Therefore, this is done inside a section of code that is read-locked using the resource name "my lock". The writer threads update the resource in pieces. These updates therefore occur within a section of code that is write-locked against the same resource name. You can see the entire example for yourself at http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/core_example/src/org/apache/manifoldcf/examples/LockExerciser.java.

The code for the reader thread class is straightforward enough. Please examine listing 5.14.

Listing 5.14 Synchronization example reader thread class

```
protected static class ReaderThread extends Thread
{
  protected String threadName;
  protected MyResource resource;

  public ReaderThread(String threadName, MyResource resource)
  {
    super(threadName);
    setDaemon(true);
    this.threadName = threadName;
    this.resource = resource;                                          #1
  }

  public void run()
  {
    IThreadContext tc = ThreadContextFactory.make();

    try
    {
      ILockManager lockManager = LockManagerFactory.make(tc);          #2

      int counter = 0;

      while (true)                                                     #3
      {                                                                #3
        if (Thread.currentThread().isInterrupted())                    #3
          break;                                                       #3
```

```
            String eventName = threadName + ", event "+
              Integer.toString(counter);

            lockManager.enterReadLock(LOCK_NAME);                          #4
            try                                                            #5
            {                                                              #5
              String theString = resource.firstPart;                       #6
              System.out.println(eventName + ": "+theString+"...");        #6
              Thread.yield();
              System.out.println(eventName + ": "+theString +              #7
                resource.secondPart);                                      #7
              Thread.yield();
            }
            finally                                                        #8
            {                                                              #8
              lockManager.leaveReadLock(LOCK_NAME);                        #8
            }                                                              #8
            counter++;
            Thread.yield();
          }
        }
        catch (ManifoldCFException e)
        {
          if (e.getErrorCode() != ManifoldCFException.INTERRUPTED)
          {
            Logging.misc.error("Thread got unexpected exception: "+
              e.getMessage(),e);
          }
        }
      }
    }
```

#1 Save a reference to the shared resource
#2 Create an ILockManager handle
#3 Loop until interrupted
#4 Enter a read lock for the resource
#5 Begin the try-finally
#6 Grab the first chunk from the resource and print
#7 Grab the second chunk, assemble, and print
#8 Leave the read lock

At #1, we save a reference to the shared resource. The handle to the `ILockManager` interface is created at #2. The code loops until interrupted at #3. At #4, we enter a read lock against the resource name `LOCK_NAME`, which just happens to be "my lock". Immediately thereafter, we begin the `try-finally` block at #5. The code at #6 represents the first activity that is protected; we read the first part of the shared resource here, and print that. At #7, we print both the saved first part and the second part. At #8, we leave the read lock, to repeat the same activities.

The write thread code can be found in listing 5.15.

Listing 5.15 Synchronization example writer thread class

```
protected static class WriterThread extends Thread
{
  protected String threadName;
  protected MyResource resource;
  protected String firstString;
  protected String secondString;

  public WriterThread(String threadName, MyResource resource,
    String firstString, String secondString)
  {
    super(threadName);
    setDaemon(true);
    this.threadName = threadName;
    this.resource = resource;                                          #1
    this.firstString = firstString;                                    #2
    this.secondString = secondString;                                  #2
  }

  public void run()
  {
    IThreadContext tc = ThreadContextFactory.make();

    try
    {
      ILockManager lockManager = LockManagerFactory.make(tc);          #3

      int counter = 0;

      while (true)                                                     #4
      {                                                                #4
        if (Thread.currentThread().isInterrupted())                    #4
          break;                                                       #4

        lockManager.enterWriteLock(LOCK_NAME);                         #5
        try                                                            #6
        {                                                              #6
          resource.firstPart = firstString;                            #7
          Thread.yield();
          resource.secondPart = secondString;                          #8
          Thread.yield();
        }
        finally                                                        #9
        {                                                              #9
          lockManager.leaveWriteLock(LOCK_NAME);                       #9
        }                                                              #9
        counter++;
        Thread.yield();
      }
    }
    catch (ManifoldCFException e)
    {
      if (e.getErrorCode() != ManifoldCFException.INTERRUPTED)
      {
        Logging.misc.error("Thread got unexpected exception: "+
          e.getMessage(),e);
      }
```

```
        }
      }
    }
```

#1 Save a reference to shared resource
#2 Save the first and second strings
#3 Get an ILockManager handle
#4 Loop until interrupted
#5 Enter write lock
#6 Start the try-finally
#7 Make first modification
#8 Make second modification
#9 Leave write lock

This thread's operation is designed to update the shared resource in two pieces, each of which is represented by a string. At #1, a reference to the shared resource is saved. At #2, the two strings that the writer thread will update the resource with are saved. Each writer thread is given a different pair of strings, so we can see which thread did the update. At #3, we create an `ILockManager` handle, then we loop until we're told to shut down at #4. At #5, we enter a write lock for the resource. At #6, we begin the required `try-finally` block. At #7, we make the first of two independent modifications. At #8, we make the second. Finally, at #8, we leave the write lock, and the code loops around and repeats.

BUILDING AND RUNNING THE EXAMPLE

As before, you can run the example to see what it does by checking out the following directory and using `ant` to run it. Unpack it or check it out using the same procedure as used for the unique ID generator example. To run the synchronization example, type:

```
ant run-lock-exerciser
```

You should then see output that includes something like this:

```
[java] Reader#2, event 3: the quick brown fox...
[java] Reader#0, event 3: the quick brown fox...
[java] Reader#1, event 3: the quick brown fox jumps over the lazy dogs
[java] Reader#2, event 3: the quick brown fox jumps over the lazy dogs
[java] Reader#0, event 3: the quick brown fox jumps over the lazy dogs
[java] Reader#2, event 4: oh, be a fine girl...
[java] Reader#0, event 4: oh, be a fine girl...
[java] Reader#2, event 4: oh, be a fine girl, kiss me
[java] Reader#0, event 4: oh, be a fine girl, kiss me
[java] Reader#2, event 5: oh, be a fine girl...
[java] Reader#0, event 5: oh, be a fine girl...
```

You can see that the example is working, because the strings printed out in two parts are always self-consistent. Other things to note are that multiple reader threads access the resource at the same time; this is clearly visible by the interleaving of reader activity in the output.

Now that you understand the fundamentals of locking, you are ready to move on to the next core component, the Cache Manager.

5.4 Cache management services

One of the key core services that ManifoldCF provides is the cache management service. Caching of objects and data has the greatest potential of any system to improve performance. This is perhaps doubly true of a system like ManifoldCF that relies on disk storage (via the database) to perform most of its crawling activities.

The caching model ManifoldCF uses is general, robust, and flexible, but is also somewhat difficult to grasp. This is because of two factors. The first factor is the very generality that makes the caching system so powerful in the first place. The second factor is the fact that not just one way of using the caching system is available; there are at least two viable ways of doing it, each with its own advantages and disadvantages. Nevertheless, it's all part of the same system, and is deserving of description.

> **Note** A good strategy for implementing caching in a connector is to first implement the connector without caching, but keep a caching plan in mind. It is usually straightforward to add caching support later.

5.4.1 The caching abstraction

Like synchronization and locking, there is a central interface which describes all of the cache management functionality in ManifoldCF. This interface can be found at `org.apache.manifoldcf.interfaces.ICacheManager`. You create one of these objects in the following way:

```
ICacheManager cacheManager = CacheManagerFactory.make(threadContext);
```

If you look at the `ICacheManager` interface, the first thing you will notice is that there are two ways of managing caching. The first way I will call the *all-in-one sequence*, and it involves a single cache manager call with a number of supporting objects that create and manage the actual objects being cached. The second way breaks this one method down into smaller steps, making the whole process more flexible. I'll call this style the *piecemeal sequence*.

You may also notice that there are a number of interfaces that this central interface refers to. Unless we describe these carefully, it will be quite difficult to understand exactly what the cache manager is up to. Thus, table 5.11 attempts to summarize the secondary interfaces and their meanings. Each interface also describes which caching style it is intended to work with.

Table 5.11 Cache management interfaces and their meanings

Interface	Caching style	Meaning
`ICacheDescription`	All-in-one, piecemeal	An object implementing this interface describes an object to be cached, e.g. its parameters, so it can be retrieved from the cache when needed

`ICacheClass`	All-in-one, piecemeal	An object that describes the behavior of a class of objects, such as the name of the class, and the maximum number of objects to be cached for the class
`ICacheExecutor`	All-in-one	An object encapsulating the logic needed to create cache-able objects, or process them if they already exist
`ICacheHandle`	Piecemeal	A handle describing a cache session, which consists of object location and potentially object creation
`ICacheCreateHandle`	Piecemeal	A handle associated with the creation of an object which was requested from the cache but which does not yet exist

The overall caching model in both cases is straightforward enough. The cache stores objects, on a per-process basis, whose precise form the Cache Manager could not care less about. Each object it stores is described by an object that implements the `ICacheDescription` interface. The object is used as a hash map lookup key, and thus its `hashCode()` and `equals()` methods must be properly implemented. But, in addition, the `ICacheDescription` interface requires that the object also furnish information that will allow the Cache Manager to throw locks based on the object, and also tell Cache Manager how the object should be expired. The specific requirements are listed in table 5.12.

Table 5.12 The ICacheDescription methods, and their meanings

Method	**Meaning**
`getObjectKeys()`	Returns a set of arbitrary key strings, where explicit invalidation of any string will invalidate the described object
`getCriticalSectionName()`	Returns the name of a unique critical section used to restrict object creation to a single thread per process
`getObjectClass()`	Returns an `ICacheClass` class description object, which describes the class of objects to which this object belongs, for the purposes of controlling the size of the class's cache via an LRU mechanism
`getObjectExpirationTime()`	Returns an expiration time for the object, in milliseconds since January 1, 1970.

How these methods are implemented for an `ICacheDescription` object depends on the caching strategy you would intend to use with it. Let's explore these strategies one at a time.

EXPLICIT INVALIDATION

The first method, explicit invalidation, is what `getObjectKeys()` is all about. It is widely used, and is likely to be your most important invalidation technique. It works as follows.

- Each cached object has one or more invalidation keys associated with it
- These keys are arbitrary strings that you get to make up
- Invalidating any **one** of the keys causes the cached object to be invalidated

So, for example, if you wanted to cache individual rows of a database table, you could have a set of invalidation keys each representing individual rows in a database table, as well as an overall database table invalidation key, just in case you need to do something major to the table as a whole. Then, changing an individual database row might invalidate the specific row's key, while deleting rows according to a complex WHERE clause might invalidate the overall database key. Each cached object, which represents a database row, would then need to be associated with both a row-specific invalidation key and the overall table invalidation key.

LRU INVALIDATION

The second method which can be used to expire cached objects is the LRU, or Least Recently Used, method. The way this method works is by way of the `getObjectClass()` method. This `ICacheDescription` method returns an object which implements `ICacheClass`. This object describes the object's class for the purpose of limiting the number of objects kept in the cache. Specifically, it returns the name of the class of objects to which the object belongs, as well as a maximum number of cached objects for that overall class.

The Cache Manager uses this number to decide whether to flush the least most recently used class member whenever a new object is saved that belongs to the class. Every Cache Manager object access also moves the object to the most-recently-used position in the list of objects kept for that class. This all happens automatically, and is designed to prevent the size of the cache from growing out of bounds.

If you recall our example of caching rows from a database table, you might see how this would be helpful. If the table was known to be small, then there would be little point in limiting the size of the cache. If, however, the table could be large, then it would make sense to limit the number of rows that got cached to some fixed number. All that you would need to do to enforce this limit would be to create a class that implemented `ICacheClass`, and make sure that the `ICacheDescription` object which was intended to describe a table row returned an instance of that class from the `getObjectClass()` method.

TIMED EXPIRATION INVALIDATION

In case LRU is not the right model for your application, the Cache Manager also includes yet another way to expire cached objects. This is via a specific expiration time. The time for expiration is returned by the `getObjectExpirationTime()` method.

The way this works is by the Cache Manager making a decision as to whether a cached object is still valid at the time it is fetched. So, if you think you might be able to use an expiration time to free up cache memory that is being held by old objects, without some extra work, think again – these objects do not in fact get purged unless and until an attempt is made to access the old object, or until the `ICacheManager`'s `expireObjects()` method is called. (That method is indeed called periodically within most long-running ManifoldCF processes.) Nevertheless, the implementation of the timed expiration strategy is perfectly adequate for the purpose it was designed: making sure the cached information is reasonably current, without there being a need for explicit invalidation.

For example, suppose you found that an authority service which accessed a repository was under high load, because the UI that the user interacted with did not merely do a single search every time a page was fetched, but did dozens of searches instead. Each of those searches translates into an invocation of your authority, and thus no matter how well it performs, it would never be able to keep up with the load. It has occurred to you that what you might do in your authority is cache the access results from that authority for the given user. However, since there's no possibility of being informed when the user's information might change, this could not work unless the cached information expired at some point.

Astute readers might note that there is still a problem with this example as it has been stated – namely that cached objects for lots of different users would accumulate in memory over time, no doubt eventually using significant memory. This would indeed be a real problem, if it was not possible to combine the three expiration methods together in whatever way made the most sense. Since it is possible to do this, the solution to the problem is to use timed expiration in conjunction with LRU expiration, thus guaranteeing a limit on the number of cached per-user objects.

> **Note** There is no reason whatsoever that you cannot use multiple techniques for expiring the same cached object. Indeed, much of the power of the Cache Manager comes from its flexibility in allowing the combination of different expiration methods.

THE CACHING TRANSACTION

No matter which caching style is used, there is a fundamental caching transaction involved. This transaction involves the following steps:

1. Create an array of objects that describe the cached objects being sought
2. Create a set of invalidation keys that will be used to flush cache entries as the result of this caching transaction taking place

3. Provide logic which can create the objects being sought, if they are not found
4. Execute the transaction, either in pieces, or as one operation

And here we find a source of confusion. The Cache Manager combines cached object lookup, cached object creation, and invalidation all together in the same flow, because the locking required to perform these activities must cover it all, or there will be race conditions. But that does not mean you need to actually do all of these things at the same time. Indeed, cached object lookup and creation are often distinct from cache invalidation. When we explore the two styles of interacting with Cache Manager, next, we will see examples of both cached object lookup, and cache invalidation.

5.4.2 Piecemeal cache transaction sequence

The piecemeal cache transaction sequence is the most basic caching sequence, and it is also the most flexible. In this sequence, the caching transaction starts by obtaining an `ICacheHandle` transaction handle by using the `enterCache()` method. From there, the flow continues through the transaction, and can take many forms. The transaction ends when the `leaveCache()` method is called. Table 5.13 describes all the methods in `ICacheManager` that make up the piecemeal caching sequence.

Table 5.13 ICacheManager methods involved in the piecemeal caching sequence

Method	Meaning
`enterCache()`	Start a cache transaction, supplying a set of `ICacheDescription` objects, and a set of invalidation keys
`enterCreateSection()`	Begin a lookup/create section within a cache transaction
`lookupObject()`	Lookup an object, given a lookup/create section handle and an `ICacheDescription` description object
`saveObject()`	Save an object into the cache, given a lookup/create section handle and an `ICacheDescription` description object
`leaveCreateSection()`	End a lookup/create section within a cache transaction
`invalidateKeys()`	Invalidate all objects related to the cache transaction's invalidation keys
`leaveCache()`	End a cache transaction

USAGE

In order to use the methods that make up the piecemeal cache management sequence, we need to understand what we are actually trying to do. If our primary goal is looking up a possibly cached object, and creating it if it doesn't yet exist, then the code in listing 5.16 would be a good place to start.

Listing 5.16 Canonical form of code which primarily does cache lookup and object creation

```
ICacheDescription objectDescription =                                    #A
  new MyObjectCacheDescription(...);                                     #A
Object retrievedObject = null;
ICacheHandle transactionHandle = cacheManager.enterCache(                #B
  new ICacheDescription[]{objectDescription},null,null);                 #B
try
{
  ICacheCreateHandle createHandle =                                      #C
    cacheManager.enterCreateSection(transactionHandle);                  #C
  try
  {
    retrievedObject =                                                    #D
      cacheManager.lookupObject(createHandle,objectDescription);         #D
    if (retrievedObject == null)
    {
      ...                                                                #E
      retrievedObject = ...;                                             #E
      cacheManager.saveObject(createHandle,objectDescription,            #F
        retrievedObject);                                                #F
    }
  }
  finally
  {
    cacheManager.leaveCreateSection(createHandle);                       #G
  }
}
finally
{
  cacheManager.leaveCache(transactionHandle);                            #H
}
```

#A Create object description
#B Enter the cache transaction
#C Enter the lookup/create section
#D Look for the object
#E If not found, create the object
#F Save the created object in the cache
#G Safely terminate the create section
#H Safely terminate the cache transaction

Just as with lock management, it is extremely important to use `try-finally` logic during the cache transaction, to ensure that the cache transaction is properly ended. The cache manager needs this because it throws locks under the covers, and needs to exit those locks. That is, effectively, what a cache transaction actually does.

If your primary purpose is invalidating cached objects, then the model is slightly different. In particular, you will still need to execute the code that actually invalidates the cache within the cache transaction. But there is no need to pretend to look for a cached object or objects in order to make that happen. Listing 5.17 has the canonical form of a cache transaction designed solely for invalidation.

Listing 5.17 Canonical form of code which primarily performs cache invalidation

```
StringSet invalidationKeys = new StringSet();                          #A
...                                                                    #A

ICacheHandle transactionHandle = cacheManager.enterCache(              #B
  new ICacheDescription[0],invalidationKeys,null);                     #B
try
{
  ...                                                                  #C

  cacheManager.invalidateKeys(transactionHandle);                      #D
}
finally
{
  cacheManager.leaveCache(transactionHandle);                          #E
}
```

#A Build the set of invalidation keys
#B Enter the cache transaction
#C Do stuff that invalidates the keys
#D Tell cache manager keys are invalid
#E Safely terminate the cache transaction

As you can see, the invalidation transaction is relatively simple in design. The only tricky part is making sure the activity that actually invalidates the cached data occurs within the caching transaction. This is necessary to avoid a situation where cached data is already invalid, but the cache is unaware of that fact, which would represent a race condition.

5.4.3 All-in-one cache transaction sequence

The all-on-one cache transaction sequence is actually implemented using the piecemeal transaction sequence, so there is nothing really different about what it is doing. It does, however, differ in **how** it does things. Specifically, instead of giving the coder freedom to organize their code as they see fit, the all-in-one sequence requires the coder to create a helper object whose methods do all of the situation-specific work. The caching transaction is thus hidden completely when you use this style. You may find it easier to use, or harder, depending on your preferred programming style.

The `ICacheManager` methods that make up the all-in-one cache transaction sequence consist solely of the one method `findObjectsAndExecute()`. This method receives an array of `ICacheDescription` object descriptions, as well as a set of invalidation keys, just like the `enterCache()` method of the piecemeal sequence. But, in addition, the method must also be passed a class that implements the `ICacheExecutor` interface. This class therefore must have implementations for the methods listed in table 5.14.

Table 5.14 Methods of ICacheExecutor objects, and their meanings

Method	Meaning
`create()`	This method is called when the cache manager determines that one or more of the

	objects requested does not exist in the cache, and needs to be created. It receives a list of the `ICacheDescription` objects describing those objects, and is expected to return a corresponding list of `Object` instances.
`exists()`	This method is called for all objects that the cache manager was able to find in its cache, as well as the ones that were safely created by means of the `create()` method above.
`execute()`	This method is called after all `create()` and `exists()` method calls are complete.

You can easily see how the methods in `ICacheExecutor` map back to user-provided sections of code in the canonical cache usage patterns I provided earlier. There's really no more magic than that. The only major difference is in the style of the code needed to use each kind; the all-in-one sequence requires no `try-finally` blocks in order to use it, because there are no cache sessions to leave at the end. I will therefore leave it as an exercise for the student to explore this sequence in more depth, as you see fit.

5.4.4 Caching example

Our caching example attempts to show a relatively straightforward use of the caching infrastructure, both to show how it is done, and to demonstrate one of the particular limitations of the Cache Manager as it exists today. Since the cache manager potentially works across processes, but still needs to be highly efficient, it has been very carefully designed to minimize the amount of signaling between processes. This has an interesting side effect, as we'll see in a moment.

The example starts up a number of threads, of two different kinds. One kind, which I've called an *access thread*, accesses data, potentially using a cached value. The other kind modifies the underlying data upon which the cached value depends, and invalidates the appropriate cache keys as it does this. These threads are of the *changer thread* type. Each changer thread sets the underlying resource to a specific value, which then the access threads should pick up and incorporate into the cache. The example lets all threads run for five seconds, and then terminates them.

Listing 5.18 presents the code for the access thread class, and listing 5.19 shows the code for the changer thread class. You may also find the `ICacheDescription` implementation interesting; listing 5.20 presents that. You can see the complete code at http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/core_example/src/org/apache/manifoldcf/examples/CacheExerciser.java.

Listing 5.18 The access thread class for the caching example

```
protected static class AccessThread extends Thread
{
  protected String threadName;
```

```
protected UnderlyingResource resource;

public AccessThread(String threadName, UnderlyingResource resource)
{
  super(threadName);
  setDaemon(true);
  this.threadName = threadName;
  this.resource = resource;                                          #1
}

public void run()
{
  IThreadContext tc = ThreadContextFactory.make();

  try
  {
    ICacheManager cacheManager = CacheManagerFactory.make(tc);       #2

    ResourceDescription rd = new ResourceDescription(OBJECT_NAME);   #3

    while (true)                                                     #4
    {                                                                #4
      if (Thread.currentThread().isInterrupted())                    #4
        break;                                                       #4

      String eventName =
        new Long(System.currentTimeMillis()).toString() +
        ": " + threadName;

      ICacheHandle transactionHandle = cacheManager.enterCache(      #5
        new ICacheDescription[]{rd},null,null);                      #5
      try
      {
        ICacheCreateHandle createHandle =                            #6
          cacheManager.enterCreateSection(transactionHandle);        #6
        try
        {
          String cachedString =                                      #7
           (String)cacheManager.lookupObject(createHandle,rd);       #7
          if (cachedString == null)
          {
            System.out.println(eventName +
              ": Created cached value and saved it");
            cachedString = resource.resourceValue;                   #8
            cacheManager.saveObject(createHandle,rd,cachedString);   #8
          }
          System.out.println(eventName + ": '"+cachedString+"'");
        }
        finally                                                      #9
        {                                                            #9
          cacheManager.leaveCreateSection(createHandle);             #9
        }                                                            #9
      }
      finally                                                       #10
      {                                                             #10
        cacheManager.leaveCache(transactionHandle);                 #10
```

```
          }                                                          #10

          Thread.yield();
        }
      }
      catch (ManifoldCFException e)
      {
        if (e.getErrorCode() != ManifoldCFException.INTERRUPTED)
        {
          Logging.misc.error("Thread got unexpected exception: "+
            e.getMessage(),e);
        }
      }
    }
  }
```

#1 Keep a reference to the shared underlying resource
#2 Create an ICacheManager instance
#3 Construct the description of the cached object
#4 Repeat until interrupted
#5 Enter a cache transaction
#6 Enter a cache create section
#7 Look for the cached object
#8 Not found: create and save it
#9 Always exit the create section
#10 Always exit the cache transaction

At #1, we record a reference to the underlying resource, which we'll need in order to build our cached value, should it not be in the cache. At #2 we create the `ICacheManager` instance, using the factory class. The description of the cached object we create at #3. At #4 we loop until we're told to stop. Inside the loop, at #5, we enter a cache transaction. Immediately thereafter, we also enter a cache create section at #6. Now, we're able to look for the cached object at #7. If we don't find it, we create it from the underlying resource at #8, and we also print that we've created it, so we can see what is going on. We leave the cache create section and the cache transaction at #9 and #10, respectively.

Listing 5.19 The changer thread class for the caching example

```
protected static class ChangerThread extends Thread
{
  protected String threadName;
  protected UnderlyingResource resource;
  protected String myResourceValue;

  public ChangerThread(String threadName, UnderlyingResource resource,
    String myResourceValue)
  {
    super(threadName);
    setDaemon(true);
    this.threadName = threadName;
    this.resource = resource;
```

```
    this.myResourceValue = myResourceValue;                              #1
  }

  public void run()
  {
    IThreadContext tc = ThreadContextFactory.make();

    try
    {
      ICacheManager cacheManager = CacheManagerFactory.make(tc);         #2

      StringSet invalidationKeys = new StringSet(OBJECT_NAME);           #3

      while (true)                                                       #4
      {                                                                  #4
        if (Thread.currentThread().isInterrupted())                      #4
          break;                                                         #4

        ICacheHandle transactionHandle =                                 #5
          cacheManager.enterCache(new ICacheDescription[0],              #5
          invalidationKeys,null);                                        #5
        try
        {
          resource.resourceValue = myResourceValue;                      #6
          System.out.println(System.currentTimeMillis()+                 #6
            ": Underlying resource changed to '"+                        #6
            myResourceValue+"'");                                        #6
          cacheManager.invalidateKeys(transactionHandle);                #7
        }
        finally                                                          #8
        {                                                                #8
          cacheManager.leaveCache(transactionHandle);                    #8
        }                                                                #8

        Thread.yield();
      }
    }
    catch (ManifoldCFException e)
    {
      if (e.getErrorCode() != ManifoldCFException.INTERRUPTED)
      {
        Logging.misc.error("Thread got unexpected exception: "+
          e.getMessage(),e);
      }
    }
  }
}
```

#1 Keep a reference to the underlying resource
#2 Get an ICacheManager cache manager handle
#3 Construct invalidation keys
#4 Repeat until interrupted
#5 Enter the cache
#6 Change the underlying resource
#7 Invalidate the dependent cached objects
#8 Exit properly

Once again, at #1 we must keep a reference to the underlying resource around, because we will be changing that resource. At #2, we obtain an `ICacheManager` handle. We construct the invalidation keys at #3, which consist only of the cache key representing the one resource we're interested in. At #4, we loop indefinitely until interrupted. In that loop, we enter the cache at #5, and change the underlying resource at #6. We invalidate the specified invalidation keys at #7, and leave the cache at #8.

Listing 5.20 The ICacheDescription implementation for the caching example

```
protected static class ResourceDescription extends                    #1
  org.apache.manifoldcf.core.cachemanager.BaseDescription              #1
{
  protected String objectName;
  protected StringSet objectKeys;

  public ResourceDescription(String objectName)
  {
    super(null);                                                       #2
    this.objectName = objectName;
    this.objectKeys = new StringSet(objectName);
  }

  public StringSet getObjectKeys()                                     #3
  {                                                                    #3
    return objectKeys;                                                 #3
  }                                                                    #3

  public String getCriticalSectionName()                               #4
  {                                                                    #4
    return "exampleobject-"+objectName;                                #4
  }                                                                    #4

  public int hashCode()                                                #5
  {                                                                    #5
    return objectName.hashCode();                                      #5
  }                                                                    #5

  public boolean equals(Object o)                                      #6
  {                                                                    #6
    if (!(o instanceof ResourceDescription))                           #6
      return false;                                                    #6
    return ((ResourceDescription)o).objectName.equals(objectName);     #6
  }                                                                    #6
}
```

#1 Extend the standard base class
#2 The object has no cache class
#3 Return the cache keys for the object
#4 Return a cache critical section name for the object
#5 Always calculate a proper hash code
#6 Always provide a proper equals method

At #1, we extend the base class, which insulates our code from future changes to the interface and also provides standard implementations for the methods. At #2, the base class constructor receives a null object class, so there is no LRU behavior. At #3, the class must return the cache keys for the represented object. In this case, the cache key is a constant based on the object's name. Cache keys are global in scope. At #4, the object must have a critical section name, which also must be globally unique to the object. Always provide a proper hash code method (at #5) and equals method (at #6), or your `ICacheDescription` object is useless.

BUILDING AND RUNNING THE EXAMPLE

To build and run the cache exerciser example, either unpack it or use `svn` as demonstrated for the unique ID example above. Once that's done, you can run the cache example by just typing:

```
ant run-cache-exerciser
```

When you do this, you will see quite a bit of output, most of it having the following general nature:

```
[java] 1296941523758: Access#2: 'I've got a lovely bunch of coconuts'
[java] 1296941523758: Access#1: 'I've got a lovely bunch of coconuts'
[java] 1296941523855: Access#0: 'I've got a lovely bunch of coconuts'
[java] 1296941523855: Access#2: 'I've got a lovely bunch of coconuts'
[java] 1296941523856: Underlying resource changed to 'e=mc^2'
[java] 1296941523856: Access#1: Created cached value and saved it
[java] 1296941523856: Access#1: 'e=mc^2'
[java] 1296941523856: Access#0: Created cached value and saved it
[java] 1296941523856: Access#0: 'e=mc^2'
[java] 1296941523856: Access#2: Created cached value and saved it
[java] 1296941523856: Access#2: 'e=mc^2'
[java] 1296941523857: Access#1: Created cached value and saved it
[java] 1296941523857: Access#1: 'e=mc^2'
[java] 1296941523857: Access#0: 'e=mc^2'
[java] 1296941523858: Access#2: 'e=mc^2'
[java] 1296941523860: Access#1: 'e=mc^2'
```

Here we encounter the issue that I alluded to at the beginning of this section. As you can see, the cache manager seems to be working quite well, except for one case, where we see four "Created cached value" lines in a row, with no change to the underlying resource in between them. What's going on here?

The answer involves the timing of the requests. Cache invalidation, in the current Cache Manager implementation, involves asserting a timestamp representing the time at which the cache key was made invalid. But, since the implementation uses a millisecond clock, it is not always possible to tell when a cached object is indeed invalid, if the object is created during the same millisecond as it was last invalidated. Thus, until the clock ticks over, multiple object creations may take place. This does not affect correctness, but it may impact performance slightly, under the right conditions. The conditions occur when it takes less than a millisecond to create the cached object – which begs the question as to why you'd be using the cache manager in the first place to cache this object.

All of this has been building up to something, and that something is what we will be talking about last: how to use ManifoldCF's database services.

5.5 Using database services

The database component of ManifoldCF is one of the key backbones of the entire system. That level of importance would indicate that we spend a good deal of time exploring the database component's abstraction layer. But at the outset, I should point out that although any connector in ManifoldCF can have its own database tables, it is quite rare for a connector to actually **need** this functionality.

The reason is simple. ManifoldCF provides a rich array of services a connector can use, and thus support is already pretty comprehensive. Nevertheless, there may come a day when you, or a friend, might need your connector to have its very own database table or tables. That's what this section is about.

5.5.1 The database abstraction

The database abstraction in ManifoldCF can be considered from both the caller's perspective and the internal, database-specific, platform support perspective. Since our primary interest is in developing connectors, we're going to confine ourselves to exploring the former.

The interface that you must use when dealing with the database can be found at `org.apache.manifoldcf.core.interfaces.IDBInterface`. You create an instance of this interface using `org.apache.manifoldcf.core.interfaces.DBInterfaceFactory.make()`. You will need the appropriate database name and credentials to use this factory method, but these are easily obtained; we'll be covering that in the example.

If you examine the interface class, you will notice that there's a considerable number of methods which you can use to create, populate, and query database tables. In addition, you may also notice that many of the methods accept cache keys and invalidation keys. That's because database resultset caching is built right into the abstraction – you can choose to cache results of any query, and you can invalidate those results on most operations that would change the database.

For supporting a connector, you'll probably only need a subset of the methods in `IDBInterface`. I've summarized what I think may be useful in table 5.15.

Table 5.15 Subset of IDBInterface methods used by connectors

Method	Meaning
`performInsert()`	Insert a row into a database table.
`performUpdate()`	Update rows in a database table.
`performDelete()`	Delete rows in a database table.

`performCreate()`	Create a database table.
`performAddIndex()`	Add an index to a database table.
`performDrop()`	Drop a database table.
`performQuery()`	Perform a query and return a resultset.
`beginTransaction()`	Begin a transaction.
`endTransaction()`	End a transaction.
`signalRollback()`	Signal that the current transaction should be rolled back when it is ended.
`getSleepAmt()`	Get a random number of milliseconds to sleep for.
`sleepFor()`	Sleep for a specified number of milliseconds.

Most of these operations are pretty self-explanatory, and we'll get a chance to use them in our database example. But first, let's talk a bit about transactions.

DATABASE TRANSACTIONS

Using transactions in a database gives you all sorts of control over the atomicity of operations that you perform. This is important on many levels. You can, for instance, guarantee that if ManifoldCF shuts down suddenly, your data's internal constraints are still met. You can also make sure that multiple threads each see a consistent snapshot of the data. Furthermore, in ManifoldCF, database transactions are also integrated with the Cache Manager, just like database operations are, so you can comfortably use transactions along with caching, and commits and rollbacks all work just as you would expect.

The most basic form of transaction in ManifoldCF looks something like the code in listing 5.21.

Listing 5.21 Canonical form of a ManifoldCF database transaction

```
database.beginTransaction();                                    #A
try                                                             #B
{
  ...                                                           #C
}
catch (ManifoldCFException e)                                   #D
{                                                               #D
  database.signalRollback();                                    #D
  throw e;                                                      #D
}                                                               #D
catch (Error e)                                                 #E
{                                                               #E
  database.signalRollback();                                    #E
  throw e;                                                      #E
}                                                               #E
catch (RuntimeException e)                                      #F
{                                                               #F
```

```
    database.signalRollback();                                          #F
    throw e;                                                            #F
  }                                                                     #F
  finally                                                               #G
  {                                                                     #G
    database.endTransaction();                                          #G
  }                                                                     #G
```

#A Start the transaction
#B Enter the try-catch-finally
#C Do some combination of database changes and reads
#D Signal rollback for every explicit kind of exception
#E Signal rollback for Errors
#F Signal rollback for RuntimeExceptions
#G Always terminate the transaction

As you can see, just as in working with the synchronization component and the cache management component, you have to be very careful how you construct your code so that it is properly resilient against whatever might happen. This design places a significant burden on you, the programmer, to exercise care.

There is another potential complication that we should discuss. It is possible, as we have hinted earlier, that activities that take place within a database transaction may become deadlocked at the database level. When this happens, the database component will throw a `ManifoldCFException` of type `DATABASE_TRANSACTION_ABORT`. This is your signal that you need to back out of all transactions, wait for a time, and try the whole thing all over again. See listing 5.22 for the proper canonical form of this kind of transaction.

Listing 5.22 Canonical form of a database transaction with database abort and retry

```
while (true)                                                            #A
{
  database.beginTransaction();                                          #B
  try
  {
    ...                                                                 #C
    break;                                                              #D
  }
  catch (ManifoldCFException e)
  {
    database.signalRollback();
    if (e.getErrorCode() !=                                             #E
      ManifoldCFException.DATABASE_TRANSACTION_ABORT)                   #E
      throw e;                                                          #E
  }
  catch (Error e)
  {
    database.signalRollback();
    throw e;
  }
  catch (RuntimeException e)
  {
    database.signalRollback();
    throw e;
```

```
    }
    finally
    {
      database.endTransaction();
    }
    database.sleepFor(database.getSleepAmt());                    #F
  }
```

#A Potentially infinite retry loop
#B Start the transaction, as before
#C Do some combination of reads and writes
#D If we make it all the way through, break out of loop
#E Rethrow ManifoldCFException only if not a transaction abort
#F Transaction abort fall-through: sleep a random amount and retry

This is not too much more complex than the non-deadlock-catching canonical transaction I presented in the first place. Although many transactions have no real chance of deadlocking, the fact is that it is sometimes difficult to predict when a database will deadlock, because the database's locking mechanism is complex and often opaque. Worse, the ways in which database deadlock can occur vary from one underlying database to another. For that reason I've come to prefer the latter form for most transactions.

> **Note** Transactions can, of course, be nested. In the case of a deadlock recovery, the recovery wait and retry code **must** be outside of **all** database transactions, or it will do no good.

EXTENDING BASETABLE

It is often a good idea to treat a single database table as a data abstraction, of a sort. In this coding model, you provide logical methods for operating on the database table as an opaque data structure, rather than expose the internal representation.

This technique is used throughout ManifoldCF. It is so common, in fact, that a table abstraction base class has been created to assist with the formation of new table-related manager objects. The base class is located at `org.apache.manifoldcf.core.database.BaseTable`.

The `BaseTable` base class is quite easy to use. You can extend it to add your semantic level functionality to its basic internal database logic. You will find that it has most of the methods available in `IDBInterface`, except that you do not need to explicitly specify the name of the database table in any of them.

5.5.2 Database usage example

Our database example implements a simple key/value store using the `BaseTable` base class. I've tried to keep this fairly simple, but still demonstrate some of the techniques necessary to use the database effectively in ManifoldCF connectors. This necessarily means there are some tradeoffs – for example, there are no transactions in the example whatsoever. However, I've been able to include some commonly-needed functionality, such as usage of integrated caching.

The full example class can be found at http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/core_example/src/org/apache/manifoldcf/examples/DatabaseExample.java. In listing 5.23, I attempt to highlight the important parts.

Listing 5.23 Selected portions of the database example class

```
public DatabaseExample(IThreadContext threadContext)
  throws ManifoldCFException
{
  super(DBInterfaceFactory.make(threadContext,                          #1
      ManifoldCF.getMasterDatabaseName(),                               #1
      ManifoldCF.getMasterDatabaseUsername(),                           #1
      ManifoldCF.getMasterDatabasePassword()),                          #1
    DATABASE_TABLE_NAME);                                               #1
  this.threadContext = threadContext;
}

public void initialize()
  throws ManifoldCFException
{
  Map columnMap = new HashMap();
  columnMap.put(idField,                                                #2
    new ColumnDescription("BIGINT",true,false,null,null,false));        #2
  columnMap.put(keyField,                                               #2
    new ColumnDescription("VARCHAR(255)",false,false,null,null,false));#2
  columnMap.put(valueField,                                             #2
    new ColumnDescription("VARCHAR(255)",false,true,null,null,false)); #2
  performCreate(columnMap,null);                                        #2
  performAddIndex(null,new IndexDescription(true,                       #3
    new String[]{keyField,valueField}));                                #3
}

public String[] findValues(String key)
  throws ManifoldCFException
{
  StringSet cacheKeys = new StringSet(new String[]{
    TABLE_CACHEKEY,makeKeyCacheKey(key)});
  ArrayList params = new ArrayList();                                   #4
  params.add(key);                                                      #4
  IResultSet set = performQuery("SELECT "+valueField+" FROM "+          #4
    getTableName()+" WHERE "+keyField+"=?",params,cacheKeys,null);      #4
  String[] results = new String[set.getRowCount()];
  int i = 0;
  while (i < results.length)
  {
    IResultRow row = set.getRow(i);                                     #5
    results[i] = (String)row.getValue(valueField);                      #6
    i++;
  }
  return results;
}
```

```
public void deleteKeyValues(String key)
  throws ManifoldCFException
{
  ArrayList params = new ArrayList();
  params.add(key);
  StringSet invalidationKeys =                                          #7
    new StringSet(new String[]{makeKeyCacheKey(key)});                  #7
  performDelete("WHERE "+keyField+"=?",params,invalidationKeys);        #8
}

public void addKeyValue(String key, String value)
  throws ManifoldCFException
{
  Map fields = new HashMap();
  fields.put(idField,IDFactory.make(threadContext));
  fields.put(keyField,key);
  fields.put(valueField,value);
  StringSet invalidationKeys =                                          #9
    new StringSet(new String[]{makeKeyCacheKey(key)});                  #9
  performInsert(fields,invalidationKeys);                              #10
}

public void deleteValue(String value)
  throws ManifoldCFException
{
  ArrayList params = new ArrayList();
  params.add(value);
  StringSet invalidationKeys =                                         #11
    new StringSet(new String[]{TABLE_CACHEKEY});                       #11
  performDelete("WHERE "+valueField+"=?",params,invalidationKeys);     #12
}

public void destroy()
  throws ManifoldCFException
{
  performDrop(tableCacheKeySet);                                       #13
}
```

#1 Construct an IDBInterface handle, and pass to Database constructor
#2 Lay out the fields in the table, and create it
#3 Add an appropriate index
#4 Construct a query with the proper cache keys
#5 Get the appropriate row
#6 Get the value field contents
#7 Invalidate everything based on the key's cache key
#8 Perform the deletion
#9 Invalidate everything based on the key's cache key
#10 Perform the insertion
#11 Invalidate everything based on the overall table cache key
#12 Perform the deletion
#13 Drop the database table, invalidating all cached entries

At #1, we construct an `IDBInterface` handle, using the standard procedure to obtain the main database name and credentials, and we pass that to the superclass constructor, along with the name of our database table. At #2, we specify the columns in the table, and

create it. At #3, we add an index to our table as well – this one a unique index on the key column and value column. At #4, we construct a query to obtain the key values for a key. The cache keys we cache this query against consist of both a cache key based on the key name, as well as a cache key designed to cover changes to the entire table. Decoding the results, at #5, we access the data one row at a time, and then read the column values we wish at #6.

The operation to remove all values for a key only invalidates the cache key based on the key name (at #7), and performs the actual deletion at #8. Adding a new key/value pair also invalidates based on the cache key for the key name, at #9, and inserts the new row at #10. Deleting key/value records based on value, on the other hand, invalidates using the overall table's cache key (at #11), because we have no idea what key names might be affected by such an operation. The delete itself takes place at #12. Finally, we drop the table at #13, invalidating all the data cached against the table's cache key in the process.

RUNNING THE EXAMPLE

To run the database example, simply unpack or use `svn` to check out the `core_example` area, as demonstrated for the unique ID example. Then, use `ant` to build and run it, as follows:

```
ant run-database-example
```

If all went well, you should see output that consists, in part, of the following:

```
run-database-example:
     [java] Configuration file successfully read
     [java] key1 values: {value1,value2}
     [java] key2 values: {value1,value3}
     [java] key1 values again: {value1,value2}
     [java] key1 values, after delete: {value2}
     [java] key2 values, after delete: {value3}
     [java] key1 values, after 2nd delete: {}
     [java] key2 values, after 2nd delete: {value3}
```

The meaning of the output will become clear once you have a look at the full example code.

Congratulations! You've successfully written code using each of the core ManifoldCF services!

5.6 Using the throttling abstraction

As we discussed earlier in this chapter, there is one common situation which can arise at a connector level in a multi-process cluster which requires coordination across cluster members. This is the need to enforce throttling constraints, such as connection limits, fetch rate limits, and bandwidth limits.

If you recall, the section of this chapter describing synchronization and locking describes the implementation of a concept called a "service". Services are generic, and allow information to be amalgamated across the entire cluster. But while this is, in principle, sufficient to build a throttling implementation, in practice this implementation is tricky and

involved. So the decision was made in ManifoldCF 1.5 to create a general, service-based, throttling implementation than any connector could use.

The throttling implementation organizes throttles into *throttle groups*. Each throttle group has a type and a name. The type maps pretty well to an individual connector that is trying to apply throttling independently of other connectors. The name is meant to be used for situations where a finer grained breakdown of throttle groups within a single connector is necessary.

The operations available for throttle groups are described in table 5.16.

Table 5.16 Key IThrottleGroups methods

Method	Function
`getThrottleGroups()`	Get the set of throttle groups corresponding to a particular throttle group type
`removeThrottleGroup()`	Remove a throttle group, given a throttle group type and name
`createOrUpdateThrottleGroup()`	Create or update a throttle group, given a type, name, and throttle specification
`obtainConnectionThrottler()`	Given a throttle group type and name, create a connection throttling object

A connection throttling object fundamentally maps to a connection pool. It tracks all outstanding connections, and presumes that there is some attempt being made to pool connections which have been returned from use but have not expired. Therefore, once you have a connection throttling object, your connector code should use that to manage creation and pooling of your actual connections. See table 5.17.

Table 5.17 Key IConnectionThrottler methods

Method	Function
`waitConnectionAvailable()`	Wait until a connection can be created, or taken from the connection pool
`getNewConnectionFetchThrottler()`	Create a new fetch throttler, after waitConnectionAvailable() signals it is okay to do so
`noteReturnedConnection()`	Log that a connection is not needed any longer, and signal whether it should be pooled or destroyed

`noteConnectionReturnedToPool()`	Log that a returned connection has been placed back in the connection pool
`noteConnectionDestroyed()`	Log that a returned connection has been destroyed

The methods implemented for the `IConnectionThrottler` interface have been designed carefully to be generic but nevertheless work properly without race conditions or locks when used as designed. The tricky part here is that `IConnectionThrottler` does not actually implement a connection pool. Instead, it helps you manage your own pool implementation. So be sure to carefully read the Javadoc for `IConnectionThrottler`, though, before using these methods, if you don't want to have to rewrite your code later.

A fetch throttler object, which you obtain using the `getNewConnectionFetchThrottler()` method described above, limits how often you can begin a fetch. Its important methods are listed in table 5.18.

Table 5.18 Important IFetchThrottler methods

Method	Function
`obtainFetchDocumentPermission()`	Wait until allowed to fetch the next document from the connection
`createFetchStream()`	Create a stream throttler object to use to throttle stream data access

Once you have permission to fetch a document, you can create a stream throttler object by means of the `createFetchStream()` method. This method allows you to restrict the rate at which data is fetched. See table 5.19.

Table 5.19 Important IStreamThrottler object methods

Method	Function
`obtainReadPermission()`	Obtain permission to read a specified number of bytes from the stream
`releaseReadPermission()`	Log the fact that a certain number of bytes have been read from the stream
`closeStream()`	Log that the stream has been closed

This is all you need to implement robust throttling of connections, fetches, and bytes.

5.7 Summary

In this chapter, we've had a good time learning how to use the more important components of ManifoldCF, specifically those that will likely be needed if your task is to develop a connector. On the basic side, we've touched on threading, exceptions, and logging. We then learned much about how to use the ManifoldCF components that perform unique ID generation, locking and synchronization, and caching. Finally, we experimented with the ManifoldCF database abstraction.

In the next chapter, we will build on this basic coding knowledge and learn the ground rules for coding connectors, in general. We'll dive into how to make a connector instance survive in a connection pool, and also discuss how to build connector UI's, in general.