CHAPTER 1

Introducing Active Record

One of the first jobs Kevin had as a teenager was as a dishwasher at a local diner. For those of you who aren't familiar with the job, dishwashers are generally at the bottom of the totem pole in most kitchens. If there's a job nobody wants to do, like digging through the trash for a retainer someone left on a plate, the dishwasher is the one who ends up having to do it. As you can imagine, he hated that job. Still, he did learn a lot of good life lessons, and he learned to be a jack-of-all-trades at an early age.

As a developer, you can probably relate to the jack-of-all-trades situation (though we hope you don't have to dig through the trash like Kevin did!). Developers are expected to know everything there is to know about our language of choice, our development and production platforms, our database software, and, of course, our business logic. In reality, that's a lot of stuff, and just completing a simple task often requires changing hats from a developer to a database administrator to a designer to an end-user. Active Record helps free our brains up a little bit by combining some of these roles into one simple skill set—that of Active Record developer.

Since this entire book covers the niche topic of Active Record for Ruby, it's probably safe to assume that you already know at least the very basics of what the Ruby Active Record library is. That is, you've heard that it's an object relational mapping (ORM) library that is the model part of the Rails model, view, controller (MVC) framework and primarily allows for create, read, update, and delete (CRUD) database operations. If nothing else, you got that much information from the back cover of this book!

But maybe you skipped the back cover and just flipped to this section to see if this book is worth buying (it is, and we recommend two copies; we hear it makes a great gift!), or maybe you're like us and hate acronyms, or your eyes just glaze over when you hear many technical terms in a row like that. Whatever the case, we don't feel like this explanation helps people to understand what Active Record really is or what can actually be done with it. So here's our layman's explanation, which we hope is a bit more direct and easier to digest:

> *Active Record is a Ruby library that allows your Ruby programs to transmit data and commands to and from various data stores, which are usually relational databases.*

In even more basic terms, you might say:

> *Active Record allows Ruby to work with databases.*

Admittedly, there's a lot more to Active Record than just this basic explanation, but hopefully, this gives you the core idea of what the Active Record library was designed to accomplish. Throughout the rest of this book, we'll dig into a lot of little tips, tricks, and features that will turn you into a master of Active Record for Ruby. But before we get too deep into the guts of it all, let's lay a little groundwork and cover some of the background of the Active Record library and the concepts it incorporates, just so we're all on the same page at the start.

The Story Behind Active Record

Active Record is actually a design pattern originally published by Martin Fowler in his book *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional, 2002). The now-famous creator of Rails, David Heinemeier Hansson (commonly referred to online and throughout the rest of this book as simply DHH), took the concepts laid out by Mr. Fowler and implemented them as a Ruby library that he also called Active Record.

■ **Note** Since both the design pattern and the Ruby library are called Active Record, it can quickly become confusing which we're referring to throughout this book. Since the majority of this book is specifically written for and about the Active Record library for Ruby, when we refer to something as simply "Active Record," we mean the Active Record library for Ruby. Therefore, when we refer to the Active Record design pattern, we will use the full label "Active Record design pattern."

When DHH released the Rails framework to the public, Active Record was part of the core bundle, and it's now also available as its own Ruby gem.

As is often the case with open source projects, once the initial library was out there, a number of Ruby and Rails contributors took it upon themselves to take the next step so that the library could be used with almost all of the popular database applications. They did this by developing various database-specific adapters for Active Record. Active Record adapters are basically custom implementations of various parts of the Active Record library that abstract the proprietary bits of each database system, such as connection details, so that the Active Record library pretty much works the same regardless of the backend database system you are using. The most popular and widely used of these adapters are now also directly included as part of the library (we'll mention many of the contributors and developers later in this chapter when we cover the specifics of each database adapter for Active Record).

Active Record Mostly Adheres to the ORM Pattern

The core concept of Active Record and other object relational mapping (ORM) libraries is that relational databases can be represented reasonably in object-based code if you simply think of database tables as classes, table rows as objects, and table fields as object attributes. Looking at a quick example will help to explain this concept best, so assume we had something like the following accounts table in some type of database:

```
Accounts table
    ID field (integer; auto-incremented; primary key)
    Username field (text type field)
    Password field (text type field)
```

Our Active Record `Account` class, or model as it's commonly referred to, would look something like this:

```
Class Account < ActiveRecord::Base
end
```

And finally, throughout our Ruby or Rails code, we would create instances of `account` objects like this:

```
# creates a new account object in memory and a new account record in our database
newacc = Account.new
newacc.Username = "Kevin"
newacc.Password = "Marshall"
newacc.save

# creates an Account object in memory from data in Account table with ID of 1
# (equivalent to the ANSI SQL statement of "select * from accounts where ID = 1")
findacc = Account.find(1)

# deletes records from database that have username of "Kevin"
Account.delete("username = 'Kevin'")
```

Don't worry if all this sort of seems like magic at this point—right now, we're simply trying to show you the ORM concept without any clutter. We'll dive into the details of all this stuff and explain all the ins and outs of Active Record syntax in later chapters.

Active Record Is a Different Kind of ORM

Active Record differs from other ORM libraries, such as Java's Hibernate, mostly in the way it's configured or, rather, in the general lack of initial configuration it requires. Out of the box, Active Record makes a number of configuration assumptions, without requiring any outside XML configuration files or mapping details, so nearly everything just works as DHH believed most would expect or want it to—in fact, our previous example showed this was the case and took full advantage of Active Record assumptions. We weren't required to do any additional configuration or set up any special files or instructions. We just opened a text program and typed a few short lines of code, and before you knew it, we had a fully functional Active Record program.

In fact, the lack of configuration and taking advantage of the default assumptions Active Record makes on our behalf is most likely why the previous example felt like magic. Later in the book, we'll go into more detail about configuration and the default assumptions Active Record makes, as well as how to override any of those assumptions whenever you need.

Active Record Is One Part of the MVC Concept

Active Record is probably most famous as being an important part of the Ruby on Rails framework. And if we had to pick one single thing about the Rails framework that we think makes it successful, it would be the fact that it adheres to the MVC design. The concept of MVC is to break code into logical groupings and programs into logical functional groupings. Traditionally, the model section is where the majority of your business logic code would be; the view is where your user interface code would be, and the controller code primarily deals with the communication between the model and view. Rails MVC implementation is a little bit different. With Rails, the model section is generally your Active Record classes and other data-descriptive or data-communication code. The view section remains primarily for the user interface, which tends to be a heavy dose of HTML in most Rails applications. The controller also handles the communication between the models and the views; however, it also tends to host a larger part of the business logic than traditional MVC systems might.

Since we are focusing on Active Record and not Rails throughout this book, we won't spend too much time on MVC concepts or details. From strictly an Active Record developer's point of view, it doesn't really matter where our code is located or how it's sectioned off. But the MVC design is worth knowing about when you plan to build programs of any serious size. And it's especially important to understand where Active Record fits into the picture of the MVC framework when you are building Rails applications.

Active Record Is Primarily Used for CRUD Database Transactions

There are four general tasks you perform when working with databases: creating (C), reading (R), updating (U), and deleting (D) rows of data. As a group, these actions are often referred to as CRUD. Almost all modern applications perform CRUD operations, and Active Record was specifically designed to make CRUD operations easy to write and understand. The following examples display the four basic CRUD operations as you would see them in most Active Record programs:

```
newacc = Account.new(:username => "Kevin")
newacc.save #=> creates the new record in the account table

temp = Account.find(1)
# => selects the record associated with id of 1 from the account table
temp.username = 'Kevin' # => assigns a value to the username attribute of the object
temp.save #=> does the actual update statement applying the changes we just stated.

Account.destroy_all(1) #=> deletes the record in the account table with id of 1
```

Of course, there are a lot more options and ways to do things than the preceding examples show, but these are the most generic, and probably most common, ones you'll see in Active Record applications. In the next chapter, we'll talk about the Active Record CRUD operations and their various options in detail.

The Active Record Library Is Ruby Code

Probably the most important thing to remember when working with Active Record is that in the end, it's all really just Ruby code. This means anything you can do with Ruby objects, such as inheritance, overriding of methods, metaprogramming, and more, also can be done with Active Record objects. True, the object attributes are generally populated with data pulled from a database through SQL statements, and in most cases, the object attribute values are eventually written out to a database through SQL statements. But outside of those two important processes, everything else you do with or to Active Record objects is really done just like you are working with any other Ruby object.

Though the whole idea is to represent database records as objects, it's important to remember that they really are two separate things: Ruby objects and database records. As such, you can (and will) sometimes have your database record in a different state or with a different value than its corresponding Active Record object and its attributes. This is probably most obvious when you are dealing with data validations. When a data validation fails during an attempt to save, your Active Record object attribute will still have the value assigned by your application (which fails validation), but your database record will not have been updated. We talk more about this issue, and data validation in detail, in Chapter 4.

From Active Record Objects to Database Records and Back Again

Even though Active Record objects are really just Ruby objects, when packaged as the Active Record library, they do go through a number of built-in steps or methods each time they are created, accessed, updated, or deleted. Whether you are saving new records, updating existing ones, or simply accessing data with Active Record, there are three general steps to follow:

1. Create an Active Record object.
2. Manipulate or access the attributes of the object.
3. Save the attributes as a record in the database.

As mentioned previously, updating data can be done using the previous steps or with a special `update` call shown in the following example:

```
Account.update(1, "Username = Kevin")
```

Deleting data from a database, on the other hand, is a little bit of a special situation, since you often want your database records to exist long after your Active Record objects have been destroyed or gone out of scope. If we tied the deletion of data from the database to the life cycle of our objects, every time our code was finished executing, our objects would be removed from memory and our data deleted from our database. That would be a very bad thing. Therefore, deleting data is done by special destroy or delete statements—not by simply removing the object from memory. The following example shows one way of deleting the record with a primary key of `1`:

```
Account.delete(1)
```

If it seems like we are glossing over the details of all this, don't worry; we'll break down the specifics of each of these steps throughout different parts of this book. For now, let's just take a peek at the basics of these three steps, so you have a base understanding of how things work.

Creating an Active Record Object

Most often, you create your Active Record objects with a call to the `create` or `new` method. Both of these methods also allow you to set the values of your object's attributes directly, as shown in the following example:

```
example = Account.new(:Account_Name => "Kevin Marshall",
 :Account_Username => "Falicon")
```

The other common way to create an Active Record object is to use one of the various find methods. All of these methods populate the object's attributes from records in the database that matched the search criteria. The following example creates an object that is populated with the data of the record with a primary key of `1`:

```
example = Account.find(1)
```

Again, we will cover all the various details and options of create, new, update, delete, and find methods throughout the following chapters.

Manipulating or Accessing the Attributes of the Object

Once you have an Active Record object, you have the ability to set or get all of its attributes. The attributes are usually directly mapped from the fields of your database table. So for example, if our Account table had an `Account_Username` field, then our `Account` Active Record objects would have a corresponding `Account_Username` attribute. The following example shows one way of directly setting an attribute's value as well as how to access the value of a given attribute:

```
example.Account_Username = "Falicon"
puts "Your username is now #{example.Account_Username}"
```

Saving the Attributes as a Record in the Database

It's important to remember that when you are working with an Active Record object you are really only setting and accessing the attributes of a Ruby object. Your changes are not reflected within your database until you make a call to the `ActiveRecord::Base.save` method.

The `save` method is where most of the real action and power of the Active Record library takes place:

```
Example.save
```

It's this method that has built-in support for things like callbacks, data validations, and many of the other features explained throughout the remainder of this book.

Why Active Record Is a Smart Choice

Active Record is easy to install, simple to write and read, and full-featured object-based code. Out of the box, it comes with support for most all modern database systems, is platform independent, and goes a long way in abstracting the messy details of dealing with various database implementations. All this means that you, as a developer, can focus on learning just one thing, Active Record, to deal with storing and retrieving data from your database. You don't have to worry about learning all the ins and outs of your specific database software, the unique version of SQL it supports, or the related tips and tricks for massaging data in and out of the database. That leaves you more time and energy for coding your real applications.

If you've been reading through this chapter in hopes of deciding if Active Record is worth learning more about, we hope that you are now anxious to dive into the details with us. However, if you aren't yet quite sold on working through the rest of the book, consider the following list of added benefits to the Active Record approach, each of which we will cover in detail throughout the remainder of this book:

- Simplified configuration and default assumptions
- Automated mapping between tables and classes and between columns and attributes
- Associations among objects
- Aggregation of value objects
- Data validations
- Ability to make data records act like lists or trees
- Callbacks
- Observers for the life cycle of Active Record objects
- Inheritance hierarchies
- Transaction support on both the object and database level
- Automatic reflection on columns, associations, and aggregations
- Direct manipulation of data as well as schema objects
- Database abstraction through adapters and a shared connector
- Logging support
- Migration support
- Active Record as an important part of the Ruby on Rails framework
- Active Record as it's integrated in other emerging frameworks like Merb and Camping

This is just a small list of the features of Active Record, but I hope it gives you an idea of just how powerful Active Record can be. Still, before you can take advantage of anything Active Record has to offer, you must first get it installed and configured, so let's get started with that step now.

Installing and Configuring Active Record

One of the primary design goals of Active Record (and Rails for that matter) was to favor, as DHH puts it, "convention over configuration." This means, from a developer's point of view, it should be very quick and simple to install and start to use. A developer should not have to spend hours setting up and learning about all the various configuration options and files before even starting to do some real coding. As you can imagine, this is a lofty goal for any library designer, but it's one that DHH was actually able to achieve! In fact, it's probably the single biggest reason that Active Record (and Rails) is being so quickly adapted by developers around the world. In this chapter, we'll walk you through the very simple three-step process to get Active Record installed for your specific situation.

Since Active Record is really just a collection of Ruby code, it stands to reason that you must first have Ruby correctly installed on your machine. And since Active Record is primarily distributed as a gem, it should be no surprise that you must also have the Ruby Gem system correctly installed on your machine. There are many good books and resources that cover the installation of these requirements, so we won't go into the details of these here and will instead assume that you already have them installed.

■ **Note** If you are looking for more information on installing Ruby or the Ruby Gem system, two good web sites full of Ruby resources are `http://www.rubycentral.com` and `http://www.rubyforge.com`.

Assuming that you do, in fact, have Ruby and the Ruby Gem system installed correctly on your machine, installing Active Record requires just three simple steps:

1. Install the Active Record gem.
2. Depending on the database adapter you intend to use, install the required files or libraries.
3. Supply the adapter-specific connection information to make a connection to the database.

Let's look at each of these steps in a little more detail. When we're finished with this chapter, you'll have Active Record fully installed, and you'll be ready to dive into coding!

Installing the Active Record Gem

You are probably already familiar with the idea of Ruby Gems—a simple system for packaging, distributing, and installing various Ruby libraries. You're probably also already aware that `www.rubyforge.com` is the default remote gem distribution site. So it should be no surprise to learn that Active Record is, in fact, a gem available through the RubyForge.com system and that the most basic command to install the Active Record gem is to simply type **gem install activerecord** at a command line. The gem system should then walk you through any additional steps that are required for installing the library, including installing the Active Support library, which is a Ruby requirement for Active Record.

■ **Note** If you prefer, you can download the Active Record library for local installation from `www.rubyforge.com`. However, it's generally easier and, therefore, recommended that you simply use the remote gem installation procedure described in this section.

Installing Any Additional Required Libraries or Gems

Active Record handles communication between your code and the database through the use of database-specific adapters. Because each of these adapters is unique and specific to the database that it communicates with, each adapter also has unique and varying underlying requirements in addition to those required by the general Active Record library.

Since Active Record is really just Ruby code, you can view the source code at any time. The source code for each Active Record adapter can be found in your Ruby installation directory under the `lib/ruby/gems/1.8/gems/activerecord-1.15.1/lib/active_record/connection_adapters` directory. Looking directly at the source code is the best possible way to get familiar with the real ins and outs of what each adapter actually does and supports. If you're serious about becoming an Active Record expert, I highly recommend taking a peek at the inner workings of each. It's also a great way to see high-level Ruby programming and design in action.

Out of the box, Active Record comes with adapters for connecting to the most popular and commonly used databases currently on the market: DB2, Firebird, FrontBase, MySQL, OpenBase, Oracle, PostgreSQL, SQLite, SQL Server, and Sybase. Let's take a little more detailed look at the specific dependencies of each database adapter:

DB2: The DB2 adapter was written and is currently maintained by Maik Schmidt. The adapter requires the ruby-db2 driver or Ruby DBI with DB2 support to be installed on the machine as well. You can obtain the ruby-db2 library or the Ruby DBI files from `www.rubyforge.org/projects/ruby-dbi`.

Firebird: The Firebird adapter was written and is currently maintained by Ken Kunz. The adapter requires the FireRuby library to be installed on the machine as well. You can install the FireRuby library via the gem command `gem install fireruby`.

FrontBase: The FrontBase adapter does not currently have any author or maintenance information in its source code. The adapter requires the ruby-frontbase library to be installed on the machine as well. You can obtain the ruby-frontbase library via the gem command `gem install ruby-frontbase`.

MySQL: The MySQL adapter does not currently have any author or maintenance information in its source code. The adapter requires the MySQL library to be installed on the machine as well. You can obtain the MySQL library via the gem command `gem install mysql`.

OpenBase: The OpenBase adapter does not currently have any author or maintenance information it in its source code. The adapter requires the OpenBase library to also be installed on the machine. You can obtain the OpenBase library via the gem command `gem install openbase`.

Oracle: The Oracle adapter was originally written by Graham Jenkins and is currently maintained by Michael Schoen. The adapter requires the ruby-oci8 library, which itself requires that the OCI8 API be installed on your machine. The OCI8 API can be installed as part of the Oracle client available via `www.oracle.com`, and the ruby-oci8 library files can be obtained from `www.rubyforge.org/projects/ruby-oci8`.

PostgreSQL: The PostgreSQL adapter does not currently have any author or maintenance information in its source code. The adapter requires the ruby-postgres library to be installed on the machine as well. You can obtain the ruby-postgres library via the gem command gem install `ruby-postgres`.

SQLite: The SQLite adapter was originally written by Luke Holden and was updated for SQLite3 support by Jamis Buck. The adapter requires the sqlite-ruby library for SQLite2 support and the sqlite3-ruby library for SQLite3 support. You can obtain the sqlite-ruby library via the gem command `gem install sqlite-ruby`. You can obtain the sqlite3-ruby library via the gem command `gem install sqlite3-ruby`.

SQLServer: The SQLServer adapter was written by Joey Gibson with updates provided by DeLynn Berry, Mark Imbriaco, Tom Ward, and Ryan Tomayko. The adapter is currently maintained by Tom Ward. The adapter requires the Ruby DBI library and support for either ADO or ODBC drivers be installed on the machine. You can obtain the DBI library from `www.rubyforge.org/projects/ruby-dbi`. If you intend to use the ADO drivers, included in the DBI download should be the file `bdi-0.1.0/lib/dbd/ADO.rb`. Once the DBI library is installed, this `ADO.rb` file should be copied to `your-ruby-install-directory/lib/ruby/site_ruby/1.8/DBD/ADO/` directory. ODBC driver support varies for each operating system and is outside of the scope of this book. Please refer to your specific operating system's documentation for details on properly setting up ODBC driver support.

■ **Note** You will probably need to manually create the `ADO` directory within the `DBD` directory before placing the `ADO.rb` file in it.

Sybase: The Sybase adapter was written and is maintained by John R. Sheets. The adapter requires the Sybase-ctlib library to be installed on the machine as well. You can obtain the Sybase library via `http://raa.ruby-lang.org/project/sybase-ctlib/`.

Supplying the Adapter-Specific Information

The final step before you can start to actually use Active Record is to establish a connection to your specific database. If you are connecting to Active Record through a Rails application, you generally provide these details in a `database.yml` file in your applications `config` directory. You supply these connection details in YAML format. However, the YAML approach is really just Rails syntactic shorthand for calling the `ActiveRecord::Base.establish_connection` method. Since this is a book about Active Record (and not Rails), throughout our examples, we will generally call the `establish_connection` method rather than use the YAML file option.

The `establish_connection` method expects parameters to be passed as hash values, and each adapter has its own set of acceptable parameters. Let's take a look at each situation in

detail. We will also provide an example call of the `establish_connection` method for each adapter.

DB2 Parameters

The minimum DB2 requirements are the adapter and database parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for a DB2 database. The value can be either `db2` or `ibm-db2` for the IBM adapter.

database: The name of the database that you are attempting to connect to.

username: Optional parameter containing the username of the user as whom you wish to connect to the database. The default value is nothing.

password: Optional parameter containing the password of the user as whom you wish to connect to the database. This value is provided in plain text. The default value is nothing.

schema: Optional parameter containing the initial database schema to be set.

The following example shows how to open an Active Record database connection for DB2:

```
ActiveRecord::Base.establish_connection(:adapter => "db2",
 :database => "artest", :username => "kevin", :password => "test")
```

Firebird Parameters

The minimum Firebird requirements are the adapter and database parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for a Firebird database. The value should be `firebird`.

database: The name of the database that you are attempting to connect to. This value can be either an alias of the Firebird database, the full path of the database file, or a full Firebird connection string.

■ **Note** If you provide a full Firebird connection string in the database parameter, you should not specify the host, service, or port parameters separately.

---

username: Optional parameter containing the username of the user as whom you wish to connect to the database. If this value is not provided, the underlying operating system user credentials are used (on supporting platforms).

password: Optional parameter containing the password of the user as whom you wish to connect to the database. This value is provided in plain text. This parameter is required if the username parameter is supplied but should be omitted if the username is not provided.

host: Optional parameter containing the domain name of the machine that hosts your database. You should not provide this parameter if you are providing the full connection information in the database parameter. Some platforms require that you set this to `localhost` when connecting to a local Firebird instance through a database alias.

port: Optional parameter containing the port on which the database is available for connections. This parameter is required only if the database is only available on a nonstandard port and the service parameter is not provided. If the service parameter is provided, this value will not be used.

service: Optional parameter containing the service name. This parameter is required only if the host parameter is set and you are connecting to a nonstandard service.

charset: Optional parameter containing the character set that should be used for this connection. You should refer to your Firebird documentation for the valid values that can be used with this parameter.

The following example shows how to open an Active Record database connection for Firebird:

```
ActiveRecord::Base.establish_connection(:adapter => "firebird",
 :database => "test", :host => "www.yourdbserver.com",
:username => "kevin", :password => "test")
```

FrontBase Parameters

The minimum FrontBase requirements are the adapter, database, and port parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for a FrontBase database. The value should be `frontbase`.

database: The name of the database that you are attempting to connect to.

username: Optional parameter containing the username of the user as whom you wish to connect to the database.

password: Optional parameter containing the password of the user as whom you wish to connect to the database. This value is provided in plain text.

host: Optional parameter containing the domain name of the machine that hosts your database.

The following example shows how to open an Active Record database connection for FrontBase:

```
ActiveRecord::Base.establish_connection(:adapter => "frontbase",
 :database => "test", :host => "www.yourdbserver.com")
```

MySQL Parameters

The minimum MySQL requirements are the adapter and database parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for a MySQL database. The value should be `mysql`.

database: The name of the database that you are attempting to connect to.

username: Optional parameter containing the username of the user as whom you wish to connect to the database.

password: Optional parameter containing the password of the user as whom you wish to connect to the database. This value is provided in plain text.

socket: Optional parameter that contains the socket that should be used to communicate with the MySQL database. If this parameter is omitted, the adapter assumes a value of `/tmp/mysql.sock`.

port: Optional parameter containing the port on which the database is available for connections.

sslkey: Required parameter if you are connecting to a MySQL database via SSL.

sslcert: Required parameter if you are connecting to a MySQL database via SSL.

sslca: Required parameter if you are connecting to a MySQL database via SSL.

sslcapath: Required parameter if you are connecting to a MySQL database via SSL.

sslcipher: Required parameter if you are connecting to a MySQL database via SSL.

The following example shows how to open an Active Record database connection for MySQL:

```
ActiveRecord::Base.establish_connection(:adapter => "mysql", :database => "test",
:username => "kevin", :password => "test")
```

OpenBase Parameters

The minimum OpenBase requirements are the adapter and database parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for an OpenBase database. The value should be `openbase`.

database: The name of the database that you are attempting to connect to.

username: Optional parameter containing the username of the user as whom you wish to connect to the database.

password: Optional parameter containing the password of the user as whom you wish to connect to the database. This value is provided in plain text.

host: Optional parameter containing the domain name of the machine that hosts your database.

The following example shows how to open an Active Record database connection for OpenBase:

```
ActiveRecord::Base.establish_connection(:adapter => "openbase",
:database => "test", :host => www.yourdbserver.com,
:username => "kevin", :password => "test")
```

Oracle Parameters

The minimum Oracle requirements are the adapter and database parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for an Oracle database. The value should be oracle.

database: The name of the database that you are attempting to connect to.

username: Optional parameter containing the username of the user as whom you wish to connect to the database.

password: Optional parameter containing the password of the user as whom you wish to connect to the database. This value is provided in plain text.

The following example shows how to open an Active Record database connection for Oracle:

```
ActiveRecord::Base.establish_connection(:adapter => "oracle",
 :database => "test", :username => "kevin", :password => "test")
```

PostgreSQL Parameters

The minimum PostgreSQL requirements are the adapter and database parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for a PostgreSQL database. The value should be `postgresql`.

database: The name of the database that you are attempting to connect to.

username: Optional parameter containing the username of the user as whom you wish to connect to the database.

password: Optional parameter containing the password of the user as whom you wish to connect to the database. This value is provided in plain text.

port: Optional parameter containing the port that the database is available for connections.

host: Optional parameter containing the domain name of the machine that hosts your database.

min_messages: Optional parameter that allows you to set the `min_message` value within your database for this connection.

schema_search_path: Optional parameter containing a comma-separated list of schema names to use in the schema search path for the connection.

allow_concurrency: Optional parameter that contains either the value `true` or `false`. If the value is set to `true`, the connection uses asynchronous query methods, which will help prevent the Ruby threads from deadlocking. The default value is `false`, which uses blocking query methods.

encoding: Optional parameter that allows you to specify the encoding to use.

The following example shows how to open an Active Record database connection for PostgreSQL:

```
ActiveRecord::Base.establish_connection(:adapter => "postgresql",
 :database => "test", :username => "kevin", :password => "test")
```

SQLite Parameters

The minimum SQLite requirements are the adapter and database parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for a SQLite database. The value should be `sqlite`.

database: The name of the database that you are attempting to connect to.

The following example shows how to open an Active Record database connection for SQLite:

```
ActiveRecord::Base.establish_connection(:adapter => "sqlite", :database => "test")
```

SQL Server Parameters

The minimum SQL Server requirements are the adapter and the database parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for a Microsoft SQL Server database. The value should be `sqlserver`.

mode: Optional parameter containing the mode in which you wish to make the connection. Valid values are `ado` or `odbc`. If this parameter is omitted, the adapter defaults to the ADO mode.

database: The name of the database that you are attempting to connect to.

host: Optional parameter containing the domain name of the machine that hosts your database.

dsn: Required parameter if the mode is `odbc`. This parameter references the name of your data source set up in your ODBC settings.

username: Optional parameter containing the username of the user as whom you wish to connect to the database.

password: Optional parameter containing the password of the user as whom you wish to connect to the database. This value is provided in plain text.

port: Optional parameter containing the port on which the database is available for connections.

autocommit: Optional parameter to turn the `autocommit` feature of SQL Server on or off. Valid values are `true` and `false`. If this parameter is omitted, the adapter defaults to `true`.

The following example shows how to open an Active Record database connection for SQL Server:

```
ActiveRecord::Base.establish_connection(:adapter => "sqlserver",
:database => "test", :username => "kevin", :password => "test",
 :host => "www.yourdbserver.com")
```

Sybase Parameters

The minimum Sybase requirements are the adapter and the database parameters. Here is the complete list of parameters to consider:

adapter: Specifies that this is connection information for a Sybase database. The value should be `sybase`.

database: The name of the database that you are attempting to connect to.

host: Optional parameter containing the domain name of the machine that hosts your database.

username: Optional parameter containing the username of the user as whom you wish to connect to the database.

password: Optional parameter containing the password of the user as whom you wish to connect to the database. This value is provided in plain text.

The following example shows how to open an Active Record database connection for Sybase:

```
ActiveRecord::Base.establish_connection(:adapter => "sybase",
:database => "test", :host => "www.yourdbserver.com",
:usrname => "kevin", :password => "test")
```

Learning More

By design, Active Record abstracts many of the details of each database, leaving the developer free to focus on the details of coding the application. Switching from one backend database to another, from an Active Record view, generally requires little more than changing your connection information. For the most part, Active Record developers are shielded from having to learn the specifics of any one database implementation—or even most of ANSI SQL for that matter.

Still, each database is fundamentally different and will provide varying levels of support for features and data types. Some will readily support triggers, sequences, and stored procedures; others will not. Some will have elegant ways of dealing with CLOB and BLOB data types; others will not. Each ActiveRecord adapter does its best to create a common denominator for

each of these issues, so that nearly all Active Record methods, techniques, and data types are available for each type of database. But as you can imagine, this is a difficult goal to achieve. Databases, like any software application, continue to grow more and more complex and add new features all the time.

With all this in mind, I recommend that you become as familiar as you can with the specific database application you intend to use. I also highly recommend that you learn at least the basics of ANSI SQL. These two chores will help you tremendously throughout your career in debugging and developing even the most advanced Active Record programs. The following list is a rundown of the most common databases available today and some good starting points for learning more about each:

DbB2: DB2 has been around for a very long time, and some even consider it to be the first database product to use SQL. DB2 is a commercial product provided by IBM and comes in a variety of forms for a variety of platforms. For more information about DB2 you should visit `www-306.ibm.com/software/data/db2`.

Firebird: Firebird is a free-of-charge relational database that runs on Linux, Windows, and a variety of Unix platforms. It is based on the source code released by Inprise Corporation on July 25, 2000. For more information and to download Firebird, you should visit `www.firebirdsql.org`.

FrontBase: FrontBase is a relational database primarily designed for Mac OS X. Licenses for FrontBase are now free. For more information, you should visit `www.frontbase.com`.

MySQL: MySQL is an open source relational database developed and primarily maintained by MySQL AB. There are MySQL versions for most all platforms. For more information, you should visit `www.mysql.com`.

OpenBase: OpenBase is a commercial relational database that has been around since 1991. It is provided by OpenBase International and is available for a variety of platforms including Max OS X, Linux, and Microsoft Windows. For more information on OpenBase, you should visit `www.openbase.com`.

Oracle: Oracle is a commercial relational database provided by Oracle Corporation. There are Oracle versions for most all platforms. For more information, you should visit `www.oracle.com`.

PostgreSQL: PostgreSQL is an open source, object-relational database. PostgreSQL is available for various platforms. For more information, you should visit `www.postgresql.org`.

SQLite: SQLite is a public domain C library that implements a SQL database engine. You can run SQLite on most platforms. For more information, you should visit `www.sqlite.org`.

SQL Server: SQL Server is a commercial relational database provided by Microsoft. SQL Server is primarily designed for the Microsoft platform. For more information, you should visit `www.microsoft.com/sql`.

Sybase: Sybase is a commercial relational database provided by Sybase Corporation. Sybase versions are available for a variety of platforms. For more information, you should visit `www.sybase.com`.

Building Your First Active Record Program

This section will walk you through writing your first Active Record program. It will explain the core concepts of Active Record, including the assumptions it makes in order to dramatically simplify development. Finally, we'll begin to explore the ways you can change these assumptions (a topic which we'll dig deeper into later on in the book).

As previously mentioned, Active Record is an ORM library. ORM is a way of persisting objects to and from relational databases. Recall that, with ORM, an object is analogous to a database table, and individual instances of that object are represented as rows in the table. Finally, the individual member variables of an object are represented as columns in the table.

The elements of a standard Active Record program follow:

1. Include or require the Active Record gem.
2. Establish a connection to your database using the appropriate adapter.
3. Define your Active Record classes by extending the `ActiveRecord::Base` class.
4. CRUD away.

Recall the accounts table from earlier in this chapter:

```
Accounts table
    id field (integer; auto-incremented; primary key)
    username field (text type field)
    password field (text type field)
```

We'll use this accounts table in our examples throughout the rest of this chapter.

Your First Example

Below is the source code for your first Ruby program that uses the Active Record library. The program simply establishes a connection, creates an account object, and stores the attributes of that account object in the database as a new record:

```
require "rubygems"
require_gem "activerecord"

ActiveRecord::Base.establish_connection(
  :adapter => "mysql",
  :host => "localhost",
  :username => "project",
  :database => "project_development")

class Account < ActiveRecord::Base
end

account = Account.new
account.username = "cpytel"
account.save
```

This simple Active Record program includes the Active Record gem, which you installed previously. It establishes a connection to the `project_development` database with username project.

Next, the `Account` class is defined. Notice that there is nothing in the class. Our Active Record objects will eventually have stuff in them, but for now, its important to note that no configuration is needed to get up and running with basic functionality. We've merely supplied the database connection parameters.

Finally, we instantiate a new `Account` object, set the username member variable, and save the instance of the object back to the database.

It's possible to merely connect to the database and be up and running because of the assumptions that Active Record is making and because Active Record gets the rest of its configuration from the database itself.

Active Record Assumptions and Conventions

Our first Active Record program example makes full use of Active Record assumptions and coding conventions. This speeds our development, eases our typing workload, and makes our example seem almost magical. Active Record makes the following assumptions:

- It infers database table names based on class names.
- It assumes the existence of certain database columns.

The first assumption of an Active Record class is the table name. In the case of our `Account` class, the table Active Record assumes is `accounts`. It makes this assumption based on the following guidelines:

- The name of the table within the database is the pluralized name of the class defined in your Active Record program. In our experience, this assumption turns out to be one of the large productivity boosts you'll recognize with Active Record once you get used to it, because it enables the developer to gloss over the naming conventions and instead concentrate on the programming aspects.
- The table name is in lowercase. This is important to note because each database may support case in a variety of ways. Since Ruby variables start with lowercase characters and constants start with uppercase characters, Active Record prefers to force all table and column names to lowercase (via a `downcase` method call). In many of the database systems, case does not really matter when referring to a table or column, so the Active Record downcasing should not cause a problem. For the select few in which case is important, Active Record jumps through as many hoops for you as it can to keep its lowercase preference in line with the specific adapter code for that database.
- If the class name includes multiple words that begin with capital letters, the words will be separated by underscores in the table name.

Table 3-1 lists some examples of assumptions Active Record would make based on the guidelines we've just outlined.

Table 3-1. *Examples of Active Record Table Pluralization*

Class Name	Table Name
`Account`	`accounts`
`Person`	`people`
`UserImage`	`user_images`
`Address`	`addresses`
`Currency`	`currencies`
`Mouse`	`mice`

As you can see from the Table 3-1, Active Record is intelligent about pluralizing the class names. In addition, Active Record also assumes that each table has an automatically incremented integer primary key column named `id`.

When an Active Record class is instantiated and any data is accessed within the class, Active Record reads the columns of the table and maps these to the class's attributes. While there aren't formal conventions for the naming of columns, since Active Record only creates an attribute in the Active Record class that matches the name of the column, many of the Ruby and Rails naming conventions are seen in a typical Active Record table, including the liberal doses of underscores.

When Active Record reads the columns of the database table and creates the attribute mappings, it also reads the data types of those columns and makes sensible mappings among the attribute types and the database column types, as you might expect. However, the `boolean` attribute type is a little different for two reasons. First, a `boolean` type is not supported in all databases supported by Active Record. Second, in Ruby only the `false` constant and the value `nil` are considered false. As a workaround, Active Record attribute methods expand the values considered false to include an empty string, `0`, `"0"`, `"false"`, and `"f"`. Conversely, the values `1`, `"1"`, `"true"`, and `"t"` are considered true.

These few assumptions, coupled with the dynamic language features provided by Ruby (such as duck typing), provide a foundation that makes it possible to provide an incredibly powerful, yet straightforward, feature set.

■ **Note** Duck typing is a form of dynamic typing in which the type of an object is not determined strictly by its class but by its capabilities. This term comes from the idea that if it walks like a duck, and quacks like a duck, it must be a duck. You can read more about duck typing at `http://en.wikipedia.org/wiki/Duck_typing`.

Overriding the Assumptions

While staying true to the Active Record way of doing things can free you up to worry about other things during application development, obviously your application may have some constraints that require you to override some of the assumptions that Active Record is making, particularly if you are working with a legacy database.

If you want table names to be singular instead of plural, you can set the configuration parameter `pluralize_table_names`:

```
ActiveRecord::Base.pluralize_table_names = false
```

If, instead, you need to override a table name completely, you specify this in the Active Record class itself. For instance, if our `Accounts` class should persist to a table named `AccountBean`, we would specify the `Account` class as follows:

```
class Account < ActiveRecord::Base
  set_table_name "AccountBean"
end
```

Additionally, if your primary key column is not named simply `id`, you can override this from within the class definition as well:

```
class Account < ActiveRecord::Base
  set_primary_key "accountId"
end
```

If you want to use a primary key other than an automatically incremented integer, you must set the value of the primary key yourself, and you must still use the `id` attribute to do so. Additionally, you should only use the `id` attribute to *set* the primary key. To retrieve the value of the primary key, you must use your overridden attribute name.

For example, if we've overridden the account primary key to be `account_number`, and we want to use a custom key format, our `Account` creation code would need to be as follows:

```
account = Account.new
account.id = "X5476"
account.save
```

And to retrieve the `account_number` of an account, you would use this:

```
puts account.account_number #=> X5476
```

Retrieving Objects from the Database

With the groundwork laid regarding Active Record knowledge about our database, the dynamic nature of Ruby Active Record is able to help us work with our objects. For instance, to retrieve objects from the database we have a core method `find`. If we know the value of the primary key that we want, for instance 1, we can simply call it:

```
Account.find(1)
```

In addition, it is possible to use a feature of Active Record called dynamic finders. These allow you to easily find records by their attribute values. For example, if you wish to find the account with the username equal to `cpytel` you can simply write:

```
Account.find_by_username("cpytel")
```

While dynamic finders are fun magic, let's be sure not to get ahead of ourselves. Using the normal `find` method, the following code would return the same result as the dynamic finder:

```
Account.find(:all, :conditions => ["username  = ?", "cpytel"])
```

■ **Note** A lot of Active Record magic, such as dynamic finders, is made possible by using the Ruby's `method_missing` function; `method_missing` allows you to handle situations when a message is sent to an object for which it doesn't have a method. The method `find_by_username` doesn't exist in the code anywhere, so it is being handled by `method_missing`.

Once we've retrieved an Active Record object, say with

```
account = Account.find_by_username("cpytel")
```

we can delete the associated record from the database by calling this method:

```
account.destroy
```

When you use the destroy method listed here, you are really only executing a SQL delete statement within your database. The record will no longer be available within your database, but your Active Record object, whose attributes were populated with data from that record, will still be available to you as a read-only instance of the object. This object will persist until it goes out of scope within your application or you specifically delete that instance. This turns out to be a handy feature when you want to report on the deletion of data, as the following code snippet shows:

```
account = Account.find(1)
# do a variety of things within your application...
account.destroy
puts "we just deleted the record with id of #{account.id} from the database"
```

We go into more detail on the various CRUD actions you can perform with Active Record in Chapter 2.

Exploring Active Record Relationships

Relationships among objects, that is, when one or more objects are associated with one another, are not only an incredibly important part of the functionality of the Active Record library, but also of any real-world application. There are several types of relationships, and we'll cover them all in detail in Chapter 4.

All configuration options for a relationship occur within the Active Record class definitions themselves. For our `Account` class, we want to add a relationship to a `Role` object, so we can tell what type of account we have on our hands. We start off by manually defining our roles table within our database:

```
Roles table
    id field (integer; auto-incremented; primary key)
    name field (text type field)
    description field (text type field)
```

We want our account class to hold the reference to the account's role, and we want the foreign key (the column in one table that points to the ID of a row in another) to be in the accounts table. So we define this relationship of roles to accounts in our account model with the `belongs_to` method. First, we add our `Role` class definition:

```
class Role < ActiveRecord::Base
end
```

Next, we modify our definition of the `Account` class as follows:

```
class Account < ActiveRecord::Base
  belongs_to :role
end
```

With those new class definitions we now have a unidirectional relationship between `Account` and `Role`. This relationship is unidirectional, because `Account` knows what role it has, but `Role` does not know what `Account` class instances have it.

With this relationship in place, we now have an attribute for the role relationship of our account objects. However, we first need to make sure that we have a role to work with.

Along with the dynamic finder methods we've already seen, Active Record also has a `find_or_create_by_*` dynamic finder. This finder works just like the normal `find_by_*` method, but if a matching object is not found, one will be created for you. We'll use this method to make sure that our desired role exists:

```
admin_role = Role.find_or_create_by_name("Administrator")
```

We can then assign our administrator role to our account:

```
account.role = admin_role
```

Putting the pieces together, we can now show a more complete and realistic example of an Active Record program. Here we set up our connection, define two models that have a one-to-many relationship, and perform a number of basic CRUD operations:

```
require "rubygems"
require_gem "activerecord"

ActiveRecord::Base.establish_connection(
  :adapter => "mysql",
  :host => "localhost",
  :username => "project",
  :database => "project_development")

class Role < ActiveRecord::Base
end

class Account < ActiveRecord::Base
  belongs_to :role
end

admin_role = Role.find_or_create_by_name("Administrator")

account = Account.new
account.username = "cpytel"
account.role = admin_role
account.save
```

```
puts "#{account.username} (#{account.id}) is a(n) #{account.role.name}"
# cpytel (1) is a(n) Administrator

# comment out the following line to avoid deleting the created account
account.destroy
puts "We have just deleted the #{account.username} account!"
```

Them's the Basics!

Believe it or not, in just one chapter, we've introduced you to Active Record and walked you completely through installing and configuration; plus, we've built and explained complete working programs showing the basic CRUD operations. It really is amazing how little you need to do to get started with Active Record!

Of course, there's a lot more to Active Record than just the basics we've covered here (otherwise, this would be a *very* short book!). In the next few chapters, we'll dig into the guts of Active Record and show you how to take full advantage of the Active Record feature set. Before you know it, you'll go beyond building simple CRUD programs and start building full-featured applications with complex business logic seamlessly integrated with your database via Active Record!