In this chapter, you'll see:

- The establish_connection method
- Tables, classes, columns, and attributes
- IDs and relationships
- Create, read, update, and delete operations
- Callbacks and transactions

CHAPTER 20

# Active Record

Active Record is the object-relational mapping (ORM) layer supplied with Rails. It's the part of Rails that implements your application's model.

In this chapter, we'll build on the mapping data to rows and columns that we did in Depot. Then we'll look at using Active Record to manage table relationships and in the process cover create, read, update, and delete operations (commonly referred to in the industry as CRUD methods). Finally, we'll dig into the Active Record object life cycle (including callbacks and transactions).

## Defining Your Data

In Depot, we defined a number of models, including one for an Order. This particular model has a number of attributes, such as an email address of type String. In addition to the attributes that we defined, Rails provided an attribute named id that contains the primary key for the record. Rails also provides several additional attributes, including attributes that track when each row was last updated. Finally, Rails supports relationships between models, such as the relationship between orders and line items.

When you think about it, Rails provides a lot of support for models. Let's examine each in turn.

### Organizing Using Tables and Columns

Each subclass of ApplicationRecord, such as our Order class, wraps a separate database table. By default, Active Record assumes that the name of the table associated with a given class is the plural form of the name of that class. If the class name contains multiple capitalized words, the table name is assumed to have underscores between these words, as shown in the table on page 304.

| Classname | Table Name |
|---|---|
| Order | orders |
| TaxAgency | tax_agencies |
| Batch | batches |
| Diagnosis | diagnoses |
| LineItem | line_items |
| Person | people |
| Datum | data |
| Quantity | quantities |

These rules reflect Rails' philosophy that class names should be singular while the names of tables should be plural.

Although Rails handles most irregular plurals correctly, occasionally you may stumble across one that's incorrect. If you encounter such a case, you can add to Rails' understanding of the idiosyncrasies and inconsistencies of the English language by modifying the inflection file provided:

rails7/depot_u/config/initializers/inflections.rb

```
# Be sure to restart your server when you modify this file.

# Add new inflection rules using the following format. Inflections
# are locale specific, and you may define rules for as many different
# locales as you wish. All of these examples are active by default:
# ActiveSupport::Inflector.inflections(:en) do |inflect|
#   inflect.plural /^(ox)$/i, "\\1en"
#   inflect.singular /^(ox)en/i, "\\1"
#   inflect.irregular "person", "people"
#   inflect.uncountable %w( fish sheep )
# end

# These inflection rules are supported but not enabled by default:
# ActiveSupport::Inflector.inflections(:en) do |inflect|
#   inflect.acronym "RESTful"
# end

ActiveSupport::Inflector.inflections do |inflect|
  inflect.irregular 'tax', 'taxes'
end
```

If you have legacy tables you have to deal with or don't like this behavior, you can control the table name associated with a given model by setting the table_name for a given class:

```
class Sheep < ApplicationRecord
  self.table_name = "sheep"
end
```

**David says:**

**Where Are Our Attributes?**

The notion of a database administrator (DBA) as a separate role from programmer has led some developers to see strict boundaries between code and schema. Active Record blurs that distinction, and no other place is that more apparent than in the lack of explicit attribute definitions in the model.

But fear not. Practice has shown that it makes little difference whether we're looking at a database schema, a separate XML mapping file, or inline attributes in the model. The composite view is similar to the separations already happening in the model-view-controller pattern—just on a smaller scale.

Once the discomfort of treating the table schema as part of the model definition has dissipated, you'll start to realize the benefits of keeping DRY. When you need to add an attribute to the model, you simply have to create a new migration and reload the application.

Taking the "build" step out of schema evolution makes it just as agile as the rest of the code. It becomes much easier to start with a small schema and extend and change it as needed.

Instances of Active Record classes correspond to rows in a database table. These objects have attributes corresponding to the columns in the table. You probably noticed that our definition of class Order didn't mention any of the columns in the orders table. That's because Active Record determines them dynamically at runtime. Active Record reflects on the schema inside the database to configure the classes that wrap tables.

In the Depot application, our orders table is defined by the following migration:

rails7/depot_r/db/migrate/20221207000007_create_orders.rb

```
class CreateOrders < ActiveRecord::Migration[7.0]
  def change
    create_table :orders do |t|
      t.string :name
      t.text :address
      t.string :email
      t.integer :pay_type

      t.timestamps
    end
  end
end
```

Let's use the handy-dandy bin/rails console command to play with this model. First, we'll ask for a list of column names:

```
depot> bin/rails console
Loading development environment (Rails 7.0.4)
3.1.3 :001 > Order.column_names
=> ["id", "name", "address", "email", "pay_type", "created_at", "updated_at"]
```

Then we'll ask for the details of the pay_type column:

```
>> Order.columns_hash["pay_type"]
 =>
#<ActiveRecord::ConnectionAdapters::Column:0x00000001094cc200
 @collation=nil,
 @comment=nil,
 @default=nil,
 @default_function=nil,
 @name="pay_type",
 @null=true,
 @sql_type_metadata=
  #<ActiveRecord::ConnectionAdapters::SqlTypeMetadata:0x00000001094dc178
   @limit=nil,
   @precision=nil,
   @scale=nil,
   @sql_type="integer",
   @type=:integer>>
```

Notice that Active Record has gleaned a fair amount of information about the pay_type column. It knows that it's an integer, it has no default value, it isn't the primary key, and it may contain a null value. Rails obtained this information by asking the underlying database the first time we tried to use the Order class.

The attributes of an Active Record instance generally correspond to the data in the corresponding row of the database table. For example, our orders table might contain the following data:

```
depot> sqlite3 -line db/development.sqlite3 "select * from orders limit 1"
        id = 1
      name = Dave Thomas
   address = 123 Main St
     email = customer@example.com
  pay_type = 0
created_at = 2022-02-14 14:39:12.375458
updated_at = 2022-02-14 14:39:12.375458
```

If we fetched this row into an Active Record object, that object would have seven attributes. The id attribute would be 1 (an Integer), the name attribute would be the string "Dave Thomas", and so on.

We access these attributes using accessor methods. Rails automatically constructs both attribute readers and attribute writers when it reflects on the schema:

```
o = Order.find(1)
puts o.name                    #=> "Dave Thomas"
o.name = "Fred Smith"          # set the name
```

Setting the value of an attribute doesn't change anything in the database—we must save the object for this change to become permanent.

The value returned by the attribute readers is cast by Active Record to an appropriate Ruby type if possible (so, for example, if the database column is a timestamp, a Time object will be returned). If we want to get the raw value of an attribute, we append _before_type_cast to its name, as shown in the following code:

```
Order.first.pay_type                  #=> "Check", a string
Order.first.pay_type_before_type_cast #=> 0, an integer
```

Inside the code of the model, we can use the read_attribute() and write_attribute() private methods. These take the attribute name as a string parameter.

We can see the mapping between SQL types and their Ruby representation in the following table. Decimal and Boolean columns are slightly tricky.

| SQL Type | Ruby Class |
|---|---|
| int, integer | Integer |
| float, double | Float |
| decimal, numeric | BigDecimal |
| char, varchar, string | String |
| interval, date | Date |
| datetime, time | Time |
| clob, blob, text | String |
| boolean | See text |

Rails maps columns with Decimals with no decimal places to Integer objects; otherwise, it maps them to BigDecimal objects, ensuring that no precision is lost.

In the case of Boolean, a convenience method is provided with a question mark appended to the column name:

```
user = User.find_by(name: "Dave")
if user.superuser?
  grant_privileges
end
```

In addition to the attributes we define, there are a number of attributes that either Rails provides automatically or have special meaning.

### Additional Columns Provided by Active Record

A number of column names have special significance to Active Record. Here's a summary:

*created_at, created_on, updated_at, updated_on*
: These are automatically updated with the timestamp of a row's creation or last update. Make sure the underlying database column is capable of receiving a date, datetime, or string. Rails applications conventionally use the _on suffix for date columns and the _at suffix for columns that include a time.

*id*
: This is the default name of a table's primary key column (in Identifying Individual Rows, on page 308).

*xxx_id*
: This is the default name of a foreign key reference to a table named with the plural form of xxx.

*xxx_count*
: This maintains a counter cache for the child table xxx.

Additional plugins, such as acts_as_list,[1] may define additional columns.

Both primary keys and foreign keys play a vital role in database operations and merit additional discussion.

## Locating and Traversing Records

In the Depot application, LineItems have direct relationships to three other models: Cart, Order, and Product. Additionally, models can have indirect relationships mediated by resource objects. The relationship between Orders and Products through LineItems is an example of such a relationship.

All of this is made possible through IDs.

### Identifying Individual Rows

Active Record classes correspond to tables in a database. Instances of a class correspond to the individual rows in a database table. Calling Order.find(1), for

1. https://github.com/rails/acts_as_list

instance, returns an instance of an Order class containing the data in the row with the primary key of 1.

If you're creating a new schema for a Rails application, you'll probably want to go with the flow and let it add the id primary key column to all your tables. But if you need to work with an existing schema, Active Record gives you a way of overriding the default name of the primary key for a table.

For example, we may be working with an existing legacy schema that uses the ISBN as the primary key for the books table.

We specify this in our Active Record model using something like the following:

```
class LegacyBook < ApplicationRecord
  self.primary_key = "isbn"
end
```

Normally, Active Record takes care of creating new primary key values for records that we create and add to the database—they'll be ascending integers (possibly with some gaps in the sequence). However, if we override the primary key column's name, we also take on the responsibility of setting the primary key to a unique value before we save a new row. Perhaps surprisingly, we still set an attribute called id to do this. As far as Active Record is concerned, the primary key attribute is always set using an attribute called id. The primary_key= declaration sets the name of the column to use in the table. In the following code, we use an attribute called id even though the primary key in the database is isbn:

```
book = LegacyBook.new
book.id = "0-12345-6789"
book.title = "My Great American Novel"
book.save
# ...
book = LegacyBook.find("0-12345-6789")
puts book.title    # => "My Great American Novel"
p book.attributes  #=> {"isbn" =>"0-12345-6789",
                   #    "title"=>"My Great American Novel"}
```

Just to make life more confusing, the attributes of the model object have the column names isbn and title—id doesn't appear. When you need to set the primary key, use id. At all other times, use the actual column name.

Model objects also redefine the Ruby id() and hash() methods to reference the model's primary key. This means that model objects with valid IDs may be used as hash keys. It also means that unsaved model objects can't reliably be used as hash keys (because they won't yet have a valid ID).

One final note: Rails considers two model objects as equal (using ==) if they are instances of the same class and have the same primary key. This means that unsaved model objects may compare as equal even if they have different attribute data. If you find yourself comparing unsaved model objects (which is not a particularly frequent operation), you might need to override the == method.

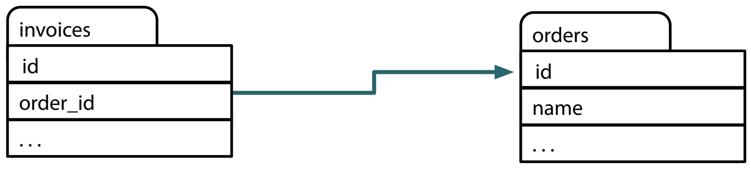
As we'll see, IDs also play an important role in relationships.

## Specifying Relationships in Models

Active Record supports three types of relationship between tables: one-to-one, one-to-many, and many-to-many. You indicate these relationships by adding declarations to your models: has_one, has_many, belongs_to, and the wonderfully named has_and_belongs_to_many.

### One-to-One Relationships

A one-to-one association (or, more accurately, a one-to-zero-or-one relationship) is implemented using a foreign key in one row in one table to reference at most a single row in another table. A *one-to-one* relationship might exist between orders and invoices: for each order there's at most one invoice.



```
class Invoice < ActiveRecord::Base
  belongs_to :order
  # . . .
end
```

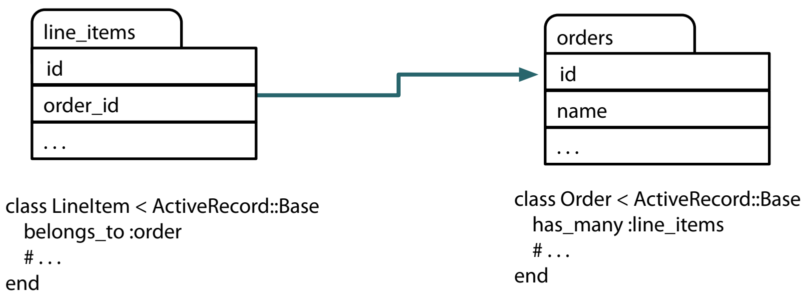
```
class Order < ActiveRecord::Base
  has_one :invoice
  # . . .
end
```

As the example shows, we declare this in Rails by adding a has_one declaration to the Order model and by adding a belongs_to declaration to the Invoice model.

An important rule is illustrated here: the model for the table that contains the foreign key *always* has the belongs_to declaration.

### One-to-Many Relationships

A one-to-many association allows you to represent a collection of objects. For example, an order might have any number of associated line items. In the database, all the line item rows for a particular order contain a foreign key column referring to that order, as shown in the figure on page 311.



```
class LineItem < ActiveRecord::Base
  belongs_to :order
  # . . .
end
```

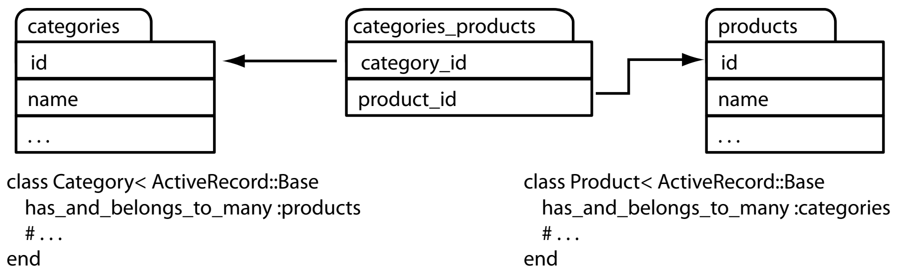
```
class Order < ActiveRecord::Base
  has_many :line_items
  # . . .
end
```

In Active Record, the parent object (the one that logically contains a collection of child objects) uses has_many to declare its relationship to the child table, and the child table uses belongs_to to indicate its parent. In our example, class LineItem belongs_to :order, and the orders table has_many :line_items.

Note that, again, because the line item contains the foreign key, it has the belongs_to declaration.

## Many-to-Many Relationships

Finally, we might categorize our products. A product can belong to many categories, and each category may contain multiple products. This is an example of a *many-to-many* relationship. It's as if each side of the relationship contains a collection of items on the other side.



```
class Category< ActiveRecord::Base
  has_and_belongs_to_many :products
  # . . .
end
```

```
class Product< ActiveRecord::Base
  has_and_belongs_to_many :categories
  # . . .
end
```

In Rails we can express this by adding the has_and_belongs_to_many declaration to both models.

Many-to-many associations are symmetrical—both of the joined tables declare their association with each other using "habtm."

Rails implements many-to-many associations using an intermediate join table. This contains foreign key pairs linking the two target tables. Active Record assumes that this join table's name is the concatenation of the two target table names in alphabetical order. In our example, we joined the table categories to the table products, so Active Record will look for a join table named categories_products.

We can also define join tables directly. In the Depot application, we defined a LineItems join, which joined Products to either Carts or Orders. Defining it ourselves also gave us a place to store an additional attribute, namely, a quantity.

Now that we've covered data definitions, the next thing you would naturally want to do is access the data contained within the database, so let's do that.

## Creating, Reading, Updating, and Deleting (CRUD)

Names such as SQLite and MySQL emphasize that all access to a database is via the Structured Query Language (SQL). In most cases, Rails will take care of this for you, but that's completely up to you. As you'll see, you can provide clauses or even entire SQL statements for the database to execute.

If you're familiar with SQL already, as you read this section take note of how Rails provides places for familiar clauses such as select, from, where, group by, and so on. If you're not already familiar with SQL, one of the strengths of Rails is that you can defer knowing more about such things until you actually need to access the database at this level.

In this section, we'll continue to work with the Order model from the Depot application for an example. We'll be using Active Record methods to apply the four basic database operations: create, read, update, and delete.

### Creating New Rows

Given that Rails represents tables as classes and rows as objects, it follows that we create rows in a table by creating new objects of the appropriate class. We can create new objects representing rows in our orders table by calling Order.new(). We can then fill in the values of the attributes (corresponding to columns in the database). Finally, we call the object's save() method to store the order back into the database. Without this call, the order would exist only in our local memory.

**rails7/e1/ar/new_examples.rb**

```ruby
an_order = Order.new
an_order.name     = "Dave Thomas"
an_order.email    = "dave@example.com"
an_order.address  = "123 Main St"
an_order.pay_type = "check"
an_order.save
```

Active Record constructors take an optional block. If present, the block is invoked with the newly created order as a parameter. This might be useful if you wanted to create and save an order without creating a new local variable.

rails7/e1/ar/new_examples.rb
```
Order.new do |o|
  o.name     = "Dave Thomas"
  # . . .
  o.save
end
```

Finally, Active Record constructors accept a hash of attribute values as an optional parameter. Each entry in this hash corresponds to the name and value of an attribute to be set. This is useful for doing things like storing values from HTML forms into database rows.

rails7/e1/ar/new_examples.rb
```
an_order = Order.new(
  name:     "Dave Thomas",
  email:    "dave@example.com",
  address:  "123 Main St",
  pay_type: "check")
an_order.save
```

Note that in all of these examples we didn't set the id attribute of the new row. Because we used the Active Record default of an integer column for the primary key, Active Record automatically creates a unique value and sets the id attribute as the row is saved. We can subsequently find this value by querying the attribute:

rails7/e1/ar/new_examples.rb
```
an_order = Order.new
an_order.name = "Dave Thomas"
# ...
an_order.save
puts "The ID of this order is #{an_order.id}"
```

The new() constructor creates a new Order object in memory; we have to remember to save it to the database at some point. Active Record has a convenience method, create(), that both instantiates the model object and stores it into the database:

rails7/e1/ar/new_examples.rb
```
an_order = Order.create(
  name:     "Dave Thomas",
  email:    "dave@example.com",
  address:  "123 Main St",
  pay_type: "check")
```

You can pass create() an array of attribute hashes; it'll create multiple rows in the database and return an array of the corresponding model objects:

**rails7/e1/ar/new_examples.rb**

```
orders = Order.create(
  [ { name:     "Dave Thomas",
      email:    "dave@example.com",
      address:  "123 Main St",
      pay_type: "check"
    },
    { name:     "Andy Hunt",
      email:    "andy@example.com",
      address:  "456 Gentle Drive",
      pay_type: "po"
    } ] )
```

The *real* reason that new() and create() take a hash of values is that you can construct model objects directly from form parameters:

```
@order = Order.new(order_params)
```

If you think this line looks familiar, it's because you've seen it before. It appears in orders_controller.rb in the Depot application.

## Reading Existing Rows

Reading from a database involves first specifying which particular rows of data you're interested in—you'll give Active Record some kind of criteria, and it will return objects containing data from the row(s) matching the criteria.

The most direct way of finding a row in a table is by specifying its primary key. Every model class supports the find() method, which takes one or more primary key values. If given just one primary key, it returns an object containing data for the corresponding row (or throws an ActiveRecord::RecordNotFound exception). If given multiple primary key values, find() returns an array of the corresponding objects. Note that in this case a RecordNotFound exception is raised if *any* of the IDs can't be found (so if the method returns without raising an error, the length of the resulting array will be equal to the number of IDs passed as parameters).

```
an_order = Order.find(27)   # find the order with id == 27

# Get a list of product ids from a form, then
# find the associated Products
product_list = Product.find(params[:product_ids])
```

Often, though, you need to read in rows based on criteria other than their primary key value. Active Record provides additional methods enabling you to express more complex queries.

### David says:
### To Raise or Not to Raise?

When you use a finder driven by primary keys, you're looking for a particular record. You expect it to exist. A call to Person.find(5) is based on our knowledge of the people table. We want the row with an ID of 5. If this call is unsuccessful—if the record with the ID of 5 has been destroyed—we're in an exceptional situation. This mandates the raising of an exception, so Rails raises RecordNotFound.

On the other hand, finders that use criteria to search are looking for a *match*. So, Person.where(name: 'Dave').first is the equivalent of telling the database (as a black box) "Give me the first person row that has the name Dave." This exhibits a distinctly different approach to retrieval; we're not certain up front that we'll get a result. It's entirely possible the result set may be empty. Thus, returning nil in the case of finders that search for one row and an empty array for finders that search for many rows is the natural, nonexceptional response.

## SQL and Active Record

To illustrate how Active Record works with SQL, pass a string to the where() method call corresponding to a SQL where clause. For example, to return a list of all orders for Dave with a payment type of "po," we could use this:

```
pos = Order.where("name = 'Dave' and pay_type = 'po'")
```

The result will be an ActiveRecord::Relation object containing all the matching rows, each neatly wrapped in an Order object.

That's fine if our condition is predefined, but how do we handle it when the name of the customer is set externally (perhaps coming from a web form)? One way is to substitute the value of that variable into the condition string:

```
# get the name from the form
name = params[:name]
# DON'T DO THIS!!!
pos = Order.where("name = '#{name}' and pay_type = 'po'")
```

As the comment suggests, this isn't a good idea. Why? It leaves the database wide open to something called a *SQL injection* attack, which the Ruby on Rails Guides[2] describe in more detail. For now, take it as a given that substituting a string from an external source into a SQL statement is effectively the same as publishing your entire database to the whole online world.

2. http://guides.rubyonrails.org/security.html#sql-injection

Instead, the safe way to generate dynamic SQL is to let Active Record handle it. Doing this allows Active Record to create properly escaped SQL, which is immune from SQL injection attacks. Let's see how this works.

If we pass multiple parameters to a where() call, Rails treats the first parameter as a template for the SQL to generate. Within this SQL, we can embed placeholders, which will be replaced at runtime by the values in the rest of the array.

One way of specifying placeholders is to insert one or more question marks in the SQL. The first question mark is replaced by the second element of the array, the next question mark by the third, and so on. For example, we could rewrite the previous query as this:

```
name = params[:name]
pos = Order.where(["name = ? and pay_type = 'po'", name])
```

We can also use named placeholders. We do that by placing placeholders of the form :name into the string and by providing corresponding values in a hash, where the keys correspond to the names in the query:

```
name     = params[:name]
pay_type = params[:pay_type]
pos = Order.where("name = :name and pay_type = :pay_type",
                  pay_type: pay_type, name: name)
```

We can take this a step further. Because params is effectively a hash, we can simply pass it all to the condition. If we have a form that can be used to enter search criteria, we can use the hash of values returned from that form directly:

```
pos = Order.where("name = :name and pay_type = :pay_type",
                  params[:order])
```

We can take this even further. If we pass just a hash as the condition, Rails generates a where clause using the hash keys as column names and the hash values as the values to match. Thus, we could have written the previous code even more succinctly:

```
pos = Order.where(params[:order])
```

Be careful with this latter form of condition: it takes *all* the key-value pairs in the hash you pass in when constructing the condition. An alternative would be to specify which parameters to use explicitly:

```
pos = Order.where(name: params[:name],
                  pay_type: params[:pay_type])
```

Regardless of which form of placeholder you use, Active Record takes great care to quote and escape the values being substituted into the SQL. Use these forms of dynamic SQL, and Active Record will keep you safe from injection attacks.

### Using Like Clauses

We might be tempted to use parameterized like clauses in conditions:

```
# Doesn't work
User.where("name like '?%'", params[:name])
```

Rails doesn't parse the SQL inside a condition and so doesn't know that the name is being substituted into a string. As a result, it will go ahead and add extra quotes around the value of the name parameter. The correct way to do this is to construct the full parameter to the like clause and pass that parameter into the condition:

```
# Works
User.where("name like ?", params[:name]+"%")
```

Of course, if we do this, we need to consider that characters such as percent signs, should they happen to appear in the value of the name parameter, will be treated as wildcards.

### Subsetting the Records Returned

Now that we know how to specify conditions, let's turn our attention to the various methods supported by ActiveRecord::Relation, starting with first() and all().

As you may have guessed, first() returns the first row in the relation. It returns nil if the relation is empty. Similarly, to_a() returns all the rows as an array. ActiveRecord::Relation also supports many of the methods of Array objects, such as each() and map(). It does so by implicitly calling the all() first.

It's important to understand that the query isn't evaluated until one of these methods is used. This enables us to modify the query in a number of ways, namely, by calling additional methods, prior to making this call. Let's look at these methods now.

#### order

SQL doesn't require rows to be returned in any particular order unless we explicitly add an order by clause to the query. The order() method lets us specify the criteria we'd normally add after the order by keywords. For example, the following query would return all of Dave's orders, sorted first by payment type and then by shipping date (the latter in descending order):

```
orders = Order.where(name: 'Dave').
  order("pay_type, shipped_at DESC")
```

### limit

We can limit the number of rows returned by calling the limit() method. Generally when we use the limit method, we'll probably also want to specify the sort order to ensure consistent results. For example, the following returns the first ten matching orders:

```
orders = Order.where(name: 'Dave').
  order("pay_type, shipped_at DESC").
  limit(10)
```

### offset

The offset() method goes hand in hand with the limit() method. It allows us to specify the offset of the first row in the result set that will be returned:

```
# The view wants to display orders grouped into pages,
# where each page shows page_size orders at a time.
# This method returns the orders on page page_num (starting
# at zero).
def Order.find_on_page(page_num, page_size)
  order(:id).limit(page_size).offset(page_num*page_size)
end
```

We can use offset in conjunction with limit to step through the results of a query *n* rows at a time.

### select

By default, ActiveRecord::Relation fetches all the columns from the underlying database table—it issues a select * from... to the database. Override this with the select() method, which takes a string that will appear in place of the * in the select statement.

This method allows us to limit the values returned in cases where we need only a subset of the data in a table. For example, our table of podcasts might contain information on the title, speaker, and date and might also contain a large BLOB containing the MP3 of the talk. If you just wanted to create a list of talks, it would be inefficient to also load the sound data for each row. The select() method lets us choose which columns to load:

```
list = Talk.select("title, speaker, recorded_on")
```

### joins

The joins() method lets us specify a list of additional tables to be joined to the default table. This parameter is inserted into the SQL immediately after the name of the model's table and before any conditions specified by the first parameter. The join syntax is database-specific. The following code returns a list of all line items for the book called *Programming Ruby*:

```
LineItem.select('li.quantity').
  where("pr.title = 'Programming Ruby 1.9'").
  joins("as li inner join products as pr on li.product_id = pr.id")
```

### readonly

The readonly() method causes ActiveRecord::Resource to return Active Record objects that cannot be stored back into the database.

If we use the joins() or select() method, objects will automatically be marked readonly.

### group

The group() method adds a group by clause to the SQL:

```
summary = LineItem.select("sku, sum(amount) as amount").
                   group("sku")
```

### lock

The lock() method takes an optional string as a parameter. If we pass it a string, it should be a SQL fragment in our database's syntax that specifies a kind of lock. With MySQL, for example, a *share mode* lock gives us the latest data in a row and guarantees that no one else can alter that row while we hold the lock. We could write code that debits an account only if there are sufficient funds using something like the following:

```
Account.transaction do
  ac = Account.where(id: id).lock("LOCK IN SHARE MODE").first
  ac.balance -= amount if ac.balance > amount
  ac.save
end
```

If we don't specify a string value or we give lock() a value of true, the database's default exclusive lock is obtained (normally this will be "for update"). We can often eliminate the need for this kind of locking using transactions (discussed starting in Transactions, on page 332).

Databases do more than simply find and reliably retrieve data; they also do a bit of data reduction analysis. Rails provides access to these methods too.

## Getting Column Statistics

Rails has the ability to perform statistics on the values in a column. For example, given a table of products, we can calculate the following:

```
average = Product.average(:price)  # average product price
max     = Product.maximum(:price)
min     = Product.minimum(:price)
total   = Product.sum(:price)
number  = Product.count
```

These all correspond to aggregate functions in the underlying database, but they work in a database-independent manner.

As before, methods can be combined:

```
Order.where("amount > 20").minimum(:amount)
```

These functions aggregate values. By default, they return a single result, producing, for example, the minimum order amount for orders meeting some condition. However, if you include the group method, the functions instead produce a series of results, one result for each set of records where the grouping expression has the same value. For example, the following calculates the maximum sale amount for each state:

```
result = Order.group(:state).maximum(:amount)
puts result  #=> {"TX"=>12345, "NC"=>3456, ...}
```

This code returns an ordered hash. You index it using the grouping element ("TX", "NC", … in our example). You can also iterate over the entries in order using each(). The value of each entry is the value of the aggregation function.

The order and limit methods come into their own when using groups.

For example, the following returns the three states with the highest orders, sorted by the order amount:

```
result = Order.group(:state).
               order("max(amount) desc").
               limit(3)
```

This code is no longer database independent—to sort on the aggregated column, we had to use the SQLite syntax for the aggregation function (max, in this case).

## Scopes

As these chains of method calls grow longer, making the chains themselves available for reuse becomes a concern. Once again, Rails delivers. An Active Record *scope* can be associated with a Proc and therefore may have arguments:

```
class Order < ApplicationRecord
  scope :last_n_days, ->(days) { where('updated < ?' , days) }
end
```

Such a named scope would make finding the worth of last week's orders a snap.

```
orders = Order.last_n_days(7)
```

Simpler scopes may have no parameters at all:

```
class Order < ApplicationRecord
  scope :checks, -> { where(pay_type: :check) }
end
```

Scopes can also be combined. Finding the last week's worth of orders that were paid by check is just as straightforward:

```
orders = Order.checks.last_n_days(7)
```

In addition to making your application code easier to write and easier to read, scopes can make your code more efficient. The previous statement, for example, is implemented as a single SQL query.

ActiveRecord::Relation objects are equivalent to an anonymous scope:

```
in_house = Order.where('email LIKE "%@pragprog.com"')
```

Of course, relations can also be combined:

```
in_house.checks.last_n_days(7)
```

Scopes aren't limited to where conditions; we can do pretty much anything we can do in a method call: limit, order, join, and so on. Just be aware that Rails doesn't know how to handle multiple order or limit clauses, so be sure to use these only once per call chain.

In nearly every case, the methods we've been describing are sufficient. But Rails isn't satisfied with only being able to handle nearly every case, so for cases that require a human-crafted query, there's an API for that too.

## Writing Our Own SQL

Each of the methods we've been looking at contributes to the construction of a full SQL query string. The method find_by_sql() lets our application take full control. It accepts a single parameter containing a SQL select statement (or an array containing SQL and placeholder values, as for find()) and returns an array of model objects (that is potentially empty) from the result set. The attributes in these models will be set from the columns returned by the query. We'd normally use the select * form to return all columns for a table, but this isn't required:

rails7/e1/ar/find_examples.rb

```
orders = LineItem.find_by_sql("select line_items.* from line_items, orders " +
                              " where order_id = orders.id                 " +
                              "   and orders.name = 'Dave Thomas'          ")
```

Only those attributes returned by a query will be available in the resulting model objects. We can determine the attributes available in a model object using the attributes(), attribute_names(), and attribute_present?() methods. The first returns a hash of attribute name-value pairs, the second returns an array of names, and the third returns true if a named attribute is available in this model object:

rails7/e1/ar/find_examples.rb

```
orders = Order.find_by_sql("select name, pay_type from orders")
first = orders[0]
p first.attributes
p first.attribute_names
p first.attribute_present?("address")
```

This code produces the following:

```
{"name"=>"Dave Thomas", "pay_type"=>"check"}
["name", "pay_type"]
false
```

find_by_sql() can also be used to create model objects containing derived column data. If we use the as xxx SQL syntax to give derived columns a name in the result set, this name will be used as the name of the attribute:

rails7/e1/ar/find_examples.rb

```
items = LineItem.find_by_sql("select *,                                  " +
                             "  products.price as unit_price,            " +
                             "  quantity*products.price as total_price,  " +
                             "  products.title as title                  " +
                             " from line_items, products                 " +
                             " where line_items.product_id = products.id ")
li = items[0]
puts "#{li.title}: #{li.quantity}x#{li.unit_price} => #{li.total_price}"
```

As with conditions, we can also pass an array to find_by_sql(), where the first element is a string containing placeholders. The rest of the array can be either a hash or a list of values to be substituted.

```
Order.find_by_sql(["select * from orders where amount > ?",
                   params[:amount]])
```

In the old days of Rails, people frequently resorted to using find_by_sql(). Since then, all the options added to the basic find() method mean you can avoid resorting to this low-level method.

David says:

### But Isn't SQL Dirty?

Ever since developers first wrapped relational databases with an object-oriented layer, they've debated the question of how deep to run the abstraction. Some object-relational mappers seek to eliminate the use of SQL entirely, hoping for object-oriented purity by forcing all queries through an OO layer.

Active Record does not. It was built on the notion that SQL is neither dirty nor bad, just verbose in the trivial cases. The focus is on removing the need to deal with the verbosity in those trivial cases (writing a ten-attribute insert by hand will leave any programmer tired) but keeping the expressiveness around for the hard queries—the type SQL was created to deal with elegantly.

Therefore, you shouldn't feel guilty when you use find_by_sql() to handle either performance bottlenecks or hard queries. Start out using the object-oriented interface for productivity and pleasure and then dip beneath the surface for a close-to-the-metal experience when you need to do so.

### Reloading Data

In an application where the database is potentially being accessed by multiple processes (or by multiple applications), there's always the possibility that a fetched model object has become stale—someone may have written a more recent copy to the database.

To some extent, this issue is addressed by transactional support (which we describe in Transactions, on page 332). However, there'll still be times where you need to refresh a model object manually. Active Record makes this possible with one line of code—call its reload() method, and the object's attributes will be refreshed from the database:

```
stock = Market.find_by(ticker: "RUBY")
loop do
  puts "Price = #{stock.price}"
  sleep 60
  stock.reload
end
```

In practice, reload() is rarely used outside the context of unit tests.

## Updating Existing Rows

After such a long discussion of finder methods, you'll be pleased to know that there's not much to say about updating records with Active Record.

If you have an Active Record object (perhaps representing a row from our orders table), you can write it to the database by calling its save() method. If this object had previously been read from the database, this save will update the existing row; otherwise, the save will insert a new row.

If an existing row is updated, Active Record will use its primary key column to match it with the in-memory object. The attributes contained in the Active Record object determine the columns that will be updated—a column will be updated in the database only if its value has been changed. In the following example, all the values in the row for order 123 can be updated in the database table:

```
order = Order.find(123)
order.name = "Fred"
order.save
```

However, in the following example, the Active Record object contains just the attributes id, name, and paytype—only these columns can be updated when the object is saved. (Note that you have to include the id column if you intend to save a row fetched using find_by_sql().)

```
orders = Order.find_by_sql("select id, name, pay_type from orders where id=123")
first = orders[0]
first.name = "Wilma"
first.save
```

In addition to the save() method, Active Record lets us change the values of attributes and save a model object in a single call to update():

```
order = Order.find(321)
order.update(name: "Barney", email: "barney@bedrock.com")
```

The update() method is most commonly used in controller actions where it merges data from a form into an existing database row:

```
def save_after_edit
  order = Order.find(params[:id])
  if order.update(order_params)
    redirect_to action: :index
  else
    render action: :edit
  end
end
```

We can combine the functions of reading a row and updating it using the class methods update() and update_all(). The update() method takes an id parameter and a set of attributes. It fetches the corresponding row, updates the given attributes, saves the result to the database, and returns the model object.

```
order = Order.update(12, name: "Barney", email: "barney@bedrock.com")
```

We can pass update() an array of IDs and an array of attribute value hashes, and it will update all the corresponding rows in the database, returning an array of model objects.

Finally, the update_all() class method allows us to specify the set and where clauses of the SQL update statement. For example, the following increases the prices of all products with *Java* in their title by 10 percent:

```
result = Product.update_all("price = 1.1*price", "title like '%Java%'")
```

The return value of update_all() depends on the database adapter; most (but not Oracle) return the number of rows that were changed in the database.

### save, save!, create, and create!

It turns out that there are two versions of the save and create methods. The variants differ in the way they report errors.

- save returns true if the record was saved; it returns nil otherwise.
- save! returns true if the save succeeded; it raises an exception otherwise.
- create returns the Active Record object regardless of whether it was successfully saved. You'll need to check the object for validation errors if you want to determine whether the data was written.
- create! returns the Active Record object on success; it raises an exception otherwise.

Let's look at this in a bit more detail.

Plain old save() returns true if the model object is valid and can be saved:

```
if order.save
  # all OK
else
  # validation failed
end
```

It's up to us to check on each call to save() to see that it did what we expected. The reason Active Record is so lenient is that it assumes save() is called in the context of a controller's action method and the view code will be presenting any errors back to the end user. And for many applications, that's the case.

But if we need to save a model object in a context where we want to make sure to handle all errors programmatically, we should use save!(). This method raises a RecordInvalid exception if the object could not be saved:

```
begin
  order.save!
rescue RecordInvalid => error
  # validation failed
end
```

## Deleting Rows

Active Record supports two styles of row deletion. First, it has two class-level methods, delete() and delete_all(), that operate at the database level. The delete() method takes a single ID or an array of IDs and deletes the corresponding row(s) in the underlying table. delete_all() deletes rows matching a given condition (or all rows if no condition is specified). The return values from both calls depend on the adapter but are typically the number of rows affected. An exception is not thrown if the row doesn't exist prior to the call.

```
Order.delete(123)
User.delete([2,3,4,5])
Product.delete_all(["price > ?", @expensive_price])
```

The various destroy methods are the second form of row deletion provided by Active Record. These methods all work via Active Record model objects.

The destroy() instance method deletes from the database the row corresponding to a particular model object. It then freezes the contents of that object, preventing future changes to the attributes.

```
order = Order.find_by(name: "Dave")
order.destroy
# ... order is now frozen
```

There are two class-level destruction methods: destroy() (which takes an ID or an array of IDs) and destroy_all() (which takes a condition). Both methods read the corresponding rows in the database table into model objects and call the instance-level destroy() method of those objects. Neither method returns anything meaningful.

```
Order.destroy_all(["shipped_at < ?", 30.days.ago])
```

Why do we need both the delete and destroy class methods? The delete methods bypass the various Active Record callback and validation functions, while the destroy methods ensure that they're all invoked. In general, it's better to use the destroy methods if you want to ensure that your database is consistent according to the business rules defined in your model classes.

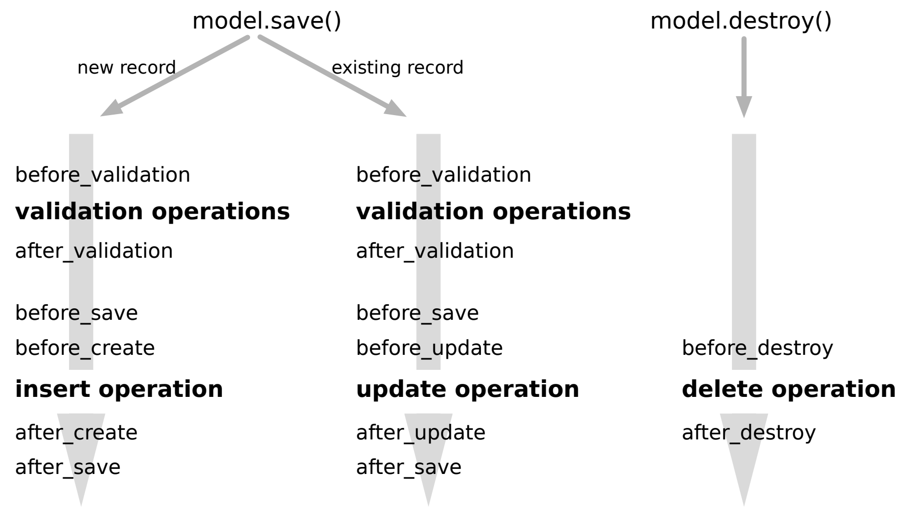
We covered validation in Chapter 7, Task B: Validation and Unit Testing, on page 85. We cover callbacks next.

## Participating in the Monitoring Process

Active Record controls the life cycle of model objects—it creates them, monitors them as they're modified, saves and updates them, and watches sadly as they're destroyed. Using callbacks, Active Record lets our code participate in this monitoring process. We can write code that gets invoked at any significant event in the life of an object. With these callbacks we can perform complex validation, map column values as they pass in and out of the database, and even prevent certain operations from completing.

Active Record defines sixteen callbacks. Fourteen of these form before-after pairs and bracket some operation on an Active Record object. For example, the before_destroy callback will be invoked just before the destroy() method is called, and after_destroy will be invoked after. The two exceptions are after_find and after_initialize, which have no corresponding before_*xxx* callback. (These two callbacks are different in other ways too, as we'll see later.)

In the following figure we can see how Rails wraps the sixteen paired callbacks around the basic create, update, and destroy operations on model objects. Perhaps surprisingly, the before and after validation calls are not strictly nested.



The before_validation and after_validation calls also accept the on: :create or on: :update parameter, which will cause the callback to be called only on the selected operation.

In addition to these sixteen calls, the after_find callback is invoked after any find operation, and after_initialize is invoked after an Active Record model object is created.

To have your code execute during a callback, you need to write a handler and associate it with the appropriate callback.

We have two basic ways of implementing callbacks.

The preferred way to define a callback is to declare handlers. A handler can be either a method or a block. You associate a handler with a particular event using class methods named after the event. To associate a method, declare it as private or protected, and specify its name as a symbol to the handler declaration. To specify a block, simply add it after the declaration. This block receives the model object as a parameter:

```ruby
class Order < ApplicationRecord
  before_validation :normalize_credit_card_number
  after_create do |order|
    logger.info "Order #{order.id} created"
  end
  protected
  def normalize_credit_card_number
    self.cc_number.gsub!(/[-\s]/, '')
  end
end
```

You can specify multiple handlers for the same callback. They will generally be invoked in the order they're specified unless a handler thows :abort, in which case the callback chain is broken early.

Alternately, you can define the callback instance methods using callback objects, inline methods (using a proc), or inline eval methods (using a string). See the online documentation for more details.[3]

## Grouping Related Callbacks Together

If you have a group of related callbacks, it may be convenient to group them into a separate handler class. These handlers can be shared between multiple models. A handler class is simply a class that defines callback methods (before_save(), after_create(), and so on). Create the source files for these handler classes in app/models.

3. http://api.rubyonrails.org/classes/ActiveRecord/Callbacks.html#label-Types+of+callbacks

In the model object that uses the handler, you create an instance of this handler class and pass that instance to the various callback declarations. A couple of examples will make this clearer.

If our application uses credit cards in multiple places, we might want to share our normalize_credit_card_number() method across multiple models. To do that, we'd extract the method into its own class and name it after the event we want it to handle. This method will receive a single parameter, the model object that generated the callback:

```
class CreditCardCallbacks

  # Normalize the credit card number
  def before_validation(model)
    model.cc_number.gsub!(/[-\s]/, '')
  end
end
```

Now, in our model classes, we can arrange for this shared callback to be invoked:

```
class Order < ApplicationRecord
  before_validation CreditCardCallbacks.new
  # ...
end

class Subscription < ApplicationRecord
  before_validation CreditCardCallbacks.new
  # ...
end
```

In this example, the handler class assumes that the credit card number is held in a model attribute named cc_number; both Order and Subscription would have an attribute with that name. But we can generalize the idea, making the handler class less dependent on the implementation details of the classes that use it.

For example, we could create a generalized encryption and decryption handler. This could be used to encrypt named fields before they're stored in the database and to decrypt them when the row is read back. You could include it as a callback handler in any model that needed the facility.

The handler needs to encrypt a given set of attributes in a model just before that model's data is written to the database. Because our application needs to deal with the plain-text versions of these attributes, it arranges to decrypt them again after the save is complete. It also needs to decrypt the data when a row is read from the database into a model object. These requirements mean we have to handle the before_save, after_save, and after_find events. Because we need to decrypt the database row both after saving and when we find a new row, we can save code by aliasing the after_find() method to after_save()—the same method will have two names:

rails7/e1/ar/encrypter.rb

```
class Encrypter
  # We're passed a list of attributes that should
  # be stored encrypted in the database
  def initialize(attrs_to_manage)
    @attrs_to_manage = attrs_to_manage
  end

  # Before saving or updating, encrypt the fields using the NSA and
  # DHS approved Shift Cipher
  def before_save(model)
    @attrs_to_manage.each do |field|
      model[field].tr!("a-z", "b-za")
    end
  end

  # After saving, decrypt them back
  def after_save(model)
    @attrs_to_manage.each do |field|
      model[field].tr!("b-za", "a-z")
    end
  end

  # Do the same after finding an existing record
  alias_method :after_find, :after_save
end
```

This example uses trivial encryption—you might want to beef it up before using this class for real.

We can now arrange for the Encrypter class to be invoked from inside our orders model:

```
require "encrypter"
class Order < ApplicationRecord
  encrypter = Encrypter.new([:name, :email])
  before_save encrypter
  after_save  encrypter
  after_find  encrypter
protected
  def after_find
  end
end
```

We create a new Encrypter object and hook it up to the events before_save, after_save, and after_find. This way, just before an order is saved, the method before_save() in the encrypter will be invoked, and so on.

So why do we define an empty after_find() method? Remember that we said that for performance reasons after_find and after_initialize are treated specially. One of the consequences of this special treatment is that Active Record won't know to call an after_find handler unless it sees an actual after_find() method in the model class. We have to define an empty placeholder to get after_find processing to take place.

This is all very well, but every model class that wants to use our encryption handler would need to include some eight lines of code, just as we did with our Order class. We can do better than that. We'll define a helper method that does all the work and make that helper available to all Active Record models. To do that, we'll add it to the ApplicationRecord class:

rails7/e1/ar/encrypter.rb

```
class ApplicationRecord < ActiveRecord::Base
  self.abstract_class = true

  def self.encrypt(*attr_names)
    encrypter = Encrypter.new(attr_names)

    before_save encrypter
    after_save  encrypter
    after_find  encrypter

    define_method(:after_find) { }
  end
end
```

Given this, we can now add encryption to any model class's attributes using a single call:

```
class Order < ApplicationRecord
  encrypt(:name, :email)
end
```

A small driver program lets us experiment with this:

```
o = Order.new
o.name = "Dave Thomas"
o.address = "123 The Street"
o.email   = "dave@example.com"
o.save
puts o.name

o = Order.find(o.id)
puts o.name
```

On the console, we see our customer's name (in plain text) in the model object:

```
ar> ruby encrypter.rb
Dave Thomas
Dave Thomas
```

In the database, however, the name and email address are obscured by our industrial-strength encryption:

```
depot> sqlite3 -line db/development.sqlite3 "select * from orders"
     id = 1
user_id =
   name = Dbwf Tipnbt
address = 123 The Street
  email = ebwf@fybnqmf.dpn
```

Callbacks are a fine technique, but they can sometimes result in a model class taking on responsibilities that aren't really related to the nature of the model. For example, in Participating in the Monitoring Process, on page 327, we created a callback that generated a log message when an order was created. That functionality isn't really part of the basic Order class—we put it there because that's where the callback executed.

When used in moderation, such an approach doesn't lead to significant problems. If, however, you find yourself repeating code, consider using concerns[4] instead.

## Transactions

A database transaction groups a series of changes in such a way that either the database applies all of the changes or it applies none of the changes. The classic example of the need for transactions (and one used in Active Record's own documentation) is transferring money between two bank accounts. The basic logic is straightforward:

```
account1.deposit(100)
account2.withdraw(100)
```

But we have to be careful. What happens if the deposit succeeds but for some reason the withdrawal fails (perhaps the customer is overdrawn)? We'll have added $100 to the balance in account1 without a corresponding deduction from account2. In effect, we'll have created $100 out of thin air.

Transactions to the rescue. A transaction is something like the Three Musketeers with their motto "All for one and one for all." Within the scope of a transaction, either every SQL statement succeeds or they all have no effect. Putting that another way, if any statement fails, the entire transaction has no effect on the database.

4. https://api.rubyonrails.org/classes/ActiveSupport/Concern.html

In Active Record we use the transaction() method to execute a block in the context of a particular database transaction. At the end of the block, the transaction is committed, updating the database, *unless* an exception is raised within the block, in which case the database rolls back all of the changes. Because transactions exist in the context of a database connection, we have to invoke them with an Active Record class as a receiver.

Thus, we could write this:

```
Account.transaction do
  account1.deposit(100)
  account2.withdraw(100)
end
```

Let's experiment with transactions. We'll start by creating a new database table. (Make sure your database supports transactions, or this code won't work for you.)

rails7/e1/ar/transactions.rb

```
create_table :accounts, force: true do |t|
  t.string  :number
  t.decimal :balance, precision: 10, scale: 2, default: 0
end
```

Next, we'll define a rudimentary bank account class. This class defines instance methods to deposit money to and withdraw money from the account. It also provides some basic validation—for this particular type of account, the balance can never be negative.

rails7/e1/ar/transactions.rb

```
class Account < ActiveRecord::Base
  validates :balance, numericality: {greater_than_or_equal_to: 0}
  def withdraw(amount)
    adjust_balance_and_save!(-amount)
  end
  def deposit(amount)
    adjust_balance_and_save!(amount)
  end
  private
  def adjust_balance_and_save!(amount)
    self.balance += amount
    save!
  end
end
```

Let's look at the helper method, adjust_balance_and_save!(). The first line simply updates the balance field. The method then calls save! to save the model data. (Remember that save!() raises an exception if the object cannot be saved—we use the exception to signal to the transaction that something has gone wrong.)

So now let's write the code to transfer money between two accounts. It's pretty straightforward:

rails7/e1/ar/transactions.rb
```
peter = Account.create(balance: 100, number: "12345")
paul  = Account.create(balance: 200, number: "54321")

Account.transaction do
  paul.deposit(10)
  peter.withdraw(10)
end
```

We check the database, and, sure enough, the money got transferred:

```
depot> sqlite3 -line db/development.sqlite3 "select * from accounts"
     id = 1
 number = 12345
balance = 90

     id = 2
 number = 54321
balance = 210
```

Now let's get radical. If we start again but this time try to transfer $350, we'll run Peter into the red, which isn't allowed by the validation rule. Let's try it:

rails7/e1/ar/transactions.rb
```
peter = Account.create(balance: 100, number: "12345")
paul  = Account.create(balance: 200, number: "54321")
```

rails7/e1/ar/transactions.rb
```
Account.transaction do
  paul.deposit(350)
  peter.withdraw(350)
end
```

When we run this, we get an exception reported on the console:

```
.../validations.rb:736:in `save!': Validation failed: Balance is negative
from transactions.rb:46:in `adjust_balance_and_save!'
  :        :        :
from transactions.rb:80
```

Looking in the database, we can see that the data remains unchanged:

```
depot> sqlite3 -line db/development.sqlite3 "select * from accounts"
     id = 1
 number = 12345
balance = 100
```

```
     id = 2
 number = 54321
balance = 200
```

However, there's a trap waiting for you here. The transaction protected the database from becoming inconsistent, but what about our model objects? To see what happened to them, we have to arrange to intercept the exception to allow the program to continue running:

rails7/e1/ar/transactions.rb

```ruby
peter = Account.create(balance: 100, number: "12345")
paul  = Account.create(balance: 200, number: "54321")
```

rails7/e1/ar/transactions.rb

```ruby
begin
  Account.transaction do
    paul.deposit(350)
    peter.withdraw(350)
  end
rescue
  puts "Transfer aborted"
end

puts "Paul has #{paul.balance}"
puts "Peter has #{peter.balance}"
```

What we see is a little surprising:

```
Transfer aborted
Paul has 550.0
Peter has -250.0
```

Although the database was left unscathed, our model objects were updated anyway. This is because Active Record wasn't keeping track of the before and after states of the various objects—in fact, it couldn't, because it had no easy way of knowing just which models were involved in the transactions.

## Built-In Transactions

When we discussed parent and child tables in Specifying Relationships in Models, on page 310, we said that Active Record takes care of saving all the dependent child rows when you save a parent row. This takes multiple SQL statement executions (one for the parent and one each for any changed or new children).

Clearly, this change should be atomic, but until now we haven't been using transactions when saving these interrelated objects. Have we been negligent?

Fortunately, no. Active Record is smart enough to wrap all the updates and inserts related to a particular save() (and also the deletes related to a destroy())

in a transaction; either they all succeed or no data is written permanently to the database. You need explicit transactions only when you manage multiple SQL statements yourself.

While we've covered the basics, transactions are actually very subtle. They exhibit the so-called ACID properties: they're Atomic, they ensure Consistency, they work in Isolation, and their effects are Durable (they're made permanent when the transaction is committed). It's worth finding a good database book and reading up on transactions if you plan to take a database application live.

## What We Just Did

We learned the relevant data structures and naming conventions for tables, classes, columns, attributes, IDs, and relationships. We saw how to create, read, update, and delete this data. Finally, we now understand how transactions and callbacks can be used to prevent inconsistent changes.

This, coupled with validation as described in Chapter 7, Task B: Validation and Unit Testing, on page 85, covers all the essentials of Active Record that every Rails programmer needs to know. If you have specific needs beyond what is covered here, look to the Rails Guides[5] for more information.

The next major subsystem to cover is Action Pack, which covers both the view and controller portions of Rails.

5. http://guides.rubyonrails.org/