Database Programming with Perl

```
#!/usr/bin/perl
use strict;

                       print "\nSaid the Llama to the Lama:\n";

            my (@Llama, %answers, @llama, @questions) = map{ print;$_ }(<>);

            die "seeking a llama to share pipe!\n" unless $Llama[0]=~/Find/;

                       print "\nSaid the Lama to the Llama:\n";


                                    my$karma=0
                                   ;for ( @Llama
                                 ){chomp; ; chomp;
                                s/Fi/Fou/;s/-/--/;
                                ;/  /;  my ( $lama,
                                $llama ) = ($‘,$’);
                                s/ //g for( $lama ,
                                 $llama); $|=0;push
                                 @questions,$llama
                                  unless $llama=~
                                    /^$/ ; push
                                     @llama,#
                                $lama unless $lama=~
                             /^$/} %answers = map { $_
                         , 0,  scalar reverse(   split //
                        ) ,0 }@questions;my   @karma=llama(
                      @llama);my  @dharma =  llama(map {  (
                    ’ ’) x $_ . $llama[$_]. (’ ’) x (1*  $#llama
                   -$_)}(0..$#llama));  my @lama=map{$a = $_;;;;;
                  s/( *)(\w+)( *)/$3$2$1/;  $a}@dharma; my  %find =
                 (lama => \@llama,dharma =>\@karma,karma  =>\@dharma
                ,llama=> \@lama); sub   llama{map{$|++; my  $grazing=
              $_  ;join(  ’’, map{ (  split //,$_[  $_ ])  [$grazing]
             } (0..$#_   ))}(0.. (  length($_[0])-1))}sub    dharma{my
           @llama  =     @{$_[0]}  ;[map{s/[a-z]/\*/g;; ;    $_}@llama
           ] };; my      %llama=  map{$_,dharma($find{$_})     } ( keys
          %find  );;      sub  karma { my $reasons=shift      ;; ; ; my
           @answers       =  @{ $find{  "$reasons" } };;      for  my
          $questions     (keys%answers){next if $answers{   $questions
            }; study     $questions;for (0..  $#answers   ) {  if (
             $answers[    $_]=~/$questions/){my($soul   ,$seeks ,
     $to_know   )=($‘,$’  ,$&)    ;s/\w/\*/g for     ($seeks,     $to_know,
   $soul);$llama{  $reasons}    ->[$_]=$soul  .    $questions   .$seeks;$karma
  ++;$answers{    $questions}  ++  ;   $answers{   reverse(  split//,$to_know)}++
  }}}};for(keys    %find){   karma($_  )};my@dali  =map{$a=  $_;$a=~s/./\*/g;$a}
   (@llama); for         my $karma(keys   %llama){  $llama    {$karma}=[ llama(
     @{$llama{$karma}})] if $karma=~/.{5}/   ;for(0  ..    $#{$llama{$karma}}){
      $llama{$karma}->[$_]  =~s/ //g;my@Lama  =split//, $dali[$_];my @lAma=
        split//,$llama{$karma}->[$_];$dali[$_]=    join(’’,map{($lAma[$_]
    ne ’*’) ?   $lAma[$_]:$Lama[$_]}(0..$#lAma));}}my @answer=map { s/(.)/$1 /g
  ;s/\*/ /g;  ($_,’ ’x(length($_)))}(@dali);my($llama,$lama)=sort($#questions,
 $#answer) ;    for   (0..$lama) {print $answer[$_],"  ",$questions[$_],"\n" }
```

Jacinta Richardson
Paul Fenwick

Database Programming with Perl

by Jacinta Richardson and Paul Fenwick

Conventions used throughout this text are based upon the conventions used in the Netizen training manuals by Kirrily Robert, and found at http://sourceforge.net/projects/spork

This training manual is maintained by Perl Training Australia. Information about Perl Training Australia’s courses, manuals and course extracts can be found at http://www.perltraining.com.au/.

This is revision 1.8 of the Perl Training Australia’s "Database Programming with Perl" training manual.

Table of Contents

List of Tables

Chapter 1. About Perl Training Australia

Training

Perl Training Australia (http://www.perltraining.com.au) offers quality training in all aspects of the Perl programming language. We operate throughout Australia and the Asia-Pacific region. Our trainers are active Perl developers who take a personal interest in Perl's growth and improvement. Our trainers can regularly be found frequenting online communities such as Perl Monks (http://www.perlmonks.org/) and answering questions and providing feedback for Perl users of all experience levels.

Our primary trainer, Paul Fenwick, is a leading Perl expert in Australia and believes in making Perl a fun language to learn and use. Paul Fenwick has been working with Perl for over 10 years, and is an active developer who has written articles for *The Perl Journal* and other publications.

Doctor Damian Conway, who provides many of our advanced courses, is one of the three core Perl 6 language designers, and is one of the leading Perl experts worldwide. Damian was the winner of the 1998, 1999, and 2000 Larry Wall Awards for Best Practical Utility. He is a member of the technical committee for OSCON, a columnist for The Perl Journal, and author of the book "Object Oriented Perl".

Consulting

In addition to our training courses, Perl Training Australia also offers a variety of consulting services. We cover all stages of the software development life cycle, from requirements analysis to testing and maintenance.

Our expert consultants are both flexible and reliable, and are available to help meet your needs, however large or small. Our expertise ranges beyond that of just Perl, and includes Unix system administration, security auditing, database design, and of course software development.

Contact us

If you have any project development needs or wish to learn to use Perl to take advantage of its quick development time, fast performance and amazing versatility; don't hesitate to contact us.

Table 1-1. Perl Training Australia's contact details

Phone:	03 9354 6001
Fax:	03 9354 2681
Email:	contact@perltraining.com.au
Webpage:	http://www.perltraining.com.au/
Address:	104 Elizabeth Street, Coburg VIC, 3058

Chapter 2. Introduction

Welcome to Perl Training Australia's *Database Programming with Perl* training module. This is a two day training module in which you will learn how to interact with databases through Perl.

Course outline

- Accessing the database
- Programming with DBI
- SQL::Abstract
- Transactions
- Database security
- Class::DBI
- Removing passwords from your programs
- Working with remote connections

Assumed knowledge

This training module assumes the following prior knowledge and skills:

- Intermediate Perl fluency, including a familiarity with Perl variable types and references, file input/output, system interaction and using Perl modules. Some experience of using Perl objects would be useful.
- Basic Unix fluency, including logging in, moving around directories, and editing files

Module objectives

- Understand what a database is and why we use them
- Understand basic SQL commands
- Be able to use the database shell to access the database
- Understand why Perl's `DBI` exists
- Use `DBI` to add, update, delete and select data in the database.
- Understand and use content from extra material.

Platform and version details

Perl is a cross-platform computer language which runs successfully on approximately 30 different operating systems. However, as each operating system is different this does occasionally impact on the code you write. Most of what you will learn will work equally well on all operating systems; your instructor will inform you throughout the course of any areas which differ.

At the time of writing, the most recent stable release of Perl is version 5.8.8, however older versions of Perl 5 are still common. Your instructor will inform you of any features which may not exist in older versions.

The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographical conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

```
Program listings and other literal listings of what appears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appear *`like this`*

☞ Notes and tips appear offset from the text like this.

❗ Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.

📖 Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.

⚠ Notes marked "Caution" contain details of unexpected behaviour or traps for the unwary.

Other materials

In addition to these notes, it is highly recommend that you obtain a copy of Programming the Perl DBI by Alligator Descartes and Tim Bunce. This book can be found at most major bookstores. Alternately, if you're planning on attending a Perl Training Australia course in the future, ask about our discount rates for this and other suggested text books.

The Programming the Perl DBI book is often referred to as the Cheetah book, due to the cheetah on the front cover. This is how it will be referred to in these notes.

These notes have been developed to be useful in their own right, however this book covers an extensive range of topics not covered in this course and discusses many of the concepts covered in these notes in more detail. Tim Bunce is the current maintainer of most of the database modules mentioned herein.

Chapter 3. Our database

Database schema

The database which we will be working with today has the table structure and contents as described below.

```
CREATE TABLE staff(
        StaffID         INT             AUTO_INCREMENT PRIMARY KEY,
        FirstName       VARCHAR(15)     NOT NULL,
        LastName        VARCHAR(15)     NOT NULL,
        Address         VARCHAR(30)             ,
        City            VARCHAR(15)             ,
        State           VARCHAR(3)              ,
        Position        VARCHAR(20)     NOT NULL,
        Wage            INT             NOT NULL
);

CREATE TABLE projects (
        StaffID         INT             NOT NULL,
        ProjectName     VARCHAR(20)     NOT NULL,
        Allocation      INT             NOT NULL,
        PRIMARY KEY (StaffID, ProjectName)      ,
        FOREIGN KEY (StaffID) REFERENCES Staff(StaffID) ON DELETE CASCADE
);

CREATE TABLE staffphone (
        StaffID         INT             NOT NULL,
        PhoneNumber     VARCHAR(15)     NOT NULL,
        PRIMARY KEY (StaffID, PhoneNumber)      ,
        FOREIGN KEY (StaffID) REFERENCES Staff(StaffID) ON DELETE CASCADE
);
```

```
+--------------------------------------------------------------------------------------+
|                       staff                                                          |
+---------+-----------+----------+---------------+-----------+-------+----------+------+
| StaffID | FirstName | LastName | Address       | City      | State | Position | Wage |
+---------+-----------+----------+---------------+-----------+-------+----------+------+
|   12345 | Jack      | Sprat    | 2 Pine Rd     | Melbourne | Vic   | Devel    |   80 |
|   12346 | John      | Doe      | 43 James Crt  | Melbourne | Vic   | Devel    |  160 |
|   12347 | Betty     | Jones    | 22 Fishing St | Melton    | Vic   | Manager  |  180 |
|   12349 | Sam       | Smith    | 55 Queens Ave | Sydney    | NSW   | Admin    |  130 |
|   12350 | Ann       | Smith    | 10 Albert St  | Brisbane  | QLD   | Devel    |   75 |
|   12351 | Peter     | Pope     | 254 King St   | Sydney    | NSW   | Manager  |  180 |
+---------+-----------+----------+---------------+-----------+-------+----------+------+
```

```
----------------------------------------+
|            projects                   |
+---------+---------------+-------------+
| StaffID | ProjectName   | Allocation  |
+---------+---------------+-------------+
|   12345 | ABC           |          50 |
|   12346 | ABC           |          45 |
|   12347 | ABC           |         100 |
|   12351 | ABC           |          70 |
|   12346 | NMO           |          60 |
|   12345 | XYZ           |          50 |
|   12349 | XYZ           |          50 |
|   12350 | XYZ           |         100 |
|   12351 | XYZ           |          30 |
+---------+---------------+-------------+
+------------------------+
|       staffphone       |
+---------+--------------+
| StaffID | PhoneNumber  |
+---------+--------------+
|   12345 | 0401 111 111 |
|   12345 | 03 9333 3333 |
|   12346 | 03 8444 4444 |
|   12347 | 0423 222 222 |
|   12347 | 03 5555 5555 |
|   12349 | 02 9333 3334 |
|   12349 | 02 8333 3222 |
|   12351 | 02 9442 2233 |
+---------+--------------+
```

Chapter 4. Accessing the database

In this chapter...

In this chapter we'll discuss how to access a database from Perl. We will cover using the database's shell, database specific modules and Perl's `DBI`.

Our training database

In this course we're going to be using the `MySQL` database. This database is released under the GNU Public License and can be down-loaded from the MySQL (http://www.mysql.com/) website. It is a multi-user, multi-threaded SQL database with a client-server implementation consisting of a server daemon (`mysqld`) and many different client programs. It's free, cross-platform, easy to use, and is a commonly seen choice for commercial applications.

Like many other databases `MySQL` is not fully ANSI 92 compliant with its SQL and some things that work in other databases will not work with MySQL. Likewise MySQL provides some functionality that won't work in your database. Please refer to your database documentation when these problems arise.

The techniques presented in this course are *not* specific to MySQL, and can be applied to any database for which a driver for Perl's `DBI` module exists. This includes most databases in common use today.

The MySQL shell

MySQL, like many other databases, provides a shell interface for direct access to the database. The MySQL shell can be invoked from the command line as follows:

```
% mysql -u username -ppassword databasename
% mysql -u username -p databasename
```

The second format is preferred where possible as it prompts the user for a password rather than showing this information to anyone who can glance at your screen, see your process information, or look through your shell's history files.

The MySQL shell prompt appears as follows:

```
mysql>
```

Add your SQL or MySQL command at this prompt. SQL and commands can span more than one line and can include any amount of whitespace. Hence the following is a valid statement:

```
mysql> SELECT StaffID
    -> FROM projects
    -> WHERE Allocation = 100;
```

Notice that MySQL changes its prompt to indicate that you are within a statement. MySQL also changes its prompt if you have opened a set of quotes which you have not closed:

```
mysql> SELECT *
    -> FROM staffphone
    -> WHERE PhoneNumber = '
    '>
```

Newlines embedded between quotes will be treated literally in comparisons.

If you're writing a complex query in MySQL, it can help to use an editor rather than writing the query on a line-by-line basis. You can invoke your editor (`vim` by default on our system), by typing `\e` at the MySQL prompt.

☞ In our discussion of SQL in the introduction to SQL you may notice that every query ends with a semicolon (`;`). This tells MySQL that we're finished with this query. Don't forget to add this when you're dealing with MySQL's shell.

MySQL commands.

Using the database shell is different for each database. The instructions we provide here work for MySQL specifically. For information on using your database's shell read your documentation.

Table 4-1. MySQL commands

Command	Action
show tables;	Lists the tables available in your database.
describe *tablename*;	Provides information about table contents
help;	Provides a list of available commands
quit;	Exit the MySQL shell
LAST_INSERT_ID()	SQL function to return the row id of the last inserted entry.

Exercises

Your MySQL username and password have been provided to you at the start of this class along with the name of your database. Start up the MySQL shell on the command line for the following exercises.

Use the **show tables** and **describe** commands to get an idea of what's available in the database.. You should be able to fetch all entries in a table with a simple `SELECT` statement.

1. `SELECT` all staff who live in Melbourne.
2. `Sam Smith` has changed to full-time work, from part-time. `UPDATE` her allocation to 'XYZ' to `100%`.
3. Tearfully, `John Doe` is leaving the company for greener pastures and higher wages. `DELETE` John from the appropriate tables.

4. Charles Apple has just joined the business. Charles lives at `98 Lizzy Hwy` in `Brisbane`. He's signed on as an 'Admin' and expects to earn 80 units. He's been assigned to the 'NMO' project at 100% capacity. `INSERT` Charles to the appropriate tables.
5. `SELECT` the names of the staff who are committed to two or more projects (harder).

Answers for these questions can be found in `exercises/answers/sql_exercises.sql`

Accessing the database from Perl

There are three main ways to access a database through Perl. These are:

1. Using `system` or `open` to access the database's shell directly.
2. Using a database specific module.
3. Using `DBI`, Perl's *D*ata*B*ase *I*nterface module or one of the modules that builds upon it.

Using the database shell directly

This can be done with code similar to the following:

```
# A VERY BAD IDEA

# Select everything from the Staff table
open (SELECT, "mysql -u $username -p$password $database -e
               'SELECT * from staff' |"
        or die "Failed for some reason: $!";

my @results = <SELECT>;
close SELECT;

# Now we need to parse each result into a form that's useful to us,
# which is quite a task by itself.  Here we just print them out.
print "@results\n";

# Insert something into the database
system(qq{
        mysql -u $username -p$password $database -e
                "INSERT INTO staffphone (StaffID, PhoneNumber)
                VALUES (12345, '02 2222 2222')"
        }
);
die "Error: $?" if $?;
```

Although it is possible to use the database shell through Perl this is a painful and tortuous procedure. It increases the complexity of your code and therefore the probability of errors for no extra gain. We cannot recommend this approach when there are more simple and elegant options available.

Using a database specific module

Over time database specific modules in Perl have slowly disappeared in favour of Perl's `DBI`. Overall this has increased flexibility in using and migrating between databases when using Perl. However in many cases the drivers for `DBI` require installation of database client libraries. Occasionally these may not exist for your operating system. In this situation, using a pure Perl database specific module which does work on your system may be the best solution.

One pure Perl MySQL module is `Net::MySQL`. This can be down-loaded from CPAN (http://search.cpan.org/perldoc?Net::MySQL), the Comprehensive Perl Archive Network. Documentation is linked to from that page. The synopsis for using `Net::MySQL` is as follows:

```
use Net::MySQL;                 # Load in module

# Create a handle to our mysql instance
my $mysql = Net::MySQL->new(
      hostname => ’localhost’,
      database => ’dbiX’,
      user     => ’dbiX’,
      password => ’secret’
);
###############
# INSERT example
###############
$mysql->query(q{
         INSERT INTO staffphone (StaffID, PhoneNumber)
                VALUES (12349, ’02 32323232’)
});
printf "Affected rows: %d\n", $mysql->get_affected_rows_length;

##############
# SELECT example
##############
$mysql->query(q{SELECT StaffID, PhoneNumber FROM staffphone});

# Create a handle to our results
my $record_set = $mysql->create_record_iterator;

# Iterate over each result printing out what we got.
while (my $record = $record_set->each) {
        printf "StaffID: %s PhoneNumber: %s\n",
        $record->[0], $record->[1];
}
$mysql->close;
```

SQL injection attacks

The `Net::MySQL` module requires the programmer to ensure that the information passed through to MySQL is *clean*. This means that if any parts of your SQL statements are coming from the user you need to make sure that all single quotation marks are properly escaped as required by the database. For example consider the following code:

```
print "Last name: ";
my $staffname = <STDIN>;
chomp $staffname;

$mysql->query(qq{SELECT * FROM staff where LastName = ’$staffname’});
```

This code is wide open to an *SQL injection attack* because it allows the user to input a value that dramatically changes the effect of the executed SQL. As an example:

```
Fred’; drop table staff; commit; ’
```

Then the query becomes (with some added whitespace):

```
SELECT * FROM staff where LastName = ’Fred’;
drop table staff;
commit;
’’
```

The `DROP TABLE` command deletes the database table and all of its data from the database.

Fortunately the MySQL database is hard to fool in this way, as it expects to receive only a single query in each call, causing the attack to fail. However an imaginative attacker can often use more advanced features, such as unions and sub-selects, to extract information from the database or manipulate the output returned to the program.

Not all databases have the restriction of a single query per call, and even the most unimaginative attacker can take advantage of this. While it is possible to write a regular expression to escape single quotes in strings, you have to remember to properly escape the escaping character too. When faced with a difficult problem like this, it's better to see who's solved it in the past, or use a better tool such as `DBI`.

We'll discuss how `DBI` helps us avoid SQL injection attacks in the next chapter. We will also discuss SQL injection attacks and other database security issues later in this course.

Group Exercises

1. Users may enter data containing quote marks which may change the function of your program. Often this is not a malicious attempt, but instead can be caused by legitimate data that contains SQL meta-characters. List cases where a user may provide valid data which will upset the database.
2. How can you escape quotes and other meta-characters in user input before passing it to the database?

Using DBI and derivatives

Interfacing directly with your database's shell or using a database specific module removes your options to make your code portable across databases. It also means that you need to learn a new module for every database engine that you use, and the advantages and disadvantages associated with that module.

Perl has a module called `DBI`, which is short for *DataBase Interface*, that helps you get around all of these issues and its use is considered *industry best practice*. `DBI` is a mature technology with a well planned interface. It is fast, efficient and an essential tool in the database programmers toolkit. `DBI` will be covered further in the rest of this course.

There are many `DBI` extensions which can be found on CPAN. Some of these extensions are stored in the `DBIx` name space and include `DBIx::Recordset` which attempts to simplify database access even further, `DBIx::Abstract` which provides methods for avoiding writing SQL and `DBIx::AnyDBD` which attempts to further increase portability by providing a database driver which can be customised to work on multiple platforms. These each have their advantages and disadvantages but will not be covered further here. For more information on these modules search for them on CPAN (http://search.cpan.org/search?query=DBIx%3A%3A).

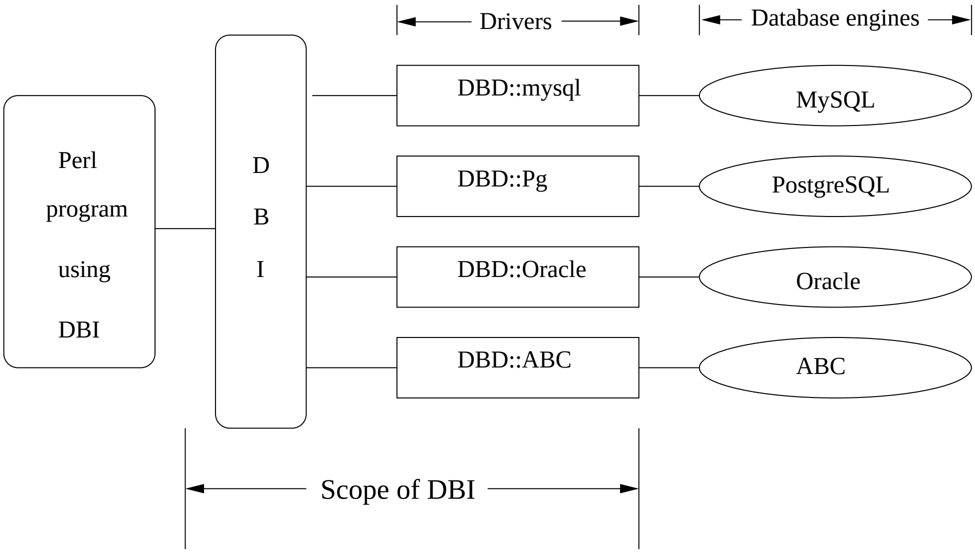
Introduction to DBI

What is DBI?

The `DBI` is Perl’s database access module. It defines the methods, variables and conventions required to provide a consistent database interface to the programmer, regardless of the actual database or databases being used.

`DBI` provides this database independence by dynamically loading appropriate database drivers to do the actual work in the background. A large number driver modules are available for Perl, including drivers for all the major database servers and several drivers for other database-independent interfaces such as `ODBC`. `DBI` also provides error checking and handling and specifies default methods for non-database specific tasks.

Figure 4-1. Architecture of a DBI Application



Each database driver (`DBD`) implements the `DBI` methods using the private interface function of its corresponding database engine. As `DBI` guarantees a consistent and uniform interface, normally the only special setup required is to ensure the appropriate `DBD` module is installed for your database.

Database-specific features are documented in the appropriate `DBD` documentation, and you should consult this if a feature you wish to use or set is not part of the standard `DBI` interface. An example of this would be setting the `LongReadLen` for oracle `long` fields.

Supported databases

While Perl comes bundled with a great many useful modules in the standard distribution, `DBI` is not one of them. As `DBI` requires you to (usually) have a separate database engine to be useful, it’s not considered part of the core distribution. Likewise `DBI` does not come with any database drivers by default. To write Perl programs that interface with databases you have to install both `DBI` and the appropriate `DBD` for your database. The best procedure to perform this installation depends upon your operating system, and includes using the `Perl Package Manager` under *ActiveState Perl* for Windows, `apt-get` under Debian/Linux, and the CPAN shell for many Unix flavours.

As `DBI` is one of the most heavily used modules in Perl, it's either standard or extremely easy to install with many operating system distributions. A full list of drivers can also be found found on CPAN (http://search.cpan.org/modlist/Database_Interfaces/DBD).

Why use DBI?

`DBI` isn't a miracle cure for difficult database applications. There are times when using `DBI` may not be appropriate, such as when you have a large application that already makes use of another method.

By providing a common interface to all databases, `DBI` makes it harder to use some database specific options. For example scrolling cursors. `DBI` also does not provide non-sequential access to data rows pulled out of the database. The only way to achieve that is to pull all the data out into your own structure and then access that non-sequentially.

Having discussed some issues with using `DBI` we feel that `DBI` is one of the Perl community's finest assets. `DBI` makes it extremely easy to change the underlying database in your program by providing that database independent interface. `DBI` protects your database from *SQL injection attacks* and increases database-to-program communication efficiency by using placeholders. It's easy to learn, very powerful and heavily optimised for speed.

Perl's `DBI` module is actively supported and has been the foundation of similar specifications in other programming languages.

Chapter summary

- Using the database's shell through Perl involves using `open` and `system`. This is a less than ideal process.
- Database specific modules can be used when an appropriate `DBI` database driver does not exist for your operating system.
- Some database specific modules are limited in their capacity to handle database meta-characters and proper escaping of this is left to the programmer.
- Unescaped database meta-characters can result in failed database queries. They can also result in allowing users to gain more information about your system than appropriate or run database commands other than what you intended.
- `DBI` is Perl's DataBase Interface. It allows Perl programs to interface with databases regardless of which database is being used.

Chapter 5. Programming with the DBI

In this chapter...

In this chapter we will cover how to interact with a database in Perl by using the `DBI` module. This will include connecting and disconnecting, selecting data and using it, inserting, updating and deleting entries. `DBI`'s ability for using placeholders and bind values will also be covered in depth.

`DBI` provides a great deal more functionality than we'll cover in this course. For more information about `DBI` read the documentation in **perldoc DBI**. If you have further questions the `DBI::FAQ` is a good place to start. If this is installed on your system you can read it at **perldoc DBI::FAQ** otherwise try its CPAN page (http://search.cpan.org/perldoc?DBI::FAQ).

Establishing a database connection

When we use `DBI` to connect to a database, we are provided with a *database connection object*. The object is commonly referred to as the *database handle*, and in code is often given the name of `$dbh`. The advantage of being given an object to represent the database connection is that it's easy to connect to multiple databases at once and to see which parts of code can access the database. This makes our code easier to audit and debug.

The Cheetah book describes database handles on page 80.

As connecting to a database usually involves a lot of overhead (establishing the connection, authentication, selecting the database you wish to use, and so on), most programs establish a connection to the database when they start, and disconnect just before they end.

If you've never used Perl's object oriented features before, then you'll need to know the following syntax:

```
$object->method(@arguments);
```

The arrow notation means that we are requesting the object (always on the left of the arrow), to perform some given action or method (on the right of the arrow). Passing arguments to methods is done in the same way as passing arguments to subroutines or in-built functions.

As an example, when we see the code:

```
$dbh->disconnect;
```

we're asking the `$dbh` object to disconnect. In the following example, we're asking the object to `do` a certain SQL statement.

```
$dbh->do("DELETE FROM user WHERE name='fred'");
```

Perl Training Australia offers a 2-day course that covers Perl's object oriented features in depth (http://perltraining.com.au/courses/perloo.html).

To connect to a database using `DBI` we call the `connect` method.

```
use DBI;

my $dbh = DBI->connect("dbi:$driver:$dsn", $username, $password,
                        { AutoCommit => 1 });
```

The first argument of `connect` is often called the *DSN* (Data Source Name) although, technically, the `DSN` is only part of this string. This string consists of three main parts;

1. the string `dbi:` (or `DBI:`, case is not important)
2. the name of the driver, for example `mysql` for MySQL or `Pg` for PostgreSQL.
3. the data source name (`DSN`), this is usually just the database name. For example: `database=dbiX`. The `DSN` can also include further information for connecting to the database remotely such as the hostname, database port etc. For example: `database=dbiX;host=example.perltraining.com.au`.

To find out more information about the `connect` method's parameters read page 85 in the Cheetah book.

To find out what information you may need to provide in the `DSN` read the perldoc for your driver. For example read **perldoc DBD::mysql** for information on the MySQL driver and **DBD::Pg** for information on the PostgreSQL driver.

The second and third arguments to `connect` allow you to specify your username and password. If you have no password setup for access to your database then provide the empty string instead. The final argument is a hash reference of connection flags. We'll cover these in more detail later in this course.

Putting it all together, we can connect to a MySQL database with the following program. This program doesn't yet do anything with the database connection after it is established. We will also discuss `AutoCommit` later in this text, but we always recommend you set it explicitly.

```
#!/usr/bin/perl -w
use strict;
use DBI;

my $driver   = "mysql";
my $dsn      = "database=dbiX";
my $username = "dbiX";
my $passwd   = "";

my $dbh = DBI->connect("dbi:$driver:$dsn", $username, $passwd, { AutoCommit => 1 });
```

Testing that the connection worked

There are many reasons for a database connection to fail. Perhaps the driver couldn't be found, or the database name is misspelled. Perhaps the username and password are incorrect. Maybe the database is down.

Good programming practice tells us to check that the database connection worked rather than assuming it did. Like many other Perl functions, `connect` returns a false value if the connection fails.

The reason for the connection failure is stored in `$DBI::errstr`. This is just a regular scalar, but the `DBI::` part of its name tells Perl to look inside the `DBI` package for it.

We can test `$dbh` for success (a true value) after the connect statement or call die upon failure:

```
my $dbh = DBI->connect("dbi:mysql:database=dbiX", $username, $passwd,
                  { AutoCommit => 1 })
        or die "Failed to connect to database: $DBI::errstr";
```

Exercise

We'll build on this script throughout the rest of this chapter.

1. Write a program which connects to your database using `DBI`. Verify that your connection was successful.

Testing the connection later

Some database drivers provide a `ping` method which you can use to see if the database handle is still connected to a database. This is rarely used in most database programs. If implemented, `ping` returns true if the database is up and false otherwise.

If you call `ping` on a database handle when your driver doesn't implement it, will receive a string which is true in a boolean context and false in a numerical context (such as "0E0"). It is best to ensure that your driver provides the `ping` method before you rely upon it.

```
$dbh->ping or die "Database has fallen over - $DBI::errstr.";
```

Simple connection flags

`DBI` allows us to specify a number of connection flags in a hash reference to the connect method. These let `DBI` and the database drivers know that we want certain extra things to happen. For example, the flag we've passed above:

```
        { AutoCommit => 1 }
```

tells `DBI` that we're not intending to use transactions. This means that every time we execute an SQL statement (`INSERT`, `UPDATE`, `DELETE`, `SELECT`) we want the database to save that result immediately rather than allowing us to undo it later.

☞ The `DBI` documentations highly recommends explicitly setting your desired `AutoCommit` behaviour, as in older versions of DBI the default `AutoCommit` behavior is driver-dependent.

`DBI` has many options you can use to affect how the database and statement handles work in certain conditions. In this course we cover the following:

Table 5-1. DBI connection flags

Flag	Meaning
AutoCommit Default: on Do not rely on this default, always set this value yourself.	If this is true then database changes cannot be rolled back (undone). If this is false then database changes do not get saved to the database until an explicit `commit` is sent to the database.
PrintError Default: on	Print warnings to `STDERR` on error. These are the raw error message from the database, and may not include the SQL that generated the error.
ShowErrorStatement Default: off	This attribute causes the SQL statement text to be included in the database error generated by `RaiseError`, `PrintError` and `PrintWarn`. This can greatly assist in debugging.
RaiseError Default: off	This attribute can be used to make database errors generate exceptions when they occur. If you don't catch these exceptions then your program will die.
TaintIn Default: off	When this attribute is set to true and Perl is running in taint mode then all arguments to `DBI` methods are checked for being tainted. If they are tainted then your program will die with a fatal error. Setting TaintIn to true has no effect when Perl is not running in taint mode.
TaintOut Default: off	When this attribute is set to true and Perl is running in taint mode then all data from the database is considered tainted. If you use this data without cleaning in an unsafe way then your program will die with a fatal error. Setting `TaintOut` to true has no effect when Perl is not running in taint mode.
Taint Default: off	Setting this attribute to true has the same effect as setting both `TaintIn` and `TaintOut` to true.

DELETE, INSERT, UPDATE

The `DELETE`, `INSERT` and `UPDATE` all have one important trait in common. They modify the database and return the number of rows changed. In contrast, the `SELECT` command pulls data from the database without modification.

Because the return values of these non-`SELECT` commands is so simple, `DBI` provides a simple way to just do them. This method is called `do`. This method is best used for single, unique statements rather than multiple statements of the same general type. For example to update an entry in your table once in your script use `do`.

```
#!/usr/bin/perl -w
use strict;
use DBI;

my $driver   = "mysql";
my $dsn      = "database=dbiX";
my $username = "dbiX";
my $passwd   = "secret";

my $dbh = DBI->connect("dbi:$driver:$dsn", $username, $passwd,
                { AutoCommit => 1 })
        or die "Failed to connect to dbi: $DBI::errstr";

$dbh->do("INSERT INTO staffphone (StaffID, PhoneNumber)
          VALUES (12351, ’02 9999 9999’)")
        or die "Failed to insert row: " . $dbh->errstr;

$dbh->do("DELETE FROM staffphone WHERE StaffID = 12349 AND
          PhoneNumber = ’02 9333 3334’")
        or die "Failed to delete row: " . $dbh->errstr;

$dbh->do("UPDATE staff SET Wage = 79 WHERE StaffID = 12346")
        or die "Failed to update row: " . $dbh->errstr;
```

The `DBI` `do` method returns the rows affected by the last operation, if available. If the number of rows cannot be determined (not known, not available or not applicable) it returns `-1`. `undef` is returned on error.

Exercises

Write a Perl program to make the following changes to your database. Use the MySQL shell to confirm your changes.

1. Insert a new employee into the database. Don’t worry about adding their details to the `projects` and `staffphone` tables for now.
2. Update an employee’s address.
3. Delete a phone number from the `staffphone` table.
4. What happens when you run this program a second time?

An answer for the above is provided in `exercises/answers/do.pl`.

Inserting, updating, deleting based on external input

A program which always makes the same inserts, updates and deletes isn’t very useful. We need a way to take user input and provide it to the database. Like all things Perl, there are many ways to do this, however some methods are better than others.

```
my $staffID     = <STDIN>;
my $phoneNumber = <STDIN>;
chomp($staffID, $phoneNumber);

# DON’T DO THIS
$dbh->do("UPDATE staff set LastName = ’$lastname’
          WHERE StaffID = $staffID ") or die $dbh->errstr;
```

Code like the above is unfortunately common, but it is also fragile and error prone. If a staff member wishes to change their last name to `O'Hara` or some other surname which includes an apostrophe, the above code will break. Breaking on valid data is a bad thing to do.

Furthermore, this code is open to SQL injection attacks as discussed in the previous chapter.

`DBI` allows us to place this data into our statement a neater and safer way.

```
my $staffID  = <STDIN>;
my $lastname = <STDIN>;
chomp($staffID, $lastname);

$dbh->do("UPDATE staff set LastName = ?
          WHERE StaffID = ?", undef, $lastname, $staffID ) or die $dbh->errstr;
```

This is the multi-argument version of the `do` method. It's general form is:

```
$dbh->do($sql, \%attrs, @bind_variables) or die $dbh->errstr;
```

The second argument, `\%attrs`, allows you to pass information to the driver implementing the method. This means you can pass in driver specific hints to improve the execution of the query. In most cases this isn't required and so we pass `undef` instead.

The subsequent values are called the *bind values* and are discussed in the next section.

The Cheetah book covers the `do` method on page 120.

Placeholders and bind values

The SQL in the previous example contained question marks instead of data. These question marks are called *placeholders*. Using placeholders allows us to signal to Perl where to put data that is then provided separately. Doing this gives the database driver an opportunity to pre-process both the query and the data before *binding* the data to the SQL. This can increase the speed of the query's execution and also increases the security of our code (by preventing *SQL injection attacks)*.

The *bind values* form data that we match with the placeholders.

You should notice in the previous example that the question marks are not enclosed in quote marks, even though quote marks would be required to provide valid SQL. This is because the database driver does the quoting for us. We don't have to be concerned about which placeholders to put quotes around and which ones to leave without (numbers and timestamps don't need quotes).

There are a few restrictions on where you can use placeholders: placeholders can only take the place of data. You cannot use them in place of the field or table names. They are also restricted to representing single scalar values. This means than you cannot use a placeholder to provide the whole `IN` list within a `SELECT`. Instead you'll have to use a placeholder for each value in the list.

Prepare and execute

So far we've learned about the `do` function to work with the database. However, you'll often find yourself wanting to execute SQL queries which are fundamentally the same, but with differing data values.

What we often want to do can be described as follows:

```
create SQL statement
   execute statement with variables set 1
   execute statement with variables set 2
   ...
```

Likewise, when we wish to `SELECT` data, we often wish to select out the same columns with different requirements on the rows. This sequence can be described as follows:

```
create SQL statement
   execute statement with variables set 1
      fetch data, fetch data, fetch data, ...
   execute statement with variables set 2
      fetch data, fetch data, fetch data, ...
   ...
```

These sequences are identical if we ignore the data fetching. As a result `DBI` provides us with methods to create a statement once and then execute it as many times as necessary thereafter (with different data each time if desired).

prepare

`DBI`’s `prepare` method creates a statement handle (like a database handle but for SQL statements) for a specific piece of SQL. Some databases use this opportunity to analyse and optimise the SQL, in preparation for the statement being executed.

To prepare a SQL statement we do the following:

```
my $sth = $dbh->prepare(q{
        SELECT FirstName, LastName, Project, Allocation
        FROM staff s, projects p
        WHERE s.StaffID = p.StaffID AND
              ProjectName = ?
        ORDER BY ProjectName
}) or die "Failed in statement prepare: " . $dbh->errstr;
```

When preparing a statement we can use placeholders without specifying the values they are to be bound to. This allows us to prepare a statement once, and execute it many times with different sets of data.

The Cheetah book discusses `prepare` and `do` statements on pages 106 - 111.

execute

Once we have a prepared statement handle we can *execute* it. We provide the *bind values* as arguments to the `execute` method. This allows us to call `execute` as many times as we need, providing different values each time. For example:

```
foreach my $project (’ABC’, ’XYZ, ’NMO’) {
        $sth->execute($project)
                or die "Failed to execute statement: ". $dbh->errstr;

        # fetch result and continue.
}
```

dump_results

`dump_results` dumps out the results of a query in an easy to view format. It is convenient for checking that the results of a query match that which is expected. `dump_results` is not intended for use in production, but rather as a programming and debugging aid. Query results are truncated to maintain a tidy output.

To use `dump_results` we can write the following.

```
# Replace $statement with your SQL statement (usually a SELECT)

my $sth = $dbh->prepare($statement) or die $dbh->errstr;
$sth->execute() or die $dbh->errstr;
$sth->dump_results();
```

Exercise

1. Write a program that selects `StaffID`, `Position` and `Wage` for all users in our database. Use `dump_results` to display the result of the select statement.
2. Now select the contents of the `staffphone` table, and again use `dump_results` to display the contents.
3. Now (using `prepare` and `execute`) change everyone's wages according to their `Position`. Wages are now organised as follows:

    - Manager: `180`
    - Admin: `130`
    - Devel: `120`

    Use your statement handle from 1. to display the new wages and verify your changes were successful.

    Answers for the above three questions can be found in `exercises/answers/dump_results.pl`.
4. Write a program which connects to the database and allows the user to add phone numbers for employees. Use placeholders and binding variables.

    You may wish to use: `exercises/insert.pl` to get you started. You can use your program from exercise 2 to verify your results if you wish.

    An answer for this exercise can be found in `exercises/answers/placeholders.pl`.

`do` VS `prepare`

As we've mentioned in the previous sections, when you wish to execute the same SQL multiple times you should use `prepare` and `execute` rather than `do`. The reason for this is that `do` internally calls `prepare` and `execute` itself. So performing five similar SQL statements one after the other, using `do` creates five new statement handles, prepares the statement five times, executes it five times and then destroys the handles for each of those five times. This can significantly increase execution time.

Preparing your statement once, and then executing it five times, saves the creation and destruction of four statement handles, increasing the speed of your program.

Statement handles

In the above discussion of `prepare` and `execute` we've used *statement handles*. The `prepare` method returns a statement handle upon which we can call certain methods, including `execute`.

A statement handle is a child object of the database object which encapsulates the specific SQL statement to be executed. You can have any number of statement handles in your script at any one time and they all work independently of each other. Just like database handles, certain database specific information can be passed to statement handles when necessary.

By convention we name generic statement handles `$sth`. However you should alway aim to give your statement handles names which accurately reflect the statement they contain.

The Cheetah book covers statement handles on pages 81 and 146.

Fetching data

There are many methods to fetch data out of the database with `DBI` and many ways to use those methods. The Cheetah book covers these in much more detail from page 111.

How many rows?

Often you want to find out how many rows a query returned or affected. For this we have the `rows` method.

```
my $rows = $sth->rows;
```

It's important to note that this value can generally only be relied upon after a non-SELECT execute (for example UPDATE and DELETE), or after fetching all the rows of a SELECT statement.

For SELECT statements, some databases will not know how many rows will be returned except by fetching them all. Some drivers will return the number of rows the application has fetched so far, but others may return -1 until all rows have been fetched. Others still will attempt to guess the number of rows, or make another call to the database to get the correct answer. Research the behaviour of your DBD before relying on the result from `rows`.

A more portable (and often faster) way if only the number of rows is desired is to use the `SELECT COUNT(...)` SQL construct.

Selecting rows

The following methods can be used to pull data out of a statement handle once you have called `execute` on it. Each of these methods return false when no more rows remain, but they can also return false when a database error has occurred.

fetchrow_array

`fetchrow_array` fetches the next row of data and returns it as a list containing the field values. All null values are translated to `undef`.

```
# Print names for all of our employees

my $sth = $dbh->prepare("SELECT FirstName, LastName FROM staff")
          or die $dbh->errstr;
$sth->execute() or die $dbh->errstr;

while(my @name = $sth->fetchrow_array) {
        print "$name[0] $name[1]\n";
}
```

The behaviour of `fetchrow_array` is undefined when called in scalar context. You may get the value of the first column selected, or the last. Don't do this.

fetchrow_arrayref

`fetchrow_arrayref` fetches the next row of data and returns a reference to an array which holds the field values. Null values are translated to `undef`. This method is much faster than calling `fetchrow_array` as no copying occurs.

```
# Print names for all of our employees

my $sth = $dbh->prepare("SELECT FirstName, LastName FROM staff")
          or die $dbh->errstr;
$sth->execute() or die $dbh->errstr;

while(my $name = $sth->fetchrow_arrayref) {
        print "$name->[0] $name->[1]\n";
}
```

When using `fetchrow_arrayref` the same piece of memory is used for each fetch. Thus the following won't work as you might hope:

```
# THIS WILL NOT WORK
my @stored_data;
while(my $result = $sth->fetchrow_arrayref) {
        push @stored_data, $result;
}
```

Instead you must copy the data yourself.

```
# DO THIS INSTEAD
my @stored_data;
while(my $result = $sth->fetchrow_arrayref) {
        push @stored_data, [@$result];
}
```

You can also use Storable's `dclone` to make this a little nicer. `dclone` provides a deep copy of a structure.

```
# THIS IS ALSO GOOD
use Storable qw(dclone);

my @stored_data;
while(my $result = $sth->fetchrow_arrayref) {
        push @stored_data, dclone $result;
}
```

fetchrow_hashref

`fetchrow_hashref` fetches the next row of data and returns a reference to a hash which holds the field values indexed by column name. Null values are translated to `undef`. This method is a little less efficient than either `fetchrow_arrayref` or `fetchrow_array` but usually the convenience vastly outweights the performance hit.

`fetchrow_hashref` provides data structures that are very easy to work with. Fields are referred to by name, not position, and so selecting extra fields requires little or no change to existing code. An example hashref from `fetchrow_hashref` would look like:

```
$result = {
        FirstName => "Bilbo",
        LastName  => "Baggins",
        Address   => "The Hill",
        City      => "Hobbiton",
        State     => "West Farthing",
}
```

Here’s an example of `fetchrow_hashref` in action:

```
# Print names and addresses for all of our employees.
my $sth = $dbh->prepare("SELECT FirstName, LastName, Address, City, State
                         FROM staff") or die $dbh->errstr;
$sth->execute() or die $dbh->errstr;

while(my $result = $sth->fetchrow_hashref()) {

        print "$result->{FirstName} $result->{LastName}\n",
              "$result->{Address}\n",
              "$result->{City} $result->{State}\n\n";

}
```

Error handling

It is always a good idea to check whether you stopped getting data because you exhausted the results returned by your statement or because the database failed. One way to do this is to check the return value of `$sth->err`, after we have read all of our data

```
while(my $result = $sth->fetchrow_hashref()) {
        ...
}

# Are we here because we’ve gone through all the data or because
# there was an error?
if( $sth->err ) {       # an error has occurred
        die "Failed to fetch all data: ". $sth->errstr;
}
```

Exercise

1. Ask the user for a city. Fetch details of all staff members who live in that city using:
   a. fetchrow_array()
   b. fetchrow_arrayref()
   c. fetchrow_hashref()

   and display them.
2. Ask the user for a StaffID. Return all of the phone numbers for that given ID. You may find it useful to refer to your insert.pl program from earlier.

   Use prepare, execute and one of the fetchrow_* methods.

You can find answers for these questions in exercises/answers/fetchrow.pl.

Advanced exercise

If you have time, try this exercise.

1. Modify the program found in exercises/update.pl to allow users to view all staff records and make appropriate changes.

An answer can be found in exercises/answers/update.pl.

Selecting a single row or column

Sometimes we just need to fetch a single row, or one or more values in a single column. For this work DBI provides a few shortcuts. Each of these methods combine the calls prepare and execute with a fetch operation.

These methods return the empty list or undef upon failure. As always, make sure you check your return value rather than assuming all is okay.

These functions have the same semantics as the do method discussed earlier. They take three arguments: an SQL statement, a hash reference of hints for the database, and the list of bind values matching the placeholders in the SQL statement. In all cases you can pass in a pre-prepared statement handle instead of the SQL string if available. In this case the prepare stage is skipped.

selectrow_array

selectrow_array fetches a single row into an array (similar to fetchrow_array).

```
my @row  = $dbh->selectrow_array($statement, \%attr, @bind_values);

my @result = $dbh->selectrow_array(
                "SELECT FirstName, LastName, Wage
                 FROM staff
                 WHERE StaffID = ?",
                 undef,
                 $staffid)
             or die "Failed to select row:". $dbh->errstr;

# print FirstName:
print $result[0];
```

The behaviour of `selectrow_array` is undefined when called in a scalar context. You may get the value of the first column, or the last. Don't do this.

selectrow_arrayref

This method fetches a single row into an array reference (similar to `fetchrow_arrayref`).

```
$ary_ref = $dbh->selectrow_arrayref($statement, \%attr, @bind_values);

my $result = $dbh->selectrow_arrayref(
                "SELECT FirstName, LastName, Wage
                FROM staff
                WHERE StaffID = ?",
                undef,
                $staffid)
             or die "Failed to select row:". $dbh->errstr;

# print FirstName:
print $result->[0];
```

selectrow_hashref

This method fetches a single row into a hash reference (similear to `fetchrow_hashref`.

```
$hash_ref = $dbh->selectrow_hashref($statement, \%attr, @bind_values);

my $result = $dbh->selectrow_hashref(
                "SELECT FirstName, LastName, Wage
                FROM staff
                WHERE StaffID = ?",
                undef,
                $staffid)
             or die "Failed to select row:". $dbh->errstr;

# print FirstName:
print $result->{FirstName};
```

selectcol_arrayref

This mthod fetches a single column from all the rwos matching the WHERE filter. It returns an array reference of these values, similar to using `fetchrow_arrayref`. Read the `DBI` documentation to learn how to select multiple columns using this method.

```
$ary_ref = $dbh->selectcol_arrayref($statement, \%attr, @bind_values);

my $result = $dbh->selectcol_arrayref(
                "SELECT LastName
                FROM staff
                WHERE Wage > ? order by Wage",
                undef,
                $wage)
             or die "Failed to select column:". $dbh->errstr;

# Print richest and poorest
print "Poorest: $result->[0], Richest: $result->[-1]\n";
```

Exercise

1. Modify a previous program to use one of the `select*` methods rather than `prepare` and `execute`.

Selecting everything

Occasionally it's useful to get all of the rows returned by the database at once. This allows you to perform processing on the rows in a block before finishing them with them. An example might be a batch script which copies all new entries in a database into a different database. In this case it may be more efficient to ask for all the entries rather than calling the fetch method for each and every row. However this can also use significantly more memory than a row-by-row approach.

`DBI` provides the following methods for this purpose:

1. `fetchall_arrayref`
2. `fetchall_hashref`
3. `selectall_arrayref`
4. `selectall_hashref`

As you should expect, the two `fetchall_` methods work on a pre-prepared and executed statement handle, while the two `selectall_` methods can be called directly on the database handle.

`selectall_hashref` is a special case of these methods. In all the previous cases we've received either a single result or an array of results. `selectall_hashref` instead returns a hash of results. As such, it takes an additional, mandatory, argument. This tells it which single field to use as the key for each record. If your table has a single field as it's primary key (such as StaffID for our staff table) then use this.

```
my $staff = $dbh->selectall_hashref(
                "SELECT StaffID, FirstName, LastName, Wage
                FROM staff
                WHERE City = ?",
                "StaffID",              # hash key
                undef,
                $city)
             or die "Failed to select data:". $dbh->errstr;

# Do something with $staff
foreach my $staffid (keys %$staff) {
        print "$staff{$staffid}{FirstName} $staff{$staffid}{LastName}\n";
}
```

Reading large data sets into memory can slow your system down or exhaust your process's memory. These methods should only be used if you are certain that the resulting memory footprint is going to be manageable. The `fetchall_arrayref` method optionally takes an argument to limit the number of rows fetched. It can then be called again to fetch more rows.

For more information about selecting large data sets from the database read the documentation for these methods in **perldoc DBI**.

Stored procedures

Anything you can do with your database directly, you can do through `prepare` and `execute` or the `select*` methods. Hence, if your database has a stored procedure called `get_abc` which takes three values (`$x`, `$y` and `$z`) and returns a single row, you can call it as follows:

```
my @array = $dbh->selectrow_array("CALL get_abc(?, ?, ?)", undef, $x, $y, $z)
        or die $dbh->errstr;

print "@array\n";
```

Calling `finish`

Generally when working with statement handles we want to use all of the data found by our SELECT. However, sometimes we may wish to stop working with a statement handle early. To tell the database that we're finished with the statement handle - regardless of any further data - we call the `finish` method:

```
$sth->execute() or die $dbh->errstr;

for ( my $i = 0; $i < 5; $i++ ) {
        if( @array = $sth->fetchrow_array() ) {
                print "@array\n";
        }
}
$sth->finish;   # end statement even if we haven't used all the data
```

`finish` is automatically called whenever a fetch method returns an end-of-data status, so most of the time you won't need to use it. Likewise calling `execute` again on your statement handle automatically finishes the previous call. `finish` is also called when a statement handle goes out of scope and gets destroyed.

If you don't call finish on statement handles which still have unprocessed data you will receive the following warning when you disconnect:

```
disconnect invalidates 2 active statement handles
(either destroy statement handles or call finish on them before disconnecting)
```

You can read more about the `finish` method on page 117 of the Cheetah book.

Disconnecting from the database

Just as it's a good idea to close a file handle once you've finished with it, it's also a good idea to disconnect from the database when you no longer need it. Perl will call disconnect on all databases at the end of your program's execution if you haven't done so yourself.

The disconnect method is a database handle method and is called as follows:

```
$dbh->disconnect();
```

Exercise

1. Add a call to `disconnect` in your `insert.pl` program.

Chapter summary

- To establish a connection to the database we use the `connect` method.
- We must remember to tell `DBI` our desired commit behaviour. We can do this by including the `{ AutoCommit => 1 }` flag when we call `connect`.
- To perform stand alone non-`SELECT` SQL queries we can use the `do` method.
- To include placeholders in an SQL statement we place question marks (`?`) where we wish to add data. We then pass the data to the method as bind values.
- When we need to perform a statement which is essentially the same each time, or when we wish to select data out from the database we use the `prepare` and `execute` methods.
- Data can be fetched from the database by using the `fetchrow_array`, `fetchrow_arrayref` and `fetchrow_hashref` methods.
- To disconnect from the database we use the `disconnect` method.

Chapter 6. SQL::Abstract

In this chapter...

SQL is an expressive and heavily used language for working with relational databases, however for a software developer it does present some problems. Very often we may have information contained in a data structure that we wish to use as the basis of a query, but transforming that data into SQL can be challenging.

This task is particularly difficult when the data may contain a variable number of fields and constraints, and the job of managing the SQL generation is often left to the individual developer. Hand generating is not only tedious, it can also be error prone.

In this chapter we will examine `SQL::Abstract`, a Perl module for automatically generating SQL queries for use with DBI.

Using SQL::Abstract

The `SQL::Abstract` module is available from the CPAN. It is uses an object-oriented interface for controlling options and generating SQL. Creating a `SQL::Abstract` object is straightforward:

```
use SQL::Abstract;

my $sql = SQL::Abstract->new();
```

Insert statements

Using `SQL::Abstract` to generate an insert statement is simple to use and understand. Provided with a hash of key/value pairs, `SQL::Abstract` will generate a corresponding insert statement. For example, using the following code:

```
my %record = (
        FirstName => ’Buffy’,
        LastName  => ’Summers’,
        Address   => ’1630 Revello Drive’,
        City      => ’Sunnydale’,
        State     => ’California’,
        Position  => ’Slayer’,
        Wage      => 50000
);

my ($stmt, @bind) = $sql->insert(’staff’,\%record);
```

would generate:

```
$stmt = "INSERT INTO staff
                (FirstName, LastName, Address, City,
                State, Position, Wage)
                VALUES (?, ?, ?, ?, ?, ?, ?)";

@bind = (’Buffy’,’Summers’,’1630 Revello Drive’,
        ’Sunnydale’,’California’,’Slayer’,50000);
```

These variables can now be used directly with DBI:

```
my $sth = $dbh->prepare($stmt);
$sth->execute(@bind);

# Alternatively:

$dbh->do($stmt,undef,@bind);
```

Being able to turn a hash into SQL can save a large amount of time both writing SQL and selecting bind values in the correct order. It also produces a more flexible program, as we can accommodate schema changes just by altering the data structure passed to `SQL::Abstract`.

We can also pass `SQL::Abstract` an array reference when generating insert queries. This produces an insert statement without column names:

```
my @record = (undef, ’Buffy’,’Summers’,’1630 Revello Drive’,
              ’Sunnydale’,’California’,’Slayer’,50000);
my ($stmt, @bind) = $sql->insert(’staff’,\@record);

# This produces:

$stmt = "INSERT INTO staff VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)";
@bind = (undef, ’Buffy’,’Summers’,’1630 Revello Drive’,
        ’Sunnydale’,’California’,’Slayer’,50000);
```

Select statements

Select statements are much more complex than simple inserts, as they almost always have a complicated *where* clause that needs to be considered. `SQL::Abstract` allows the generation of very detailed *where* statements, however we’ll only examine the most commonly used features in these notes.

The basic select generator is used by calling

```
my ($stmt, @bind) = $sql->select($table, \@fields, \%where, \@order);
```

Only the table and fields are required. The fields is a list of fields to retrieve, although `*` can be used to select all fields.

The *where* statement is based upon the contents of the hash reference provided. The two most commonly used formats are those of simple equality (represented by a simple key/value pair), and checking for the existence in a set (by using an arrayref as the value). The following example demonstrates both, and finds all Slayers, Watchers, and Software Engineers living in Sunnydale.

```
my @fields = qw(FirstName LastName);
my %where = (
        City     => ’Sunnydale’,
        Position => [’Slayer’,’Watcher’,’Software Engineer’],
);

my ($stmt, @bind) = $sql->select(’staff’,\@fields,\%where);

# This generates the following:
$stmt = ’SELECT Firstname, LastName FROM staff WHERE
        City = ? AND (Position = ? OR Position = ? OR
        Position = ?)’;

@bind = (’Sunnydale’,’Slayer’,’Watcher’,’Software Engineer’);
```

It should be noted that when an array reference is used as a value, the alternatives are *OR*ed together, however the entire statement is joined using *AND*.

`SQL::Abstract` also makes it possible to perform more complex operations than simple equality, although the syntax also becomes correspondingly more complex. To do so, we must use a hashref of `comparison => value` pairs. The following code demonstrates this to search for the same set of staff members, but only those with a wage of less than 100,000.

```
my %where = (
        City     => 'Sunnydale',
        Position => ['Slayer','Watcher','Software Engineer'],
        Wage     => { '<', 100000 },
);

# This would generate

my ($stmt, @bind) = $sql->select('staff',\@fields,\%where);

# This generates the following:

$stmt = 'SELECT Firstname, LastName FROM staff WHERE
         City = ? AND (Position = ? OR Position = ? OR
         Position = ?) AND Wage < ?';

@bind = ('Sunnydale','Slayer','Watcher','Software Engineer', 100000);
```

Being able to convert data structures into complex *where* statements is particularly useful when writing search interfaces, which is traditionally a difficult task to do with DBI alone.

Exercise

Use `SQL::Abstract` to write a program which:

1. Selects all staff members which live in Sydney.
2. Selects all staff members which live in either Sydney or Melbourne.
3. Selects all StaffIDs for staff on projects ABC or XYZ that have an allocation of 50% or more to that project.

Answers for the above problems can be found in `exercises/answers/sql-abstract.pl`.

Other statements

`SQL::Abstract` also supports *update* and *delete* statements, using the same principles as those in *select* and *insert* shown above. The syntax is simply:

```
my %fieldvals = (
        Address   => 'Room 214, Stevenson Hall, '.
                     'University of California',
        Position  => 'Slayer + Student',
);
my %where = (
        FirstName => 'Buffy',
        LastName  => 'Summers',
);
```

```
my ($stmt, @bind) = $sql->update($table,\%fieldvals,\%where);

my ($stmt, @bind) = $sql->delete($table,\%where);
```

`SQL::Abstract` also provides a *where* method, which only generates a *where* clause. This is particularly useful when you have an otherwise static query but a potentially variable WHERE clause. It’s used in the same way as other methods, except `$stmt` only contains the where clause.

```
my @order =  (’StaffID’);

my ($stmt, @bind) = $sql->where(\%where, \@order);

# This generates the following:

$stmt = ’WHERE ( FirstName = ? AND LastName = ? ) ORDER BY StaffID’;

@bind = (’Buffy’, ’Summers’);
```

Performance

When performing similar queries repeatedly, it can be inefficient to regenerate the SQL every time, particularly if we’re performing a lot of work. One example where regenerating SQL is loading a large amount of data from a file; the SQL will remain the same, but the data records will change each time. This is particularly important as DBI allows a single statement handle to be used for multiple queries, saving a large amount of processing time.

`SQL::Abstract` provides a special method called `values` that simply returns the list of bind values that would be generated by any other `SQL::Abstract` method call.

The following code example generates the SQL code and statement handle only once, and then uses a call to `$sql->values` to prepare the data for insertion each time:

```
my @data_set = (
        {
                FirstName => ’Buffy’,
                LastName  => ’Summers’,
                Address   => ’1630 Revello Drive’,
                City      => ’Sunnydale’,
                State     => ’California’,
                Position  => ’Slayer’,
                Wage      => 50000
        },
        {
                FirstName => ’Willow’,
                LastName  => ’Rosenberg’,
                Address   => ’Room 214, Stevenson Hall’,
                City      => ’Sunnydale’,
                State     => ’California’,
                Position  => ’Witch’,
                Wage      => 50000
        },
        # etc
);
```

```
my ($sth, $stmt);

for my $hash_ref (@data_set) {
        $stmt ||= $sql->insert(’table’,$hash_ref);
        $sth  ||= $dbh->prepare($stmt);
        $sth->execute($sql->values($hash_ref));
}
```

Of course, if the absolute maximum speed is needed, it’s still faster to hand-craft the SQL and data preparation, however the above technique provides a significant gain without much extra effort.

Conclusion

The `SQL::Abstract` module allows Perl data structures to be converted into SQL, saving the developer from having to write SQL by hand. Using `SQL::Abstract` is particularly useful when writing search interfaces, when dealing with data from a variety of sources, or when working with existing Perl code that already returns data in structure that `SQL::Abstract` can process.

More information on `SQL::Abstract` can be found at http://search.cpan.org/perldoc?SQL::Abstract .

Chapter 7. Practical exercise - Using the DBI

Being able to effectively use Perl's DBI module is more than just knowing the theory; it requires practical experience, too. The following exercises are modelled on a real-world project. It may use a couple of technologies you may not be familiar with, so ask your trainer if you have any questions. Don't forget that you can look-up the documentation associated with any module using the `perldoc` command.

The problem

Your client has hundreds of images which they'd like to keep in a searchable archive. The following schema is used for the images themselves:

```
create table images (
       image_id       INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
       image_name       varchar(100),
       description      text,
       photographer     varchar(100),
       photograph_date  date,
       copyright_holder varchar(100),
       image_location   varchar(200),
       original_media   varchar(100),
       image_colour     varchar(50),
       prepared_by      varchar(100)
);
```

Images from the archive are sometimes printed in publications, and your client would like to keep track of these. Not all images have been published, and some have been published multiple times. The following schema is used:

```
create table publications (
        image_id         INT UNSIGNED NOT NULL REFERENCES images(id),
        publication_name varchar(100),
        publication_date date,
        issue            varchar(100),
        page             varchar(20)
);
```

An image can be a member of multiple categories. For example an image of a rose may be a member of both the "flowers" and "plants" categories. Categories exist in a flat format, there are no such things as sub-categories. When editing image information, the client wishes to be able to choose whether to pick a category from a list, or create a new one. The categories table only has two columns:

```
create table categories (
        image_id        INT UNSIGNED NOT NULL REFERENCES images(id),
        category        varchar(200)
);
```

Implementation

Your client already has a sizeable database of image meta-data; the actual images are stored in directories on the filesystem. Some software already exists which makes use of the current database structure, and your client wishes to preserve compatibility with this software. As such, changes to the database schema are not possible.

Helper files

A project this large will take more than the few hours we have spare. As such we've already written a lot of the structural code for the following exercises.

These files will be found in your `exercises/images/` directory. Ignore the ones that start with `cdbi_` for now, they're used for an exercise later in this course.

Index page

Most of the functionality for this project can be accessed by the index page `exercises/images/www/index.html`. Your trainer will tell you the URL for accessing this page and the rest of your programs.

Exercise 1 - listing the images

The `exercises/images/cgi-bin/list_images.cgi` file should list all the existing images. It uses the template file `exercises/images/tmpl/list_images.html` for the HTML. You will *not* need to edit these files, they are mentioned only for reference purposes.

In addition to the above files you'll need the `exercises/images/lib/Image/Utility.pm` where we'll be adding all of our code for these exercises. This allows us to write subroutines which can then be used in more than one script.

1. Go to your index page as directed by your trainer. Follow the link to "list all images". Check that the script loads and returns an image. This is a placeholder image provided by our skeleton code.
2. Edit `connect_to_database()` in `Utility.pm` so that it connects to the database.
3. Edit `get_image_list()` in `Utility.pm` so that it returns an array of the images from the database, sorted by image_id.
4. Go back to the images listing and ensure that you have more images being showed.

Hints

If you find that your code isn't working check that the file compiles:

```
perl -c Utility.pm
```

If your code is loading but you're getting a SQL error, try testing that same SQL in mysql shell directly. If you're not sure what SQL is being generated turn on `ShowErrorStatement`. You may also want to turn on `RaiseError` as well.

Don't hesitate to ask your trainer for help if you need a hand.

Exercise 2 - displaying an image

For this you'll need the files `exercises/images/cgi-bin/show_image.cgi`, `exercises/images/tmpl/show_image.html`. Again, you shouldn't need to edit these as all of the work will be done in `exercises/images/lib/Image/Utility.pm`.

1. Click on one of the links from your `list_images.cgi` script. Check that your `show_image.cgi` script loads and shows the details of an image (even if it wasn't the one you clicked on).
2. Fill in `get_image()` in `Utility.pm`.
3. Fill in `get_categories()` in `Utility.pm`.
4. Fill in `get_publications()` in `Utility.pm`.
5. Check that your `show_image.cgi` script works over a number of the images.

Exercise 3 - editing images

For this you'll need the files `exercises/images/cgi-bin/edit_image.cgi` and `exercises/images/tmpl/edit_image.html`. All of the work will be done in `exercises/images/lib/Image/Utility.pm`.

1. From your `show_image.cgi` page, click the "Edit" button. This should present you with a similar page with editable data.
2. Fill in `distinct_categories()` in `Utility.pm` file so that it returns a list of all the *distinct* categories in the database, sorted alphabetically. Check that this worked.
3. Edit `update_image()` to allow editing of image details. There's some scaffolding code to help you out. Remember to use place-holders for user data! You are encouraged to use `SQL::Abstract`.
4. Test your changes by editing the title, description and copyright holder of an image. Check that the changes appear in the new results.
5. At the moment we don't allow editing of existing publications entries, however new publications can be added. Edit `add_publications()` to handle the addition of new publications.
6. Test your changes by adding a publication to a previously unpublished image. Check that it now displays that it was published in the image listing.
7. The web-page allows the addition and subtraction of categories as well as the creation of new categories. In such a situation it may be easier to delete all the categories for an image and then add all those submitted. Edit `update_categories()` so that it handles category changes.
8. Edit some images and test your functionality.

Chapter 8. Effectively using DBI Transactions

In this chapter...

Transactions are part of most modern databases. This chapter will cover how transactions can be used effectively to safeguard against errors and data corruption.

What is a transaction?

Consider the operation at a bank where `$100` is transferred from account `1234` to account `5678`. In SQL, this operation may be very simply represented by two statements:

```
UPDATE accounts SET balance = balance - 100
WHERE  acct_no = 1234;

UPDATE accounts SET balance = balance + 100
WHERE acct_no = 5678;
```

Now imagine the possibility where something goes wrong half-way through. The money has left account `1234`, but has not reached account `5678`, instead it’s just disappeared! This is an example where we either want all the statements to be executed, or none of them. The means by which we do this is by using a *transaction*.

A transaction executes a number of statements in an atomic fashion, ensuring that we can never commit to a half-completed state. It is an all-or-nothing affair. Either it all works, or it all fails. This is sometimes described as having A.C.I.D. properties:

- *Atomicity* All changes to the database in a transaction are atomic. Either they all happen or none of them happen. If for any reason the transaction cannot be completed the all changes made to the database are undone.
- *Consistency* Database consistency and integrity is preserved. That is, while a transaction may violate certain invariants in the data during execution, no other transaction will see these violations and all such inconsistencies will be eliminated by the time the transaction ends.
- *Isolation* To a given transaction it should appear as though it is running all by itself on the database. The effects of each transaction is invisible to all other transactions until the transaction is committed.
- *Durability* Once a transaction is committed its effects are guaranteed to persist even in the event of subsequent system failures. Prior to transaction commits transaction effects are guaranteed to be erased in the case of system failure.

Your database engine may also provide other guarantees for transactions. For example, your database may provide *serialisable isolation* for your transactions, ensuring that all transactions act as if they were executed one after the other. However in this course we only cover basic atomic transactions. It is also worth noting that not all databases have transaction capabilities.

Using transactions with DBI

The DBI module has an AutoCommit attribute, which is normally set when connecting. When set to a true value every statement will be committed to the database as soon as it is executed. When AutoCommit is set to false, statements are always considered part of a transaction, and will only be committed when explicitly told to do so.

While the value of AutoCommit is set to true in recent versions of DBI, this has not always been the case. As such, it is recommended that you always explicitly set the AutoCommit attribute when connecting:

```
use DBI;

# Connecting to the database without transactions
my $no_trans_dbh = DBI->connect("dbi:$driver:$dsn", $username, $password,
                                { AutoCommit => 1 });

# Every statement is committed automatically.

# Connecting to the database with transactions
my $trans_dbh = DBI->connect("dbi:$driver:$dsn", $username, $password,
                                { AutoCommit => 0 });

# We need to manually commit transactions when required.
$trans_dbh->commit();
```

☞ Trying to set AutoCommit to false results in an exception on databases that do not support transactions.

Transactions with AutoCommit set to true

A common choice is to set AutoCommit to true, and then to enable transactions only for the code that needs them. This can save the programmer the extra work of needing to make extra calls to commit, but care needs to be taken to ensure that actions that represent a transaction actually use them.

To begin a transaction when AutoCommit is turned on, you need to call the begin_work method on $dbh.

```
$dbh->begin_work or die "Could not start transaction - ".$dbh->errstr;
```

Transactions with AutoCommit set to false

If you have AutoCommit turned off, then you're always considered to be working with a transaction, so no special effort is required to begin. In fact, calling begin_work will return an error status if you do so when AutoCommit is off.

Ending the transaction

Whenever you're working with a transaction, it will end in one of two ways. You can *commit* your transaction to the database, or you can *rollback* your changes, as if they never occurred. In the situation where an SQL error occurs during your transaction, most engines will automatically issue a *rollback*, and will mark all subsequent SQL statements as an error until an explicit *rollback* is performed.

When using the `DBI`, you can use the database handle's `commit` method to commit your changes to the database.

```
# Connect and turn on transactions
my $dbh = DBI->connect($dsn,$user,$pass,{AutoCommit => 0})
        or die "Cannot connect to database ".DBI->errstr;

$dbh->do("UPDATE accounts SET balance = balance - 100
          WHERE  acct_no = 1234") or die $dbh->errstr;

$dbh->do("UPDATE accounts SET balance = balance + 100
          WHERE  acct_no = 4567") or die $dbh->errstr;

$dbh->commit or die "Failed to commit";
```

We can also use the `rollback` method to abort the changes to the database:

```
$dbh->rollback();
```

The transaction behaviour of `DBI` when `disconnect` is called is undefined. Some database drivers issue a `commit` while others issue a `rollback`. As a result good programming practices insist that you must explicitly `commit` or `rollback` your changes before disconnecting.

Fortunately most database drivers call `rollback` upon the database handle destruction, so if your program exits unexpectedly your transaction is not committed half-way through.

Exercises

1. The `exercises/trans_phones.pl` file provides staff members with new phone numbers - deleting the old numbers first. However if the database fails before all of the new numbers have been entered a staff member could be left with no numbers at all! Add transactions to the program so that if the number insertion fails, the previous numbers won't be deleted.

   An answer for this exercise can be found in `exercises/answers/trans_phones.pl`.
2. Write a script which deletes a staff member from all of the tables. As we do not wish to delete them from some tables and not others, use a transaction to ensure all work is atomic. Delete from the tables in the following order: projects, staffphone, staff.

Using exceptions with transactions

A problem with the traditional method of using transactions with `DBI` is that an automatic rollback occurs when there are problems with the SQL, but little consideration is given to what should occur should a fatal error result from your code. This matter requires serious thought -- if you were simply writing a sequence of fixed SQL statements you wouldn't need to use Perl (or any other language) at all.

Another problem commonly encountered with `DBI` programming is that every statement needs to be checked for errors. This is tiresome and error-prone, as can be seen by the example below:

```
my $dbh = DBI->connect($dsn,$user,$pass,{AutoCommit => 1})
        or die $DBI::errstr;

$dbh->begin_work or die $dbh->errstr;

my $sth = $dbh->prepare("SELECT qualification FROM students WHERE year = ?")
        or die $dbh->errstr;

$sth->execute($year) or die $dbh->errstr;

while (my $record = $sth->fetchrow_hashref) {
        # Do things with record.
}

# All done, or are we?
$dbh->commit or die $dbh->errstr;
```

It’s very easy to forget a guard condition, and in the code above we’ve actually forgotten a step. The `fetchrow_hashref` loop usually terminates because all records have been read, but it can also legitimately terminate because of an error, such as the database unexpectedly disconnecting. Unless we test `$sth->err` after the loop, we can’t tell.

Having to add the `or die ...` guards after each line also detracts from the readability of the code. Rather than showing the code’s operation in a straightforward manner, we need to mentally filter out the error conditions. It would certainly be nice if all the error-checking could be handled in one place, rather than on every single line.

Raising exceptions with `RaiseError`

Perl has the concept of *exceptions*, which allow us to separate our main code from error handling. The `DBI` also has excellent support for exceptions, and these can be turned on by setting the `RaiseError` attribute to true.

When `RaiseError` is true, any any error detected by the `DBI` will instead result in an exception being thrown. This is the same behaviour as if we had checked every statement by hand, and used `die` in the case of a failure. This immediately makes our code much tidier:

```
my $dbh = DBI->connect($dsn,$user,$pass, {
                AutoCommit         => 1,
                RaiseError         => 1,
                PrintError         => 0,
                ShowErrorStatement => 1,
});

$dbh->begin_work;

my $sth = $dbh->prepare("SELECT qualification FROM students WHERE year = ?");

$sth->execute($year);

while (my $record = $sth->fetchrow_hashref) {
        # Do things with record.
}

# All done.
$dbh->commit;
```

Turning `PrintError` off ensures that we don’t get our errors twice when we have an uncaught exception. `ShowErrorStatement` informs us of the SQL that was being executed when the error occured.

Besides from making our code cleaner, this also catches errors that we may have forgotten about, such as testing `$sth->err` after completing a fetch loop.

For more information on error handling with `RaiseError` refer to page 89 in the Cheetah book.

Catching exceptions with `eval`

Having cleaner code and more reliable error checking are worthy achievements in their own right. However what happens if we wish to be able to recover from errors in some way, rather than just having our program die outright? We can do this very easily by catching any exceptions thrown during our processing, and then taking the appropriate action.

In Perl we can catch any exceptions from a block of code by enclosing it with `eval { ... }`. If an exception is thrown, the block will terminate and the exception is placed into the special variable `$@`.

By using this method of exception handling, we’re now able to catch errors not just from the database, but also within our regular Perl code. Being able to gracefully recover from errors is important in high-availability applications, when importing data (where a single failure should not stop the operation of the whole import), or when dealing with a user-interactive application.

Here’s an example which can gracefully handle both database errors (such as duplicate rows) and business logic errors (invalid customer data).

```
my $dbh = DBI->connect(
        $dsn,$user,$pass,
        {AutoCommit => 0, RaiseError => 1,
         PrintError => 0, ShowErrorStatement => 1}
);

my $sth  = $dbh->prepare($SOME_SQL_INSERT_CODE);
my $sth2 = $dbh->prepare($SQL_FOR_SECOND_TABLE);

while (my $record = <RECORDS>) {
        eval {
                my ($fields1, $fields2) = process_record($record);
                $dbh->begin_work;
                $sth->execute(@$fields1);
                $sth2->execute(@$fields2);
                $dbh->commit;

        }; # The semi-colon here is required.

        # Error-handling.

        if (my $error = $@) {
                eval { $dbh->rollback; };  # Rollback current transaction.

                if ($error =~ /execute failed:|invalid record/) {
                        # Hmm, looks like our record was invalid or
                        # contained bad data.
                        # We’ll log that, and continue onwards.
                        log_record($record);
                }
                else {
                        # Some other kind of error?  We don't
                        # know how to deal with these, so we'll
                        # re-throw the exception.
                        die $error;
                }
        }
}
```

The above example is able to handle errors from both the database layer, and also exceptions that are thrown from inside the other subroutines called in the code.

Where to commit

It's essential to have our call to `commit` within the `eval` block. This is because `commit` can fail and if it does so, we want to make sure we `rollback` the transaction (if possible). Just because we haven't had errors prior to the `commit` doesn't mean that it won't fail. Databases are free to defer much of the real work until `commit` is called.

The `commit` call should usually be the very last thing before the end of the `eval` block.

When things go wrong

If your `eval` block exits with an error then the `$@` variable will be set to the error message. It's a good idea to print or save this error message immediately along with any other information you can provide. This means that if something else goes wrong you can still access your original error.

Protect the rollback

If your database suffers a serious error or has disconnected then calling `rollback` might itself raise an exception. If it does then the script will immediately exit or propogate the error to the next `eval`. To avoid a `rollback` triggering a further exception we must wrap it in an eval:

```
eval { $dbh->rollback; };   # This final ; is required
```

This is mostly useful if you want to perform some other kind of cleanup or include extra logging of the failure.

If you're happy with your program dying if the `rollback` fails then don't wrap it in that `eval`.

Exercises

1. Change `exercises/trans_phones.pl` to use `RaiseError` rather than calling `or die ...` each time.
2. Insert an error into your SQL and see what happens.
3. Wrap your transaction within an `eval` block and provide error handling by checking `$@`.
4. Catch and handle errors when they occur so that as many staff phone numbers as possible will be updated.

5. Insert into your `eval` block a call to a subroutine that does not exist. Run your code. What happens?

An answer for these exercises can be found in `exercises/answers/trans_phones2.pl`.

Chapter summary

- Transactions allow us to execute a number of SQL statements in an atomic fashion. This means that if any of the statements fail, it is as if none of them were executed.
- To turn transactions on with `DBI`, when `AutoCommit` is true, call the `begin_work` method.
- To commit a transaction use the `commit` method.
- To stop a transaction and undo your changes use the `rollback` method.
- Transactions protect us from errors in our SQL but not from errors in our SQL. By using exceptions, however, we can catch all errors and only `commit` if all went well.

Chapter 9. Database Security

In this chapter...

Databases are a common part of many modern applications. The most common way of integrating with a database in Perl is using the `DBI` module. This chapter covers a number of commonly seen programming mistakes with the `DBI` module.

SQL injection attacks

SQL injection attacks refers to any condition where an end-user may be able execute SQL commands without authority. Often these attacks are easy to execute, but luckily for us they're also easy to avoid.

The effects of a successful SQL injection attack can be revealing of data, corruption or destruction of data, or denial of service. In some rare circumstances an attacker may be able to even execute commands or code, depending upon the database in question and the privileges of the account being used.

The most common form of SQL injection attack involves feeding *SQL meta-characters* to a program. These meta-characters change the interpretation of the SQL being executed, in the same way that shell-meta-characters can change the execution of a shell command.

Most commonly the SQL quote character (*single-tick (')* on most systems) is used to try and prematurely escape from a string. The current command is then either altered or aborted, and a new command (or sequence of commands) are executed in its place.

The code below is an example of code that is vulnerable to an SQL injection attack:

```
my $dbh = DBI->connect($dsn,$user,$pass) or die "Cannot connect to DB\n";

my $username = <STDIN>;   # Could just as easily be from CGI
my $password = <STDIN>;
chomp($username,$password);

# Authenticate the user before allowing them access to their account.
# Verify that username and password match what is in the database.

# (DON'T DO THIS!)
my $result = $dbh->selectrow_hashref("
        SELECT account_details
        FROM   accounts
        WHERE  user = '$username' AND
               password = '$password'
");

# Work with $result...
```

The mistake that has been made is that both `$username` and `$password` may contain SQL meta-characters as these have not been escaped or cleaned in any way. Use of any variable interpolation inside an SQL statement is a strong warning sign that an SQL injection attack may be possible.

While many database drivers will only allow a single statement to be executed each request, this is not guaranteed to be the case. A simple `SELECT` statement can be manipulated by a clever attacker.

Using the example above, in some cases an attacker could gain access to any account (in this example we'll use 'buffy') by supplying a password of `' OR user = 'buffy' AND ' ' = '` . This results in the following statement being executed:

```
        SELECT account_details
        FROM   accounts
        WHERE  user = 'buffy' AND
               password = '' OR user = 'buffy' AND ' ' = ' '
```

The condition `' ' = ' '` is always true, and the injection attack succeeds, side-stepping password authentication. It's possible for successful attacks to be made against such SQL even when no knowledge of that underlying SQL is available.

A commonly seen, but less than ideal way of avoiding this problem is to use a regular expression to escape the meta-characters first, like so:

```
$username =~ s/'/\\'/g;
```

Unfortunately, this is also error-prone. Different databases will use different methods of quoting. The SQL standard uses a double apostrophe, so `don't` is quoted as `'don''t'`. Some databases require the use of a backslash instead. Some will accept either.

Aside from the inconsistency, this method of quoting places the onus upon the programmer to do the right thing. Mistakes do happen, and sometimes data ends up not being escaped, or sometimes it's escaped too many times, resulting in 'over-quoting' and data corruption.

None of this even begins to touch upon what occurs if we're dealing with binary data, which may not only require special quoting or handling, but may also contain characters (such as the null byte) which cannot be passed directly to many databases. If you're starting to think this is all too hard, then you'd be right.

Fortunately, there is a better way to do things. We can use placeholders.

The `DBI` module provides the concept of place holders, which stand in the place of data that would otherwise need quoting. Here's an example:

```
# Authenticate the user before allowing them access to their account.
# Verify that username and password match what is in the database.

my $result = $dbh->selectrow_hashref(q{
        SELECT account_details
        FROM   accounts
        WHERE  user = ? AND
               password = ?
}, undef, $username, $password);
```

Note we use a question-mark (`?`) as a *place holder* of where our data will appear. We do not use any quotes around the place holder, nor do we escape characters in the data that we are passing to the query. `DBI` handles these requirements for us, and does so in a way that is appropriate to the database that we are connected to.

Using place holders has another advantage as well, many databases are able to optimise repeated queries to run more quickly when place holders are used.

```
# We can prepare our statement once...
my $sth = $dbh->prepare("
        UPDATE accounts SET balance = 0 WHERE username = ?"
);
```

```
# ...and then use it many times.
foreach my $user (@users) {
        $sth->execute($user);
}
```

DBI and taint

Unless your Perl program has taint mode turned on, the taint features in `DBI` have no effect.

The `DBI` module is taint-aware, and from version 1.31 has supported the following attributes, which only have an effect when Perl is running in taint mode.

- `TaintIn`, when set, causes the arguments to most `DBI` method calls to be checked for taintedness. This means that you cannot insert tainted data into your database, or used tainted data as part of a `connect` call.
- `TaintOut`, when set, causes any data from fetch operations to be marked as tainted. In future versions, the results of other `DBI` calls may also return tainted data.
- `Taint` is simply a shortcut which allows the setting of both TaintIn and TaintOut at the same time.

Clearly, using `DBI`'s taint features can be a good idea. Just because data has been fetched from the database does not guarantee it's clean enough to swing past the shell on most systems. Likewise, insisting that you perform basic checks upon your data before inserting it into the database helps to ensure that no bad records are created.

```
#!/usr/bin/perl -wT
use strict;
use DBI;

my $dbh = DBI->connect($dsn, $user, $passwd, {
                AutoCommit => 1,
                RaiseError => 1,
                Taint      => 1
});

my $sth = $dbh->prepare(q{
        SELECT FirstName, LastName
        FROM StaffAddress
        WHERE StaffID = ?
});

print "Which staff id? ";
my $staffid = <STDIN>;
chomp $staffid;

$sth->execute($staffid);          # Dies with an error
```

In order to use data in an SQL query we have to untaint it. We do this by capturing it out of a regular expression.

```
my $dbh = DBI->connect($dsn, $user, $passwd, {
                AutoCommit => 1,
                RaiseError => 1,
                Taint      => 1,
});
```

```
my $sth = $dbh->prepare(q{
        SELECT FirstName, LastName
        FROM StaffAddress
        WHERE StaffID = ?
});

print "Which staff id? ";

my $staffid = <STDIN>;
chomp $staffid;

# Check that the staff id contains only digits
my ($safe_staffid) = ($staffid =~ /^(\d+)$/);

# If $safe_staffid is empty, the regex didn't succeed
if(not $safe_staffid) {
        die "staff id contains invalid characters!";
}

# No error
$sth->execute($staffid);
```

Likewise, data coming out of the database must be untainted before passing it to any function that cares about tainted data. These include `open` when opening files for writing, `system`, `exec` and many more:

```
my $results = $dbh->selectrow_hashref(q{
                SELECT FirstName
                FROM StaffAddress
                WHERE StaffID = ?
        },
        undef, 12345,
);

# Dies with an error
system("mkdir /tmp/$results->{FirstName}");
```

Exercises

1. Pass in `Taint => 1` to the `connect` method in your answer to a previous exercise. Run Perl under taint mode (`-T`) and observe what happens to your script.
2. Untaint your data before adding it to the database.

Temporarily disabling Taint

`DBI`'s taint features are very flexible; it's possible to turn `TaintIn` and `TaintOut` on and off for particular statement handles. For example, you may disable `TaintOut` for some select statements that you consider trustworthy.

```
#!/usr/bin/perl -wT
use strict;
use DBI;

$ENV{PATH} = ’/bin:/usr/bin’;

my $dbh = DBI->connect($dsn, $user, $passwd, {
                AutoCommit => 1,
                RaiseError => 1,
                Taint      => 1,
});

my $sth = $dbh->prepare(q{
        SELECT FirstName, LastName
        FROM StaffAddress
        WHERE StaffID = ?
});

print "Input staff id: ";
my $staffid = <STDIN>;
chomp $staffid;
unless(($staffid) = ($staffid =~ /^(\d+)$/)) {
        die "staff id contains invalid characters!";
}

$sth->{TaintOut} = 0;      # Data from _this_ statement handle is safe.
$sth->execute($staffid);   # We still need to pass _IN_ untainted data

my @row = $sth->fetchrow_array();       # @row is not tainted
system ( "echo @row" );                 # no error
```

Chapter summary

- SQL injection attacks refer to any condition where an end-user may be able execute SQL other than that intended.
- Using place holders in our SQL allows us to prevent SQL injection attacks.
- If we’re using taint checking in our program then we can also ensure that tainted data is not added to the database and that data from the database is considered tainted. This allows us to increase the security of our program.

Chapter 10. Debugging and Profiling

In this chapter...

As with any system, it's important to know how to find problems when things are going wrong or are taking longer than desired to execute. This chapter covers DBI's debugging and profiling options.

Basic debugging

Most of the common errors you'll encounter when working with DBI are due to problems with your SQL statements or a disparity between the number of bind values expected and the number of bind values provided.

When these errors occur most DBI methods return false to indicate failure. Thus it is common to see code such as:

```
my $sth = $dbh->prepare("SELECT * FROM table") or die $dbh->errstr;
```

As discussed in the chapter on DBI transactions it is possible to omit the test for error and instead ask DBI to throw an exception whenever an error occurs. We pass such requests in upon connection.

```
my $dbh = DBI->connect($dsn, $user, $pass, {
                AutoCommit         => 1,
                RaiseError         => 1,
                PrintError         => 0,
                ShowErrorStatement => 1,
}) or die DBI->errstr;
```

To re-cap, `RaiseError` asks for an exception to be thrown whenever a database error occurs. Turning off `PrintError` ensures that we don't get these errors twice when we have an uncaught exception. `ShowErrorStatement` ensures that we get to see the SQL that was attempted in case it is the cause of the error. If we don't capture the exception thrown by `RaiseError` our program will terminate printing the error to its STDERR handle.

Trace

For further information about DBI's trace functionality refer to pages 98-101, 199-20, and 202 in the Cheetah book. Further read the documentation of the DBI module

If you need to track down more difficult to find bugs, or if you wish to see what DBI is doing with your query, you can use DBI's runtime tracing information. There are fifteen tracing levels, each providing more and more information.

Table 10-1. Trace levels

Trace level	Information
0	Disables trace.
1	Trace DBI method calls returning with results or errors. Good for a simple overview of what's happening.
2	Trace method entry with parameters and returning with results. Good general purpose trace level. Trace will default to this level once turned on.
3	As above, adding some high-level information from the driver and some internal information from the DBI.
4	As above, adding more detailed information from the driver.
5 to 15	As above but with more and more detailed information.

Typically trace level 2 will provide you with sufficient information to identify the problems you are experiencing. In some cases where you need even more information you may find higher levels (up to 9) more useful. Levels 10 through to 15 are typically only required for debugging DBI and its drivers.

Trace levels can be set for all DBI operations from that point onwards as well as for any DBI handle. Setting the trace level on a handle overrides any previous trace level which might have applied. This makes it possible to have multiple trace levels occurring for different parts of your program.

Let's see trace in action. The below code can be found in `exercises/trace.pl`.

```
my $dbh = DBI->connect($dsn, $username, $passwd, {
                AutoCommit => 1,
                PrintError => 0,
                ShowErrorStatement => 1,
                RaiseError => 1,
}) or die DBI->errstr;

# Set global trace level to 2.
DBI->trace(2);

# Prepare and execute a simple statement
my $succeeds = $dbh->prepare("select Firstname from Staff");
$succeeds->execute();

# Prepare and execute a simple statement which will fail.
my $fails = $dbh->prepare("select Title from Staff");
$fails->execute();


__END__
    DBI 1.21-nothread dispatch trace level set to 2
    -> prepare for DBD::mysql::db (DBI::db=HASH(0x817c8a8)~0x817c908 'select Firstname from Staff')
Setting mysql_use_result to 0
    <- prepare= DBI::st=HASH(0x817bb70) at trace.pl line 27
    -> execute for DBD::mysql::st (DBI::st=HASH(0x817bb70)~0x82afbe4)
    -> dbd_st_execute for 0817ba14
    <- dbd_st_execute 11 rows
    <- execute= 11 at trace.pl line 28
    -> prepare for DBD::mysql::db (DBI::db=HASH(0x817c8a8)~0x817c908 'select Title from Staff')
```

```
Setting mysql_use_result to 0
    <- prepare= DBI::st=HASH(0x817d784) at trace.pl line 30
    -> execute for DBD::mysql::st (DBI::st=HASH(0x817d784)~0x82afbd8)
    -> dbd_st_execute for 081c82cc
Unknown column ’Title’ in ’field list’ error 1054 recorded: Unknown column ’Title’ in ’field list’
    <- dbd_st_execute -2 rows
    !! ERROR: 1054 ’Unknown column ’Title’ in ’field list”
    <- execute= undef at trace.pl line 31
DBD::mysql::st execute failed: Unknown column ’Title’ in ’field list’ [for statement “select Title f
    -> DESTROY for DBD::mysql::st (DBI::st=HASH(0x82afbe4)~INNER)
    <- DESTROY= undef
    -> DESTROY for DBD::mysql::st (DBI::st=HASH(0x82afbd8)~INNER)
    <- DESTROY= undef
    -> DESTROY for DBD::mysql::db (DBI::db=HASH(0x817c908)~INNER)
imp_dbh->svsock: 82c1e0c
    <- DESTROY= undef
    -- DBI::END
    -> disconnect_all for DBD::mysql::dr (DBI::dr=HASH(0x81c3b80)~0x817c86c)
    <- disconnect_all= ” at DBI.pm line 533
    -> DESTROY in DBD::_::common for DBD::mysql::dr (DBI::dr=HASH(0x817c86c)~INNER)
    <- DESTROY= undef during global destruction
```

As you can see above, trace generates a lot of information. Each line starting with `->` signals entry into a method. Each line starting with `<-` signals exit from a method. Key things to observe are:

- Line 1 contains version number of DBI, information about threading, and trace level.
- Query visible at start of prepare statement (line 2)
- Number of rows returned at exit of execute (line 8)
- Error propagation from second query.
- Error from RaiseError appearing just before destructors.

Sending trace information to a file

By default, DBI’s trace information is sent to STDERR. Should you instead wish to send it to a file, pass the filename as a second argument to `trace`.

```
DBI->trace(2, ’trace.log’);
```

Trace output filenames *cannot* be set on a per handle basis. Setting a file for any trace output results in all trace output going there.

Statement handles

Setting a trace level on a statement handle is just a matter of calling the `trace` method on the statement handle.

```
my $sth = $dbh->prepare("select LastName from Staff");

$sth->trace(3);

$sth->execute();
```

Setting a trace level on a statement handle provides you with all trace information from that handle and any extra handles created internally. Tracing on statement handles is usually preferred as it allows you to concentrate on the particular statements you wish to debug. To trace DBI methods which do not provide easy access to a handle, set the trace on your database handle.

```
# Turn on tracing
$dbh->trace(2);

my $results = $dbh->selectall_arrayref("select * from Staff where
                                       Lastname=’Smith’");

# Now turn off tracing
$dbh->trace(0);

__END__
    DBI::db=HASH(0x817c914) trace level set to 2 in DBI 1.21-nothread
    -> selectall_arrayref in DBD::_::db for DBD::mysql::db (DBI::db=HASH(0x817c8b4)~0x817c914 ’sele
2   -> prepare for DBD::mysql::db (DBI::db=HASH(0x817c914)~INNER ’select * from Staff where Lastnam
Setting mysql_use_result to 0
2   <- prepare= DBI::st=HASH(0x817d778) at DBI.pm line 1194
    -> execute for DBD::mysql::st (DBI::st=HASH(0x817d778)~0x817d808)
    -> dbd_st_execute for 081c832c
    <- dbd_st_execute 2 rows
    <- execute= 2 at DBI.pm line 1196
    -> fetchall_arrayref in DBD::_::st for DBD::mysql::st (DBI::st=HASH(0x817d778)~0x817d808 undef)
2   -> fetch for DBD::mysql::st (DBI::st=HASH(0x817d808)~INNER)
    -> dbd_st_fetch for 081eb44c, chopblanks 0
      Storing row 0 (12349) in 0817d940
      Storing row 1 (Sam) in 0817d934
      Storing row 2 (Smith) in 0817d928
      Storing row 3 (55 Queens Ave) in 0817d91c
      Storing row 4 (Sydney) in 0817d910
      Storing row 5 (NSW) in 0817d904
      Storing row 6 (Admin) in 0817d8f8
      Storing row 7 (130) in 0817d8ec
    <- dbd_st_fetch, 8 cols
2   <- fetch= [ ’12349’ ’Sam’ ’Smith’ ’55 Queens Ave’ ’Sydney’ ’NSW’ ’Admin’ ’130’ ] row1 at DBI.pm
2   -> fetch for DBD::mysql::st (DBI::st=HASH(0x817d808)~INNER)
    -> dbd_st_fetch for 081eb44c, chopblanks 0
      Storing row 0 (12350) in 0817d940
      Storing row 1 (Ann) in 0817d934
      Storing row 2 (Smith) in 0817d928
      Storing row 3 (10 Albert St) in 0817d91c
      Storing row 4 (Brisbane) in 0817d910
      Storing row 5 (QLD) in 0817d904
      Storing row 6 (Devel) in 0817d8f8
      Storing row 7 (75) in 0817d8ec
    <- dbd_st_fetch, 8 cols
2   <- fetch= [ ’12350’ ’Ann’ ’Smith’ ’10 Albert St’ ’Brisbane’ ’QLD’ ’Devel’ ’75’ ] row2 at DBI.pm
2   -> fetch for DBD::mysql::st (DBI::st=HASH(0x817d808)~INNER)
    -> dbd_st_fetch for 081eb44c, chopblanks 0
2   <- fetch= undef row2 at DBI.pm line 1400
    <- fetchall_arrayref= [ ARRAY(0x817d9c4) ARRAY(0x817da30) ] row2 at DBI.pm line 1204
    <- selectall_arrayref= [ ARRAY(0x817d9c4) ARRAY(0x817da30) ] at trace.pl line 35
    -> DESTROY for DBD::mysql::st (DBI::st=HASH(0x817d808)~INNER)
    <- DESTROY= undef
    -> DESTROY for DBD::mysql::db (DBI::db=HASH(0x817c914)~INNER)
imp_dbh->svsock: 82b5894
    <- DESTROY= undef
```

Exercises

Use an existing program for these exercises.

1. Turn on tracing for a program of your choice. If you don't have any working programs, use `exercises/answers/update.pl`.
2. Compare tracing for the whole program with just tracing on one statement handle.
3. Change the tracing to send output to a file rather than to STDERR.
4. Try trace levels 0, 1 and 3 and compare the differences in information.

The DBI_TRACE environment variable

To turn on tracing without editing your code you can use the `DBI_TRACE` environment variable. Doing this turns on tracing globally for any programs you then execute. The `DBI_TRACE` environment variable can be set to any of the following values:

Table 10-2. DBI_TRACE values

Value	Equivalent
N	`DBI->trace(N);` where *N* is a number between 0 and 15.
filename	`DBI->trace(2, filename);`
N=filename	`DBI->trace(N, filename);`

For example, under Unix-like systems you can turn on tracing without editing your program as follows:

```
DBI_TRACE=1 ./myprogram.pl
```

Threaded Perl

If your Perl interpreter has been built to enable threading then the thread id will also appear in the trace output. This will appear at method entry, for example:

```
    -> selectall_arrayref in DBD::_::db for DBD::mysql::db
    (DBI::db=HASH(0x817c8b4)~0x817c914 'select * from Staff where
    Lastname='Smith'') thr0
```

`DBI` ensures that each `DBD` is only accessed by one thread at a time. This makes determining which thread is generating what trace output possible.

Profiling

Profiling is used to determine which parts of your code are taking up the most execution time. This allows us to best decide the which locations to focus our optimising efforts. For example, pretend some code uses two subroutines A and B. Through testing we determine that subroutine A takes 50 time-units to execute whereas subroutine B takes only 2 time-units.

The naive approach would be to assume that our optimising efforts are best spent on speeding up subroutine A. However, if we profile our subroutines when used under normal conditions, we may discover that subroutine A is only being called once per program invocation. On the other hand, subroutine B may be called hundreds of times. With such results, a 25% speed increase to subroutine B is likely to significantly outweigh a 50% speed increase to subroutine A.

Good profiling tools will provide information both on which subroutines take the longest to execute and which subroutines are called most often. Should you need to optimise your code, these tools can identify potential bottlenecks and focus your efforts where they can do most good.

Profiling Perl

For further information read the documentation for `Devel::DProf` and **dprofpp**. There is also a good article on profiling in Perl at Perl.com (http://www.perl.com/pub/a/2004/06/25/profiling.html).

There are a few tools to profile Perl code and which one is *best* depends on your current focus. A commonly used one is `Devel::DProf` and its post-processor **dprofpp** (these both come standard with Perl). To use these type:

```
% perl -d:DProf some_program.pl
% dprofpp
```

This will give you results similar to:

```
Total Elapsed Time = 1.229688 Seconds
 User+System Time = 0.559792 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 85.7   0.480  0.480    100   0.0048 0.0048  main::sub_b
 14.2   0.080  0.560      1   0.0799 0.5599  main::sub_a
 0.00   0.000 -0.000      1   0.0000      -  main::BEGIN
 0.00   0.000 -0.000      1   0.0000      -  strict::import
 0.00   0.000 -0.000      1   0.0000      -  strict::bits
```

The first two lines tell us how many seconds the program took to run. The first line says that the program finished 1.2 seconds after we started it and the second line tells us that had the the code been the only other process on the machine, it would have taken only 0.56 seconds. For the remaining 0.64 seconds the CPU was working on other things.

The first column is the percentage of time of total program invocation that the subroutine took. The exclusive seconds (second column) mark the time that the subroutine used itself without including data from subroutines it called. The cumulative seconds (third column) represent the entire time for the subroutine including further subroutine calls.

By looking at these columns we can determine that our `sub_b` subroutine was called 100 times, took an average of 0.0048 seconds each time for a total of 0.48 seconds. This resulted in it taking up 85.7% of the program execution time.

On the other hand, each call to `sub_a` took 0.08 seconds (exclusively), 16 times longer than `sub_b`. However, as `sub_a` was only called once this resulted in `sub_a` only taking up 14.2% of program execution time.

Removing `sub_a` would yield a 14% improvement. Increasing it's speed by 50% would yield a speed up of 7%. However, improving `sub_b` by only 25% would result in a massive 21% speed up for our program.

These results tell us that although `sub_a` takes up much more time, per call, than `sub_b`, `sub_b` still makes the best first candidate for optimisation.

Profiling DBI

Perl's profiling engine is very good, but it has limitations. In particular it isn't very helpful when we want to know which SQL queries are taking the most time, or how often a set of queries are run.

Fortunately, there are other tools we can use to do this for us. These are `DBI::ProfileDumper` and **dbiprof**. To use these we turn profiling on, run our code as normal and then use **dbiprof** to interpret the results.

For further information about profiling DBI read the documentation for `DBI::ProfileDumper`, `DBI::ProfileData`, **dbiprof** and `DBI::Profile`.

Profiling levels

In a similar manner to tracing, DBI profiling can be performed at a number of levels. In `DBI::Profile` parlance, these are called *paths*.

This Path value is used to control where the profile for a method call will be merged into the collected profile data. Whenever profile data is to be stored the current value for Path is used.

Table 10-3. DBI::Profile paths

Name and Integer value	Meaning
DBI 1	Merges all profile data into a single tree leaf. Ideal for getting an idea of total time spent on interacting with the database.
!Statement 2	Groups data by corresponding SQL statement. Ideal for finding out which statements are the most commonly called and how long each statement is taking.
!MethodName 4	Groups data by corresponding DBI method call. For example `prepare` and `execute`.
!MethodClass 8	Groups data by fully qualified name of the DBI method call, where multiple names exist for the same method these will be grouped together. For example `*DBD::Driver::db::prepare`.

Paths can be combined by adding their numerical values together. Thus a path value of `6` will order data by SQL statement and DBI method name.

The most frequently used path values are `2` and `6` as these yeild the most useful information.

Enabling profiling in our code

To turn profiling on, we assign a profiler to the `Profile` attribute on our database handle. In these examples we will use `DBI::ProfileDumper` (a subclass of `DBI::Profile`) as it helpfully outputs the profile to a file rather than printing it to STDOUT.

If we don't specify a path the default path of `2` (`DBIprofile_Statement`) will be used. If we don't specify a filename, the default filename of `dbi.prfo` will be used.

The methods below will work for any profiler which provides the same API as `DBI::Profile`, although it should be noted that although `DBI::Profile` accepts a file attribute, it does not use it.

```
use DBI;
# Profile with defaults (group by statement) and output file (dbi.prof)
$dbh->{Profile} = "DBI::ProfileDumper";

######################################

use DBI;
# As above, explicitly specifying file and path
$dbh->{Profile} = "2/DBI::ProfileDumper/File/dbi.prof";

######################################

use DBI::ProfileDumper;
# As above, using an object.
$dbh->{Profile} = DBI::ProfileDumper->new(
        Path => [ '!Statement' ],
        File => 'dbi.prof',
);

#####################################

use DBI::ProfileDumper;
# Group by method and statement
$dbh->{Profile} = DBI::ProfileDumper->new(
        Path => [ '!Statement', '!MethodName' ],
        File => 'dbi.prof',
);
```

Enabling profiling via an environment variable

If the `DBI_PROFILE` environment variable is set, then that value will be used to provide the value for `$dbh->{Profile}`. Its value should be set to a forward-slash delimited string specifying path level, profiler and file name. For example the following string:

```
DBI_PROFILE=2/DBI::ProfileDumper/File/dbi.prof
```

says to use path level 2 (group by statement), profiler `DBI::ProfileDumper` and filename `dbi.prof`.

Interpreting the results

Once we've enabled profiling, we run our code as normal. At the end, a file called `dbi.prof` will exist which records timing information about the code execution. This can be interpreted by using the **dbiprof** command.

The below results were generated by running `exercises/answers/update.pl`, which is a sample answer to the `insert.pl` program. In each case the staff members were listed, one was selected for update, details were changed and then all staff members were listed out again.

Example output from running with a path of `6` can look like this:

```
# --number 5 asks for only the top 5 actions.
% dbiprof --number 5 dbi.prof

DBI Profile Data (DBI::ProfileDumper 1.0)

  Program       : update.pl
  Path          : [ DBIprofile_Statement, DBIprofile_MethodName ]
  Total Records : 15 (showing 10, sorted by total)
  Total Count   : 37
  Total Runtime : 0.004613 seconds

#####[ 1 ]###########################################################
  Count         : 1
  Time          : 0.001449 seconds
  Key 1         :

UPDATE Staff SET
                FirstName = ?, LastName = ?,
                Address = ?, City = ?,
                State = ? WHERE StaffID = ?

  Key 2         : execute

#####[ 2 ]###########################################################
  Count         : 1
  Time          : 0.001384 seconds
  Key 1         :
  Key 2         : connect

#####[ 3 ]###########################################################
  Count         : 2
  Total Time    : 0.000543 seconds
  Longest Time  : 0.000331 seconds
  Shortest Time : 0.000212 seconds
  Average Time  : 0.000271 seconds
  Key 1         :

SELECT StaffID, FirstName,
        LastName, Address, City, State FROM Staff

  Key 2         : execute

#####[ 4 ]###########################################################
  Count         : 1
  Time          : 0.000386 seconds
  Key 1         :

SELECT FirstName, LastName, Address,
        City, State from Staff WHERE StaffID = ?

  Key 2         : execute

#####[ 5 ]###########################################################
  Count         : 14
  Total Time    : 0.000323 seconds
  Longest Time  : 0.000071 seconds
  Shortest Time : 0.000005 seconds
  Average Time  : 0.000023 seconds
  Key 1         :

SELECT StaffID, FirstName,
        LastName, Address, City, State FROM Staff

  Key 2         : fetchrow_array
```

The above results are truncated to only look at the top five database actions. At this level we actually have 15 actions recorded.

From the above information we can tell that we updated the staff table once and it took 0.001449 seconds to execute. We connected to the database once, selected out all Staff information twice and called `fetchrow_array` on our selected Staff information 14 times. Timing information for each activity is provided including longest time, shortest time and average time.

For comparative purposes, the same code run with the same actions with a path of `2` returns:

```
DBI Profile Data (DBI::ProfileDumper 1.0)

  Program       : update.pl
  Path          : [ DBIprofile_Statement ]
  Total Records : 4 (showing 4, sorted by total)
  Total Count   : 37
  Total Runtime : 0.004725 seconds

#####[ 1 ]###########################################################
  Count         : 11
  Total Time    : 0.001521 seconds
  Longest Time  : 0.001385 seconds
  Shortest Time : 0.000002 seconds
  Average Time  : 0.000138 seconds
  Key 1         :

#####[ 2 ]###########################################################
  Count         : 3
  Total Time    : 0.001481 seconds
  Longest Time  : 0.001407 seconds
  Shortest Time : 0.000029 seconds
  Average Time  : 0.000494 seconds
  Key 1         :

UPDATE Staff SET
                FirstName = ?, LastName = ?,
                Address = ?, City = ?,
                State = ? WHERE StaffID = ?


#####[ 3 ]###########################################################
  Count         : 19
  Total Time    : 0.001182 seconds
  Longest Time  : 0.000319 seconds
  Shortest Time : 0.000003 seconds
  Average Time  : 0.000062 seconds
  Key 1         :

SELECT StaffID, FirstName,
        LastName, Address, City, State FROM Staff


#####[ 4 ]###########################################################
  Count         : 4
  Total Time    : 0.000541 seconds
  Longest Time  : 0.000384 seconds
  Shortest Time : 0.000010 seconds
  Average Time  : 0.000135 seconds
  Key 1         :
```

```
SELECT FirstName, LastName, Address,
        City, State from Staff WHERE StaffID = ?
```

These results are not truncated.

It might seem strange that the first set of results suggest that we only update the database once, but that these results suggest we did something with the update command 3 times. Both are actually correct.

In our code (`exercises/answers/update.pl`) we `prepare` the update handle, then execute it and finally, behind the scenes, we destroy it. Each action potentially involves touching the database, first to see if the database wants to do anything special with the prepare statement, secondly to make our changes and finally to ensure that any resources associated with the statement are cleaned up.

As a result of this, you should always keep in mind that counts at level 2 do not reflect only the executions of statements. Preparing, fetching and destroying the statement handles will also affect the results.

Exercise

The above results were gained by turning on profiling for the `exercises/answers/update.pl` program. Turn profiling on for one of your programs (for example `insert.pl`) and run it, once at level 2 and once at level 6.

If your program requires input, make sure that you provide the same input for both path levels, to ensure that you receive consistent results.

Chapter summary

- Most errors in DBI code can be spotted by using `RaiseError` and `ShowErrorStatement`.
- Tracing allows you to see how your calls to DBI’s methods are being interpreted and what is occurring inside the drivers. Useful trace levels are between 1 and 5.
- Tracing can be turned on for all methods from a database handle, for all actions on a statement handle or globally for your whole program.
- Setting the DBI_TRACE environment variable turns on tracing for your whole program.
- Profiling allows us to determine which parts of your code are taking up the most execution time. This helps identify effective places to apply optimisations.
- Profiling Perl is often done with `Devel::DProf` and **dprofpp**.
- Standard Perl profiling tools are not sufficient to profile Perl’s interaction with database applications.
- To profile DBI code use `DBI::Profile` and its child classes.
- `DBI::Profile` takes as an argument a path which is used to determine how data is merged into the collected profile data.
- To turn profiling on we can edit our code directory or use the DBI_PROFILE environment variable.
- Profile information is interpreted with **dbiprof**.

Chapter 11. Class::DBI

In this chapter...

In this chapter we cover a Perl module called `Class::DBI` and how it can be used to reduce the amount of code and SQL you need to write.

The official documentation for the `Class::DBI` suite of modules can be found on CPAN (http://search.cpan.org/perldoc?Class::DBI). You may also wish to look at DBIx::Class (http://search.cpan.org/perldoc?DBIx::Class), which is another module with similar functionality.

The problems in mixing SQL and Perl

`DBI` is a great module. It allows you to abstract away a fair amount of the the *brand* specific parts of database manipulation. However it still means that you have to prepare, execute and process the results for each query. It still means that you have to write an awful lot of SQL.

SQL is all very well and good, but you'll find that a lot of the time you're writing the same SQL statements in different (but related) programs. You could abstract them out into subroutine calls but you'll find that you want the results as an array here, and a hash over there. Yet if you *don't* abstract out the SQL then you'll have lots of code to change when your database structure changes; when what was previously a column in table X becomes a new table, or if a column is renamed, or another column is inserted before it. Even if your database doesn't change, abstracting out your SQL reduces the amount of rework necessary make changes to your business logic.

SQL isn't Perl. Just as we should separate data from our programs we should separate SQL from our programs, yet doing so with DBI is hard.

What is Class::DBI?

`Class::DBI` is what makes separating SQL from code easy. It doesn't just abstract away the *type* of database, it abstracts away the *whole* database.

`Class::DBI` allows you to turn rows in your database into simple Perl objects for your program to interact with. Change the object and you change the row in the database. Gone is the need to write code to update, add, or delete a database entry. The only SQL left for you to write are the search terms you need. This gives you more time to spend on writing the important stuff: your application's logic.

`Class::DBI` does more than providing a simple database to object mapping layer. It also provides triggers, referential integrity and cascading deletes at the application level. This means that you can choose to use these features even if you don't know how to write PL/SQL code, or your database doesn't natively provide them, or you want some triggers to occur in some programs but not others. Furthermore, once you've made use of these functions with `Class::DBI` you can always use them, even if you change your database engine.

Setting up an application base class

The base class allows you to keep your database connection in one place. You can either hard-code your database username and password here or pull these values out from a configuration file. It also allows you to set any environment variables required, or add in any enhancements you'd like to make to `Class::DBI`.

To set up a base class we just inherit from `Class::DBI` and describe how we connect to the database. If desired, we can also perform other setup, and declare methods to be inherited by child classes.

```
package Staff::DBI;

use strict;
use base 'Class::DBI';  # Inherit from Class::DBI

# Any setup of environment variables (such as required for Oracle) goes
# here

# This uses the same parameters as connecting with DBI
__PACKAGE__->connection('DBI:mysql:database=dbiX', 'dbiX', 'password',
        {
                RaiseError         => 1,
                ShowErrorStatement => 1,
                PrintError         => 0,
                AutoCommit         => 1,  # No transactions
        }
);

# Any other setup we wish to do goes here.  This can include methods
# which wish to make available to all the children classes.

1;
```

__PACKAGE__

In many of our examples you'll see the strange string `__PACKAGE__`. This is a special constant in Perl which means *the name of the package I'm currently in*.

It is a good idea to use `__PACKAGE__` instead of writing out the package name in full to reduce code duplication. Should our package name change later, or our code be cut and pasted as the start of another project, the code will continue to work as it should.

If we don't appear to be in any package, we're in the `main` package.

Function information

connection

```
__PACKAGE__->connection($data_source, $user, $password, \%attr);
```

`Class::DBI`'s `connection` method has the same form as `DBI`'s `connect` method. However, to make life as easy for you as possible, it provides the following defaults in the attributes hash reference:

```
        FetchHashKeyName   => 'NAME_lc',
        ShowErrorStatement => 1,
        ChopBlanks         => 1,
        AutoCommit         => 1,           # EXCEPT for Oracle and Pg
```

If you are using either the *Oracle* or *Pg* drivers `AutoCommit` will be set to `0`. This is because both Oracle and PostgreSQL can always support transactions, regardless of table type or version number, whereas this is not necessarily the case with other databases.

It is highly recommended that you *always* set the `AutoCommit` parameter when connecting to the database, and do not rely upon the defaults. See the section on *Transactions* later in this chapter for more information.

```
__PACKAGE__->connection('DBI:mysql:database=dbiX', 'dbiX', 'password'
        {
                # Force AutoCommit to true
                AutoCommit => 1,
        }
);
```

Normalisation of column names

In the default attributes mentioned above one is of special note.

```
        FetchHashKeyName   => 'NAME_lc',
```

This attribute normalises column names to be all in lower case even though they may be in uppercase or mixed case in your database schema. Thus although our primary key for our `Staff` table is `StaffID` we will access it as:

```
        my $id = $staff->staffid();
```

and typically refer to the column in lower case.

The exception to this is in class definitions and actual SQL which will be covered below.

Setting up a class

`Class::DBI` works on a simple one class/one table model. As such, you'll need to create a class for each table in your database. Each class inherits from the application base class and thus doesn't need to connect to the database.

To setup the class we'll need to inherit from our base class, tell `Class::DBI` what our table is, what columns we're using, what our primary key is and what relationships it will have with other tables. None of this is difficult.

```
package Staff::Details;
use strict;
use base 'Staff::DBI';  # Inherit from our application base class

# Declare the name of our table
__PACKAGE__->table( 'Staff' );

# Declare our primary key field(s)
__PACKAGE__->columns(Primary => 'StaffID');

# Declare our columns
__PACKAGE__->columns(All => qw/StaffID FirstName LastName Address City
                               State Position Wage/);
```

```
# Declare our relationships to the other tables/classes
__PACKAGE__->has_many(
        phones => 'Staff::Phone',
);

__PACKAGE__->has_many(
        projects => 'Staff::Projects',
        { order_by => 'allocation' }
);

# Any Class specific searches etc would go here.

1;
```

In the above code we say that the `Staff::Details` class has a *many* relationship with the `Staff::Phone` class. That is, the primary key of the `Staff::Details` table will appear in multiple rows in the `Staff::Phone` table. The relationship of the `Staff::Phone` to the `Staff::Details` is then declared in the `Staff::Phone` definition.

```
package Staff::Phone;

use strict;
use base 'Staff::DBI';  # Inherit from our application base class

# Declare the name of our table
__PACKAGE__->table( 'StaffPhone' );

# Declare our primary key field(s)
__PACKAGE__->columns(Primary => 'StaffID', 'PhoneNumber');

# Declare our relationships to the other tables/classes
__PACKAGE__->has_a(
        StaffID => 'Staff::Details',
);

# Any Class specific searches etc would go here.

1;
```

Declaring a `has_a` relationship on the `Staff::Phone` class confirms the relationship with the previous table as being a *one-to-many*. We specify the foreign key as `StaffID` so that `Class::DBI` knows how to join these tables.

Function information

table

```
# To set the table information for our class
__PACKAGE__->table($table_name);

# To get the table name via our package name
$table = Class->table;

# To get the table name via an object
$table = $obj->table;
```

The `table` method *must* be set for each class. The table name can then be accessed in your application either via the class's name or by asking an object from that class.

columns

```
__PACKAGE__->columns(All => @all);
__PACKAGE__->columns(Primary => @primary);
__PACKAGE__->columns(Essential => @essential);
__PACKAGE__->columns(TEMP => @temporary);

# then later....
my @all_columns  = Class->columns;

# Give me all my primary key columns
my @primary      = Class->primary_columns;

# Or if you only have one primary key column
my $primary      = Class->primary_column;

# All columns marked as essential
my @essential    = Class->_essential;
```

The `columns` method allows you to specify the columns in a table. If you have a single-part primary key then it is possible to just call `columns` once, with the `All` tag, passing in the primary key as the first value.

If you have a multi-part primary key you must call `columns` for the full list of columns as well as for the primary key fields. As a general rule it is always a good idea to explicitly denote your primary key.

The `Essential` tag allows you to specify the fields which *must* be fetched from the database when an object is created. `Class::DBI` uses a form of lazy instantiation to only fetch the minimum amount of data necessary to create each object. Should you attempt to access data which isn't yet in the object, `Class::DBI` fetches the rest of the object data and fills it in.

What is essential may vary from application to application. If we were writing a Payroll system then we'd probably consider all of the information in the `Staff` table to be essential. Whereas if we were writing a Staff Meeting Scheduler we'd probably only care about `StaffID`, `FirstName`, `LastName` and `Position` for the same table.

If you do not specify which columns are essential `Class::DBI` will only use the primary key columns. It should be noted that `Essential` is simply a means of optimisation, and can be omitted entirely if desired.

The `TEMP` tag instructs `Class::DBI` that this is a transient column (ie. it does not exist in the database). If this column is populated during the life of the object, then marking it as a *temporary* column tells `Class::DBI` not to attempt to save it to the database during subsequent interactions.

has_many

```
package Staff::Details;

__PACKAGE__->has_many(phones => "Staff::Phone");

# And later in code
my @phone_numbers = $staff->phones();
```

The `has_many` method declares that the other table contains this table's primary key. By declaring this, `Class::DBI` creates a method (named as the first argument you pass in) which returns a list of all matching rows on that primary key. Thus in our example above, calling `phones` returns all rows which match our `$staff`'s `StaffID`.

When a class declares a relationship with another table via `has_many` the other table must also declare a relationship with it. This might be another `has_many`, a `has_a` or a `might_have` relationship.

has_a

```
package Staff::Projects;

__PACKAGE__->has_a(StaffID => ’Staff::Details’);

# And later in code ...

# Get my staff member (fetches whole object from database)
my $staff = $project->staffid;

# Print their first name
print $staff->firstname();
```

The `has_a` method declares that the given column is a foreign key (primary key of the specified class). Once we’ve done this, calling the accessor method for this column returns an object of the foreign class rather than just the column value. This allows us to link the two classes together more seamlessly.

Consult the Class::DBI documentation (http://search.cpan.org/perldoc?Class::DBI) for other relationships that can be declared.

Using our objects

Once we’ve set up classes for all of our tables (and written appropriate documentation and appropriate test cases) we can start to use our code. Below we’ve written an application which prints information out about the projects each user is involved in and the total amount of their time allocated:

```
#!/usr/bin/perl -w
# Lists each staff member and their project allocations
use strict;
use Staff::Details;

# Fetch all rows from our Staff table
# This returns a Class::DBI::Iterator object which allows us
# to iterate over objects one by one rather than loading them all into
# memory.  Using iterators often helps increase the speed of your program.

my $staff_it = Staff::Details->retrieve_all();

print "-" x 70, "\n";

while(my $staff = $staff_it->next()) {
        print $staff->firstname(), " ", $staff->lastname(),
              " is allocated to the following projects:\n";

        # Print allocation for each project
        my $total_allocation = 0;
        foreach my $project ($staff->projects()) {
                print $project->projectname(),
                      " @ ",
                      $project->allocation(),
                      "%\n";
                $total_allocation += $project->allocation();
        }
        print "for a total allocation of $total_allocation %\n";
        print "-" x 70, "\n";
}
```

It's also possible to call the `get` method on your object with an attribute name. This is particularly useful if you have a variable containing the attribute you wish to retrieve. The following lines are equivalent:

```
my $attribute = "projectname";

$name = $project->projectname();
$name = $project->get("projectname");
$name = $project->get($attribute);
```

Exercise

The `Staff::DBI` and `Staff::Details` files have already been created for you and can be found in your `exercises/Staff/` directory.

1. Edit `Staff::DBI` to access your database with your username and password.
2. Using the information above, create a `Staff::Projects` module. Remember to specify it's relationship to `Staff::Details`. Make sure it compiles. We'll use this class in the next exercise.
3. The above code listing each staff member and their project allocations can be found in `exercises/projects.pl`. Run it now to test your classes.

Search results

In `Class::DBI` there are two methods to gain access to search results. If you ask for your results in a scalar context you'll receive a `Class::DBI::Iterator` object. These objects allow you to retrieve each result from the database one at a time. This can reduce your memory overhead and increase the speed of your program with large data-sets.

If you ask for your results in an array context you'll receive all of your results back in that array. For large data-sets this can be memory intensive and may slow down your program execution.

Simple searches

The `retrieve_all` used above returns all the data in your table. In most cases you'll probably want to only retrieve a subset of your table data. In addition to `retrieve_all`, `Class::DBI` provides the following basic methods:

- `my $obj = Class->retrieve($primary_key)` - creates an object with the data matching that key,
- `my @results = Class->search(column1 => $value, column2 => $value ...);` - creates a list of objects matching the given search (conditions are `AND`ed together).

- `my @results = Class->search_like(column1 => $similar_string, ...);` - creates a list of objects using *like* instead of *equals* in the SQL where clause (conditions are `AND`ed together).

```
# Select Jack Sprat (id 12345)
my $staff = Staff::Details->retrieve(12345);

# Select all staff members in Melbourne, Victoria
my @victorian = Staff::Details->search(city => "Melbourne", state => "Vic");

# Select all staff members whose surnames start with S
my @s_staff = Staff::Details->search_like(LastName => 'S%');
```

Extending our classes

The `retrieve_all` method doesn't allow us to provide any sorting criteria, which can be annoying. Fortunately, if we don't like the `retrieve_all` method we can write new methods instead.

There are two helpers which we can use to create new methods. These are `add_constructor` and `set_sql` and are covered further below.

add_constructor

If we're happy for `Class::DBI` to select all *essential* columns from the database for each of our queries but we want to be able to edit the *where* clause, we'd use `add_constructor`.

The `add_constructor` method generates new methods which can then be called to return appropriate objects. The where clauses used can take placeholders if desired.

```
package Staff::Details;

# Selects Staff members who live in Melbourne and earn more than 80K
__PACKAGE__->add_constructor(lucky => q{
       City = 'Melbourne' AND Wage > 80
});

# Selects Staff who live in given state and have given position
__PACKAGE__->add_constructor(local_pos => q{
       Position = ?  AND State = ?
       ORDER BY LastName
});

### and in our code...

my @lucky_staff    = Staff::Details->lucky();
my @vic_developers = Staff::Details->local_pos('Devel', 'Vic');
```

set_sql (and aggregate searches)

If we need to have control over the full select command we can use `set_sql`. This generates a new method with the prefix *`search_`* added to your given name. Special values `__TABLE__`, `__ESSENTIAL__` and `__IDENTIFIER__` are provided to help simplify things.

```
package Staff::Details;

# Creates a method search_all which orders by StaffID
# This method can be called with ’search_all’
__PACKAGE__->set_sql(all => q{
        SELECT __ESSENTIAL__
        FROM __TABLE__
        ORDER BY StaffID
});

# A method to return highest earning staff member from given city
# This method can be called with ’search_money_bags’.
__PACKAGE__->set_sql(money_bags => q{
        SELECT __ESSENTIAL__
        FROM __TABLE__
        WHERE City = ?
        ORDER BY Wage DESC
        LIMIT 1
});

# Our ’search_money_bags’ method above always returns a
# single result, but Class::DBI doesn’t know that.  The
# ’money_bags’ method below wraps it and calls ’first’
# to extract the single result, saving us from having to
# do so in application code.

sub money_bags {
        my ($class,@args) = @_;
        return $class->search_money_bags(@args)->first;
}

# And later in our code:
my @all_staff_sorted = Staff::Details->search_all();

my $highest = Staff::Details->money_bags(’Melbourne’);
```

`set_sql` can also be used to perform aggregate searches so long as those searches return results which still map to rows in the database. For example we can order our staff members by number of projects they’re involved in.

```
package Staff::Details;

# Create a transient column on for each object
__PACKAGE__->columns(TEMP => qw/NumProjects/);

# Sort staff members by number of projects they’re involved in
__PACKAGE__->set_sql(by_num_projects => q{
        SELECT s.StaffID, count(ProjectName) as NumProjects
        FROM Staff s, Projects p
        WHERE p.StaffID = s.StaffID
        GROUP BY s.StaffID
        ORDER BY NumProjects DESC
});

# and later in our code:
my @staff = Staff::Details->search_by_num_projects();

while( ref (my $project = $projects_it->next )) {
        print $project->firstname, " ", $project->lastname, " is assigned to ",
              $project->numprojects(), " project(s)\n";
}
```

By default, `Class::DBI` will ignore any unrecognised fields which are selected in an SQL statement. If we wish to have access to the `NumProjects` value in our later code we need to tell `Class::DBI` at class construction that our objects may include this column at some point in the future.

This is where we use `TEMP`. A `TEMP` column is one which may (or may not) exist on any object but which has no corresponding field in the database and thus will not be saved to the database with the rest of its data.

Other aggregate searches

Many aggregate searches result in a loss of identity for the results returned and thus the results do not themselves map to any database row. Since `Class::DBI` classes are designed to each represent a database row, these kinds of aggregate searches do not work well in `Class::DBI`.

An example is if we were to perform a search which adds all the allocated time for each project. Our results would contain the `ProjectName` and the `TotalTime` but not a `StaffID`.

We can still perform this search but to do so we need to access the database directly. Searches such as these are best encapsulated into subroutines within the appropriate class.

```
package Staff::Projects;

sub total_allocations {
        my $class = shift;

        # Get a handle to the database
        my $dbh = $class->db_Main();

        return $dbh->selectall_hashref(q{
                SELECT ProjectName, sum(Allocation) as TotalTime
                FROM Projects
                GROUP BY ProjectName
        }, 'projectname');
}


# And later in our code
my $results = Staff::Projects->total_allocations();

print "Project  Time Allocated\n";
foreach my $result ( values %$results ) {
        print "$result->{projectname}          ".
              "$result->{totaltime}\n";
}
```

Adding and editing entries

We've seen how to fetch data from the database through searches and retrieves. Adding new data is as simple as creating a new object using `create`. Updating an object is as simple as changing the object and issuing the `update` command.

When we specify a `has_many` relationship on a class, `Class::DBI` automatically generates `add_to_<method name>` methods for us. Thus to add a phone number to an existing staff member we use the `add_to_phones` method and to add a project we use the `add_to_projects` method.

```
# Create a new staff member from data in %data
```

```
my %data = (
        firstname => ’Mary’,
        lastname  => ’Smith’
        position  => ’Devel’,
        wage      => 140,
);

my $new_staff = Staff::Details->create(\%data);

# add a few phone numbers for this staff member
foreach my $number ("03 4567 4567", "02 4567 1234") {
        $new_staff->add_to_phones(
                {
                        phonenumber => $number,
                }
        );
}

# allocate them to a project
$new_staff->add_to_projects(
        {
                projectname => "ABC",
                allocation  => 100,
        }
);

# New project is now added to database.

# change our staff member’s city
$new_staff->city("Alice Springs");
$new_staff->state("NT");

$new_staff->update();   # Issue the UPDATE command to the database
```

In addition to calling our attribute names as methods, we can also use the `set` method to change our objects. Like `get`, this is particularly useful when our attribute name is stored in a variable. The following lines are equivalent:

```
my $attribute = "state";

$new_staff->state(         "NSW");
$new_staff->set("state",   "NSW");
$new_staff->set($attribute,"NSW");
```

Transactions

Many databases allow you to use transactions as covered earlier in this course. `Class::DBI` provides only a thin layer over the transaction support offered by the `DBI` module. If you wish to use transactions as part of your application, you have two choices:

- Explicitly set `AutoCommit` to `0` when establishing your connection. All changes will now be considered part of a transaction, and changes must be explicitly commmited using `dbi_commit()` (as well as `update()` when appropriate). This is most useful when the vast majority of your changes will require transactions.
- Explicitly set `AutoCommit` to `1` when establishing your connection. Transactions must now be started manually, although `Class::DBI` does not provide an interface to `DBI’s begin_work`. The supported way to use transactions is to locally change the value of `AutoCommit` for the transaction block; an example of this is provided below.

```
# If AutoCommit is turned off, we’re always inside a transaction...

$new_staff->city("Alice Springs");
$new_staff->state("NT");

$new_staff->update();        # Issue the UPDATE command to the database
$new_staff->dbi_commit();    # Commit these changes to the database

$new_staff->state("NSW");
$new_staff->update();       # Issue the UPDATE command to the database
$new_staff->dbi_rollback(); # Oops, we don’t want to make that change
                            # Rollback

# If AutoCommit is turned on, we need to turn it off in order to
# start a transaction:
{
    local $new_staff->db_Main->{AutoCommit} = 0; # Locally disable AutoCommit

    $new_staff->city("Sydney");
    $new_staff->state("NSW");

    $new_staff->add_to_projects( {               # Add staff to project.
        projectname => "ABC",
        allocation  => 100,
    } );

    $new_staff->update();       # Issue the UPDATE command to the database

    # At the end of our block AutoCommit returns to 1, and our
    # transaction is commited automatically.  We could explicitly
    # rollback any time before this by calling $new_staff->dbi_rollback;
    # A call to dbi_commit() is not needed.

}
```

The `update` method does not *commit* your changes to the databases if you are using transactions. Calling `update` informs `Class::DBI` that you have finished making changes to the object for the time being and that you wish it to now send an appropriate `UPDATE` statement to the database.

If you wish the database to receive `UPDATE` commands for each and every change you make to an object you can turn `autoupdate` on for your object or class. It is important to understand that this may greatly increase the number of calls to the database and thus may result in performance problems. `autoupdate` can be turned on and off for both classes and individual objects.

```
# Turn off autoupdate for the whole class
Staff::Details->autoupdate(0);

# Turn on autoupdate for just this staff member
$staff->autoupdate(1);
```

Deletion

To delete an object, we issue the method call `delete`. We can also use `delete_all` to remove either all data in a table or all data in the table which relates to this object (thus I can delete either all phone numbers, or all phone numbers belonging to a particular staff member).

Deletions *cascade* where possible. This means that if we delete Jack Sprat (staff id: 12345) through `Class::DBI` then Jack’s allocations to his projects will be deleted as well as his telephone numbers. `Class::DBI` gives this to us even if your database cannot.

If you are using transactions, you will have to call either `dbi_commit` or `dbi_rollback` when appropriate after your call to either delete function.

```
my $jack = Staff::Details->retrieve(12345);

# Delete $jack (deletes from Projects and StaffPhone as well)
$jack->delete();

my $john = Staff::Details->retrieve(12346);

# Delete all of John’s phone numbers
$john->phones->delete_all();

# Delete all Admin staff
Staff::Details->search(position => "Admin")->delete_all();
```

When creating objects with a *has_a* relationship to another table (such as our Projects and StaffPhone tables) the foreign key (StaffID) is replaced with a *reference* to the object it is representing. For example:

```
 bless( {
          ’staffid’ => bless( {
                                ’staffid’ => ’12349’
                              }, ’Staff::Details’ ),
          ’projectname’ => ’XYZ’
        }, ’Staff::Projects’ )
```

In most cases, if you use this field as a string the value will *stringify* to the value of the foreign key and all will work as you expect. Unfortunately, some database drivers, including those for Oracle, have trouble with this structure and may return strange errors about being unable to use a reference as a primary key.

If you encounter this problem you can bypass it by explicitly setting the value of your primary key before object deletion. If you find that you need to do this in more than one place you may want to supply your own delete function to the appropriate class:

```
package Staff::Projects;

sub delete {
        my $self = shift;

        # Set my staff id to be the actual id, not a reference
        $self->staffid( $self->staffid . "" );

        # Call the next delete (probably Class::DBI’s)
        $self->SUPER::delete();
}
```

Fortunately this is rarely necessary.

And there's more!

Hopefully this chapter has given you enough to get started with `Class::DBI`. However, there's still a whole lot more to learn. Take a look at the documentation mentioned in the *readme* at the start of this chapter. Give `Class::DBI` a go, it's best learned by doing.

Exercise

Take a look at `exercises/cdbi_examples.pl` and `exercises/project_allocation.pl` to see some of the previous examples in action.

Chapter summary

- `Class::DBI` provides a well designed abstraction layer to assist in the separation of SQL and Perl.
- In many cases, `Class::DBI` reduces the amount of SQL to be written while increasing the re-usability of code.
- `Class::DBI` provides methods to search databases, create new rows, update and delete entries.
- Once table relationships are specified to `Class::DBI` cascade deletion is managed and access to relation tables is made easy.

Chapter 12. Practical exercises

In our previous practical exercise you would have found yourself writing a lot of SQL. Fortunately we were able to abstract much of it away into a module of SQL queries, but there was still a fair bit of it around. Without the scaffolding the whole project could have taken a week. Fortunately, Class::DBI allows us to simplify our database handing even more so that we can spend more time actually coding.

The problem

Rather than introduce a new problem, we'll reuse the same ideas from the first practical exercise. This will provide you with familiarity with the likely problems you'll encounter, and also give you a good basis for comparison of regular DBI programming and that using Class::DBI. For a scheme description, please see the first practical exercise.

Exercise 1 - creating your CDBI classes

For this exercise you'll need to edit the `exercises/images/lib/Image/DBI.pm` and its two child classes: `exercises/images/lib/Image/Publications.pm` and `exercises/images/lib/Image/Categories.pm`. The examples in the Class::DBI chapter should prove useful references here.

1. Fill in the `exercises/images/lib/Image/DBI.pm` base file remembering to connect to the database with `RaiseError => 1` and `ShowErrorStatement => 1`
2. Fill in the `exercises/images/lib/Image/Images.pm` specifying your table name, columns and relationships with other tables.
3. The `exercises/images/lib/Image/Publications.pm` and `exercises/images/lib/Image/Categories.pm` files have already been completed for your convenience. Look over these files and make sure that you understand how they work.

Exercise 2 - listing the images

Now that we've set up our base files, we can use them. For this exercise you'll need to edit `exercises/images/lib/Image/Images.pm` again. We'll also edit `exercises/images/lib/Image/CDBI_Utility.pm`.

1. The `exercises/images/cgi-bin/cdbi_list_images.cgi` script assumes that the `search_all` constructor is working. The basic structure of this has been added to `Images.pm`, fill in the missing SQL remembering to use the special tags `__ESSENTIAL__` and `__TABLE__`. Remember to order by `image_id`.
2. Edit `get_image_list()` in `Image::CDBI_Utility.pm` to use the `search_all` that you've just created. You should be able to just return the list you get as a result.

   (Hint: if this takes more than 2 lines of code you might be doing something wrong)

3. If all has gone well then `exercises/images/cgi-bin/cdbi_list_images.cgi` should list out all the images without any further work.

Exercise 3 - displaying an image

As with the earlier practical exercise, all the rest of our work will be done in `exercises/images/lib/Image/CDBI_Utility.pm`. Typically a lot of this would be done directly into the calling files, but we're doing it this way to make it easier.

Additional files for this exercise are: `exercises/images/cgi-bin/cdbi_show_images.pl` and `exercises/images/tmpl/show_image.html`.

1. Edit `get_image()` so that it returns an `Image::Images` object with the appropriate information.

   (Hint: if this takes more than 2 lines of code you might be doing something wrong).
2. Edit `get_categories()` so that it returns a reference to a list of `Image::Categories` objects.

   Ideally, the categories should be ordered by category name. How can we do that? (Hint: This can be done in the Image::Images class.)
3. Edit `get_publications()` so that it returns an array reference of publications. Remember that you should be able to access this from the `$image` object passed in.

   The publications should be ordered by publication date. How can we do that?
4. Check that your `cdbi_show_image.cgi` script works over a number of the images.

Exercise 4 - editing images

For this exercise we'll be adding to `exercises/images/lib/Image/CDBI_Utility.pm`. Additional files used are: `exercises/images/cgi-bin/cdbi_edit_images.pl` and `exercises/images/tmpl/edit_image.html`.

1. `Class::DBI` does not make it easy to perform aggregate searches that do not include complete primary keys. As a result we'll have to get our distinct categories the "old-style" way. Edit `distinct_categories()` to use the provided database handle (`$dbh`) to return an array reference of categories.

   Hint: check how you did this in Prac 1.
2. Edit `update_image()` to allow editing of image details. Make a few changes to make sure that this works.
3. Edit `update_categories()` to allow editing categories. Just like last time you'll probably want to delete the categories first and then add all selected.
4. Edit `add_publications()` to allow adding publications.
5. Edit some images and test your functionality.

Advanced exercise

Implement transactions to ensure all edits are made to an image, or no edits are. Where will you need to do this?

Chapter 13. Conclusion

What you've learnt

Now you've completed Perl Training Australia's Database Programming with Perl module, you should be confident in your knowledge of the following fields:

- What is a database? Why they're used. What a relational database is.
- What is SQL? How to `SELECT` statements from a database, how to `INSERT` new entries and `UPDATE` and `DELETE` existing entries.
- Using DBI with Perl, how to connect to the database, pull out data, and execute other queries.
- How to avoid including passwords in your database programs
- What a transaction is and how to use them through DBI.
- How taint checking can be used with DBI to ensure that tainted data either from the user or database isn't passed to the shell.
- How to bind variables for data going into the database and for data coming out from the database.

Where to now?

To further extend your knowledge of Perl, you may like to:

- Work through any material not included during the course
- Visit the websites in our "Further Reading" section (below)
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme"
- Install Perl and MySQL on your home or work computer
- Practice using Perl with databases from day to day
- Join a Perl user group such as Perl Mongers (http://www.pm.org/)
- Join an on-line Perl community such as PerlMonks (http://www.perlmonks.org/)
- Extend your knowledge with further Perl Training Australia courses such as:
  - CGI Programming with Perl
  - Perl Security
  - Object Oriented Perl

  Information about these courses can be found on Perl Training Australia's website (http://www.perltraining.com.au/).

Further reading

Books

- Alligator Descartes and Time Bunce, *Programming the Perl DBI*, O'Reilly and Associates, 2000. ISBN 1-56592-699-4
- Damian Conway, *Object Oriented Perl*, Manning, 2000. ISBN 1-884777-79-1
- Tom Christiansen and Nathan Torkington, *The Perl Cookbook*, O'Reilly and Associates, 1998. ISBN 1-56592-243-3.
- Joseph N. Hall and Randal L. Schwartz *Effective Perl Programming*, Addison-Wesley, 1997. ISBN 0-20141-975-0.

Online

- The Perl homepage (http://www.perl.com/)
- The Perl Journal (http://www.tpj.com/)
- Perlmonth (http://www.perlmonth.com/) (online journal)
- Perl Mongers Perl user groups (http://www.pm.org/)
- PerlMonks online community (http://www.perlmonks.org/)
- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup
- Comprehensive Perl Archive Network (http://www.cpan.org)

Appendix A. Introduction to databases

In this chapter...

Most people prefer things to be catalogued. In most houses, shoes go in the cupboard or under the bed, not in the fridge. Washing powder is kept next to the washing machine and food is stored in the kitchen. We consider things to be "lost" when they're not where we expect them to be.

A database is a way of storing information or records in a structured fashion, in order to make the data easy to manipulate and use.

In this chapter we'll explore what a database is and how databases help us extend our cataloguing to computer applications.

What is a database?

A database is an organised body of related information. Some simple examples of database are:

- telephone books
- library catalogs
- personal address books
- encyclopedias

The key feature of databases is that they are *organised*. This means that there is some manner in which to easily search for and find entries. Items may be organised into:

- a table of contents
- alphabetical order
- numerical order
- subject categories

What is an electronic database?

An electronic database is an organised body of information that is designed for rapid search by a computer. It may be stored on CD-ROM, DVD, hard disk drives, tape, floppy disks or other computer media.

In this course we're going to deal exclusively with electronic databases. The physical location of that database will be largely irrelevant.

Why use a database?

Many computer applications use databases in the background. This allows the application to do all the input, processing and display and to store information persistently (after the program has exited) in an easily retrievable form. Databases are also useful when more than one application needs to share the same information.

Types of databases

Typically, when we think of a database we think of a "database engine" such as Oracle, MySQL or PostgreSQL. However there are several categories of databases:

- Flat files
- Berkeley DB Files (DBM)
- Relational database
- Object-oriented database
- Hierarchical database

An example of a hierarchical database is LDAP (light-weight data access protocol). We won't cover hierarchical and object oriented databases further in this course.

Flat files

Single line records

The simplest form of flat file database holds one record per line. This mimics a simple hash of key, value pairs. Configuration files are a good example of this kind of database. For example:

```
company: Perl Training Australia Pty Ltd
email:   contact@perltraining.com.au
phone:   03 9354 6001
```

This database uses a colon character as a delimiter to separate each value from its keyword. Depending on our program we may also allow comments (lines starting with # for example) and list values.

Multiple line records

Often we wish to have multiple values per record and still have the simplicity of one value per line. This is especially useful if the file is likely to be edited manually. This form mimics a hash of hashes. This form is used in many applications including Windows .INI files. Here we use it to simulate a card-index:

```
; Three good Perl resources
[Learning Perl]
edition=   2nd
authors=   Randal L Schwartz
isbn=      1-565-92284-0
publisher= O'Reilly

[Programming Perl]
edition=   3rd
authors=   Larry Wall, Tom Christiansen, Jon Orwant
isbn=      0-596-00027-8
publisher= O'Reilly

[Object Oriented Perl]
edition=   1st
authors=   Damian Conway
isbn=      1-884-77779-1
publisher= Manning
```

In the above example, titles (and therefore new records) are denoted as new sections (enclosed in square-brackets). We also allow multiple authors by separating them with commas. We could turn the above into a hash of hashes using the `Config::IniHash` module from CPAN.

```
#!/usr/bin/perl -w
use strict;
use Data::Dumper;
use Config::IniHash ’ReadINI’;

my $filename = "books.txt";
my $books = ReadINI($filename) or die "Could not process config file - $!\n";

# Authors are separated by commas, so split on commas and add them in.
foreach my $book (keys %$books) {
    $books->{$book}{authors} = [ split(/\s*,\s*/,$books->{$book}{authors}) ];
}

print Dumper($books);
```

This prints:

```
$VAR1 = {
          ’learning perl’ => {
                               ’publisher’ => ’O\’Reilly’,
                               ’isbn’ => ’1-565-92284-0’,
                               ’authors’ => [
                                              ’Randal L Schwartz’
                                            ],
                               ’edition’ => ’2nd’
                             },
          ’programming perl’ => {
                                  ’publisher’ => ’O\’Reilly’,
                                  ’isbn’ => ’0-596-00027-8’,
                                  ’authors’ => [
                                                 ’Larry Wall’,
                                                 ’Tom Christiansen’,
                                                 ’Jon Orwant’
                                               ],
                                  ’edition’ => ’3rd’
                                },
          ’object oriented perl’ => {
                                      ’publisher’ => ’Manning’,
                                      ’isbn’ => ’1-884-77779-1’,
                                      ’authors’ => [
                                                     ’Damian Conway’
                                                   ],
                                      ’edition’ => ’1st’
                                    }
        };
```

Delimited files (CSV, TSV)

Multiple line records take up a lot of space and can contain redundant information (such as the key names where this information can be provided by position). As a result many applications use a more compact representation where data is separated by a known character, for example commas (CSV - comma separated values) or tabs (TSV - tab separated values).

When using a character such as a comma for a delimiter it is important to ensure that any data containing that character is escaped in some way, usually by included it within a quoted string. A CSV file may look like the below:

```
Introduction to Perl,"Kirrily Robert, Paul Fenwick, Jacinta Richardson",2004
Intermediate Perl,"Kirrily Robert, Paul Fenwick, Jacinta Richardson",2004
Database Programming with Perl,Jacinta Richardson,2004
Object Oriented Perl,"Paul Fenwick, Jacinta Richardson",2004
```

Writing regular expressions to detail with text-separated values can be more difficult than it first appears, however Perl has a number of excellent modules for dealing with text-separated values. One of the best of these is `Text::CSV_XS`, which handles both reading and writing of files, and can be configured to interpret a wide range of quoting and escaping mechanisms.

Perl storage

Single line records, multiple line records and delimited files all have the advantage of being human readable. It's easy to examine and change their contents. However there may be occasions where speed and compactness are more important than human-readability. If we're only expecting Perl applications to work with our data, then we can create simple databases by using Perl's `Cache::Cache` and `Storable` modules. Strictly speaking these aren't really *flat file* databases, as you can dump very complex data structures into them. They can also use a variety of storage mechanisms, including both files and relational databases.

`Cache::Cache` allows data to be stored in a persistant manner for a specified period of time, or until a certain amount of space or other resource constraint is met. `Storable` allows Perl data structures to be saved to a file in an easily retrievable format on a more permanent basis.

Limitations with flat file databases

Flat file databases are simple to code and use. They don't require installation of extra software and can be adapted over time as your programs develop. Many programs use flat file databases for configuration data and small data storage. Using modules such as `Cache::Cache` and `Storable`, it is also possible to store native Perl objects and other data structures. These represent an ideal solution for many problems where persistent data is restricted to a single application.

However flat file databases quickly become unmanageable once you start dealing with large amounts of information. Effective use of flat file databases usually requires loading the database fully into memory for searching, ordering and changing. It's important to regularly write out any changes to the database to avoid loss in the case where your program terminates unexpectedly, but all file locking and recovery is left to your program.

Furthermore, flat file databases limit how well programs can share data with each other. While one program has a changed copy of the database in memory it is important that other programs avoid making changes themselves. This can slow down program execution while each program waits for its turn to modify possibly disparate portions of data.

It is due to these reasons that independent database engines have been developed.

Berkeley DB Files

DBM files, or Database Manager files, are a simple database format. Each item in a DBM file consists of a key and a value, similar to a Perl hash. If you know the key you can access the value very quickly.

There are several DBM packages available including ODBM, NDBM, SDBM, GDBM and BSD-DB. Most Unixes will have at least one of these already installed. Perl comes with a complete SDBM package bundled with it, but you can use the other DB file formats if they are available on your system.

DBM databases are suitable for a wide range of applications. They are well suited to simple record based databases where each key uniquely identifies each record. There are individual Perl modules to access each of the above mentioned DBM, there is also a Perl module called AnyDBM_File which Perl can use to automatically find a suitable DBM implementation on your system.

Using a DBM is like using a hash in Perl. Changes made to the hash are made to the database. Reading data from the hash retrieves the value from the database.

```
#!/usr/bin/perl -w
use strict;
use POSIX;
use AnyDBM_File;

my %dbm;
my $db_file = "some_database";

tie %dbm, "AnyDBM_File", $db_file, O_RDWR|O_CREAT, 0666
        or die "Can’t open $db_file: $!";

$dbm{last_read} = localtime;    # Change the last read time

foreach my $data (sort keys %dbm) {
        print "$data is $dbm{$data}\n";
}
```

DBM limitations

Many DBM databases require that their values be scalars. Assigning Perl references to these keys will result in their stringified values being assigned to the database (eg `ARRAY(0x80f57ac)`) rather than the data pointed at via the reference.

To assign more complex values use Perl’s `Storable` module to `freeze` your structure before storing it. This is often called *serialising* your data. For example:

```
use Storable qw(freeze thaw);

my @friends = qw/Simon Jane Jack Paul Fred/;
$dbm{friends} = freeze(@friends);

# and later:

print thaw($dbm{friends});      # Prints out our list of friends
```

The multi-level DBM (MLDBM)

The `MLDBM` module performs serialisation of complex data structures for us. We can specify which DBM and which serialisation module we wish to use in the import list, otherwise it will pick `SDBM` and `Data::Dumper`, both of which come bundled as part of the core Perl distribution.

```
#!/usr/bin/perl -w
use strict;
use POSIX;
```

```
# Tell MLDBM to use AnyDBM_File and Storable
use MLDBM qw(AnyDBM_File Storable);

my %dbm;
my $db_file = "some_database";

tie %dbm, "MLDBM", $db_file, O_RDWR|O_CREAT, 0666
        or die "Failed to open $db_file: $!";

my @friends = qw/Simon Jane Jack Paul Fred/;
my %pets = (
                rabbit  => "Shadow",
                chicken => "Dig-dug",
                dog     => "Ming",
);

# Copy list of friends into database.
$dbm{friends} = \@friends;

# Store the NUMBER of friends in the database.
$dbm{num_of_friends} = @friends;

# Copy pets into database
$dbm{pets} = \%pets;

# Change list of friends in database:
my @friends_update = @{$dbm{friends}};
push @friends_update, "Bella";
$dbm{friends} = \@friends_update;

# Change list of pets in database:
my $pets_update = $dbm{pets};
$pets_update->{dog} = "Rusty";
$dbm{pets} = $pets_update;
```

`MLDBM` allows us to store any kind of structure that can be created in Perl. These data structures can be arbitarily deep, and can contain self-referrential elements.

`MLDBM` overcomes many of the issues that flat file databases and DBM databases suffer from. Furthermore as `MLDBM` builds upon DBMs which are often installed on Unix machines it provides a portable way to use databases in your code without having to install a full database system.

It’s important to remember that when we assign a reference to a Perl data structure to a MLDBM, we’re *making a copy* of that structure. Changing our `%pets` hash after the assignment would not have changed it in the database. Instead, if we wish to change a data structure within the database it is best to pull it out, make the change and then reassign it into the database.

Keep in mind that context matters. If you assign an array, rather than an array reference into your database you’ll get the number of elements in the array rather than a serialised version of your array. Likewise if you assign a hash into your database you’ll get the number of hash buckets used, which isn’t very useful.

`MLDBM` itself does not come with any locking ability. This means that if you may have more than one program attempting to use the database at the same time, then you are responsible for making sure that you lock and unlock the database files yourself.

`Tie::MLDBM` modules provides locking framework to `MLDBM`. You can read more about this in its CPAN documentation (http://search.cpan.org/perldoc?Tie::MLDBM).

Relational databases

Relational database servers abstract away all of the data storage and (most of the) data integrity issues. Programmers are not required to understand where the database files are stored or how they’re implemented. Relational databases provide locking across entries or tables so that many applications can access data from the same place at the same time without having to be aware of each other.

Relational databases allow data to be inserted into tables (also called relations). Each table is made up of records and fields and is identified by a unique name. Tables can be related to each other by sharing columns (fields) with matching values.

An example table might be as follows:

```
First Name      | Surname       | Student ID
---------------------------------------------
Jacinta         | Richardson    | 001
Paul            | Fenwick       | 002
Kirrily         | Robert        | 003
Damian          | Conway        | 004
Larry           | Wall          | 042
```

This allows us to ask questions like the following:

1. Who are our students? (Jacinta, Paul, Kirrily, Damian and Larry)
2. What are the first names of students whose surnames start with ’R’ (Jacinta and Kirrily)
3. Do we have a student whose first name is ’Henry? (No)

If we assume that Student ID is unique then we know that we can refer to each student by their ID rather than by their first and last names. This means that if we eventually have two students with the same name we won’t have to change anything. Adding a further table:

```
Student ID      | Course                         | Score
---------------------------------------------------------
001             | Introduction to Perl           | 97
001             | Intermediate Perl              | 99
001             | Database Programming with Perl  | 91
002             | Object Oriented Perl           | 100
002             | Introduction to Perl           | 94
002             | Database Programming with Perl  | 97
004             | Advanced Object Oriented Perl   | 100
```

allows us to map multiple courses and their corresponding scores to each student. Notice that we’re not required to have every student appear in this table, perhaps some students haven’t done these courses. Combining this table with the previous table we can ask the following further questions:

1. Which students studied Introduction to Perl? (Paul and Jacinta)
2. Which student got the lowest score of all courses? (Jacinta)
3. Which students do not appear in the course table? (Larry and Kirrily)

This access methodology makes the relational model a lot different from, and a lot better than the early database models. It is a much simpler model to understand and as such is probably the main reason for the popularity of relational database systems today. Another benefit of a relational database is that it describes data independently of its physical representation.

For a more in depth coverage of relational databases read pages 53 - 56 in the Cheetah book.

Why use Perl to talk to databases?

Many databases provide forms to allow data entry and call stored procedures to process the data. In these situations large collections of procedures can build up over time each solving its own small problem. In such a system where could Perl possibly come in use?

Perl is regularly used with databases when some sort of processing is needed between the application and the database. For example many websites use Perl’s CGI and database capabilities to provide dynamic content. Perl is also useful in automated scripts which perform certain changes on the database; for example a script which encrypts all unencrypted passwords and sends email to new users.

Stored procedures, if written carefully, often can stand separately from the form from which they are usually called. When this is so, Perl can be used to perform pre or post processing on the data in conjunction with the stored procedures where necessary.

Perl vs SQL

A lot of the processing that can be done by the database can also be done purely in Perl. For example an SQL join can be simulated in Perl by reading out all the data of the relevant database tables and matching the data on the fields we are interested in. Select filters can be simulated by reading out all the data and throwing away the data which doesn’t match our criteria.

Just because this processing *can* be done in Perl doesn’t mean that it should be. Most databases are highly optimised for speed and memory efficiency and can join tables and filter out rows much more efficiently than can be done in Perl. As a general rule of thumb, it’s better to let the database do as much of the work as possible.

Chapter summary

- Databases are a natural extension of our desire to categorise things for later access.

- An electronic database is an organised body of information that is designed for rapid search by a computer.
- We use databases with computers when we want data persistence and/or when we want data to be shared with other programs.
- There are several types of databases including flat files, Berkeley DB Files, relational, object-oriented and hierarchical databases.
- Many programs use flat file databases to record configuration information.
- Flat file databases become unmanageable once they get large.
- Berkeley DB Files provide a simple independent database applications with some limitations.
- Relational databases remove all the database maintenance requirements away from the programmer and provide both data persistence and safe data sharing between programs.
- We use Perl to talk to databases when we need to do more than merely insert, update, select or delete the data.

Appendix B. Introduction to SQL

In this chapter...

In this chapter we’ll discuss how to use SQL (structured query language) to create tables and to insert, delete, update and select data thereafter. Each database system has its own dialect of SQL but the basic syntax we’ll be discussing here remains essentially the same. Make sure you check your database’s documentation for any areas you are unsure about.

SQL statements

SQL is a *declarative* language rather than a *procedural* language (like Perl). This means that we have to *declare* to the computer what we want it to do rather than tell it how to do it.

This might be best described by an analogy. Consider a shopping list. An imperative program would contain instructions such as go to shelf `x` in aisle `y` and get `z` many items. The equivalent declarative statement would say give me `z` many loaves of bread. *How* the bread is to be found is passed off to the database server to handle.

These declarative statements in SQL are made as English-like as possible. In order to perform any action on the database we must execute a statement. These statements start with a keyword (verb) which tells the database what we want. These are covered in the rest of this chapter.

To find out what complexity of SQL statements your database allows read your database documentation.

The SELECT statement

The `SELECT` statement is used to fetch data from the database. Despite being the most used SQL construct, it is also one of the most complex. We will not cover all of its features in this course. For further information about what you can do with `SELECT`s read your database documentation.

```
# A (reduced) general case
SELECT
    [column [as alias], column [as alias], ...]
    [FROM table_references]
    [WHERE where_definition]
    [ORDER BY {col_name | expr | position}
    [ASC | DESC] ]
```

Let’s try this with a few examples. To get all of the staff members and their addresses from our database we’d write the following:

```
SELECT FirstName, LastName, Address, City, State FROM Staff;
```

This would return the following results:

```
+-----------+----------+--------------+----------+-------+
| FirstName | LastName | Address      | City     | State |
+-----------+----------+--------------+----------+-------+
| Jack      | Sprat    | 2 Pine Rd    | Melbourne | Vic   |
| John      | Doe      | 43 James Crt  | Melbourne | Vic   |
| Betty     | Jones    | 22 Fishing St | Melton    | Vic   |
| Sam       | Smith    | 55 Queens Ave | Sydney    | NSW   |
| Ann       | Smith    | 10 Albert St  | Brisbane  | QLD   |
| Peter     | Pope     | 254 King St   | Sydney    | NSW   |
+-----------+----------+--------------+----------+-------+
```

What we've done here is we've asked for the data from each of the columns `FirstName, LastName, Address, City, State` for all the rows in the Staff table.

We can get all the columns without typing all the column names by using a `*` (star), for example:

```
SELECT * FROM Staff;
```

Using the `SELECT *` usage is strongly discouraged when programming. In the majority of programming tasks, only a specific set of columns are required. Even if the columns you require covers the entire table from which you're selecting, you're still encouraged to name them explicitly. This means that should your database schema change, your code won't select more data than it needs, or will immediately give you an error if a required column is missing.

A notable exception to this rule is when you're writing a program that is backing up a database, or otherwise obtaining data from tables without actually caring about the meaning of the data itself.

Conditional selections

In the above section we saw how to select all the staff members out of a database table. What if we didn't want all of the staff members? What if we only want the staff members whose `LastName` was `Smith` or who live in `Melbourne`? How about staff members whose `LastName` was `Smith` and who also live in `Sydney`? This brings us into conditional selections. To specify conditions in our `SELECT` statements we have to know the relational operators.

There are seven relational operators in SQL:

Table B-1. Conditional Relational Operators

Operator	Meaning
=	Equals
<> or != (see your manual)	Not equal
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
LIKE	Matches a given pattern

We can restrict our data selections by using the `WHERE` clause. For example:

```
SELECT FirstName, LastName FROM Staff WHERE LastName = 'Smith';
```

which returns:

```
+-----------+----------+
| FirstName | LastName |
+-----------+----------+
| Sam       | Smith    |
| Ann       | Smith    |
+-----------+----------+
```

Compound Conditionals (AND, OR and NOT)

We can also string conditionals together with AND and OR. The AND operator displays a row only if that row's data satisfies the left and right conditions. The OR operator displays a row if either (or both) the left and right conditions are true. For example:

```
SELECT FirstName, LastName
FROM Staff
WHERE LastName = ’Smith’ AND
      City = ’Sydney’;
```

This will return the row of Sam Smith because her last name is Smith and she lives in Sydney. No other rows match this condition. We can use parentheses to group conditionals to build up more complex structures:

```
SELECT FirstName, LastName, Address, City, State
FROM Staff
WHERE (LastName = ’Smith’ AND City = ’Sydney’) OR
      (City = ’Melbourne’);
```

This returns the following rows:

```
+-----------+----------+---------------+-----------+-------+
| FirstName | LastName | Address       | City      | State |
+-----------+----------+---------------+-----------+-------+
| Jack      | Sprat    | 2 Pine Rd     | Melbourne | Vic   |
| John      | Doe      | 43 James Crt  | Melbourne | Vic   |
| Sam       | Smith    | 55 Queens Ave | Sydney    | NSW   |
+-----------+----------+---------------+-----------+-------+
```

Finally we can also use NOT to invert a search. For example:

```
SELECT StaffID, Position, Wage
FROM Staff
WHERE NOT (Position = ’Devel’ OR Wage < 140);
```

Which gives us:

```
+---------+---------------+------+
| StaffID | StaffPosition | Wage |
+---------+---------------+------+
|   12351 | Manager       |  180 |
|   12347 | Manager       |  180 |
+---------+---------------+------+
```

The following table lists the precedence of SQL's logical operators.

Table B-2. Logical Operator Precedence

Evaluated	Logical Operator
First	NOT
Second	AND
Third	OR

IN and BETWEEN

Sometimes it’s easier to construct a list of things that something might be in rather than string together a list of OR conditions. For example the following two statements produce the same results:

```
SELECT FirstName, LastName, City
FROM Staff
WHERE StaffID = ’12345’ OR
      StaffID = ’12347’ OR
      StaffID = ’12349’ OR
      StaffID = ’12351’;

SELECT FirstName, LastName, City
FROM Staff
WHERE StaffID in (’12345’, ’12347’, ’12349’, ’12351’);

# Producing:
+-----------+----------+-----------+
| FirstName | LastName | City      |
+-----------+----------+-----------+
| Jack      | Sprat    | Melbourne |
| Betty     | Jones    | Melton    |
| Sam       | Smith    | Sydney    |
| Peter     | Pope     | Sydney    |
+-----------+----------+-----------+
```

Likewise we can get the StaffIDs of staff who are making between 79000 and 169000 with either of the following two statements:

```
SELECT StaffID FROM Staff WHERE Wage > 79 AND Wage < 169;

SELECT StaffID FROM Staff WHERE Wage BETWEEN 79 AND 169;

# Producing:
+---------+
| StaffID |
+---------+
|   12345 |
|   12346 |
|   12349 |
+---------+
```

Fuzzy Comparisons (using LIKE)

Sometimes we want to get all the information for a less well defined set of people, such as everyone whose first name starts with J or whose last name starts with S. Perhaps we want all cities our employees live in which start with M. We can do these searching using LIKE.

```
SELECT StaffID, FirstName, LastName, City FROM Staff WHERE City Like ’M%’;

# which gives us:
```

```
+---------+-----------+----------+-----------+
| StaffID | FirstName | LastName | City      |
+---------+-----------+----------+-----------+
|   12345 | Jack      | Sprat    | Melbourne |
|   12346 | John      | Doe      | Melbourne |
|   12347 | Betty     | Jones    | Melton    |
+---------+-----------+----------+-----------+
```

The `%` (percent sign) is used to represent the rest of the string. In Perl this equates to `.*` in regular expressions although `%` will match any possible character including newlines.

Joins

When we introduced relational databases, we discussed how relational databases allow us to *join* tables together to determine who had participated in which course. There are many types of joins but we'll only cover *inner joins* in this section.

In our database, we have a Staff table and a Projects table. To join these tables to determine who is working on which project we write the following:

```
SELECT FirstName, LastName, ProjectName, Allocation
FROM Staff, Projects
WHERE Staff.StaffID = Projects.StaffID;
```

This would give us the following results:

```
+-----------+----------+-------------+------------+
| FirstName | LastName | ProjectName | Allocation |
+-----------+----------+-------------+------------+
| Jack      | Sprat    | ABC         |         50 |
| John      | Doe      | ABC         |         45 |
| Betty     | Jones    | ABC         |        100 |
| Peter     | Pope     | ABC         |         70 |
| John      | Doe      | NMO         |         60 |
| Jack      | Sprat    | XYZ         |         50 |
| Sam       | Smith    | XYZ         |         50 |
| Ann       | Smith    | XYZ         |        100 |
| Peter     | Pope     | XYZ         |         30 |
+-----------+----------+-------------+------------+
```

Note that we had to specify which `StaffID` we were comparing with which. SQL requires that whenever we do a join that results in duplicate column names, we must explicitly name them in our statements. For example, had we wished to include the `StaffID` in our selection we would have had to write one of the two (equivalent) statements:

```
SELECT Staff.StaffID, FirstName, LastName, ProjectName, Allocation
FROM Staff, Projects
WHERE Staff.StaffID = Projects.StaffID;

SELECT Projects.StaffID, FirstName, LastName, ProjectName, Allocation
FROM Staff, Projects
WHERE Staff.StaffID = Projects.StaffID;

# Producing:
+---------+-----------+----------+-------------+------------+
| StaffID | FirstName | LastName | ProjectName | Allocation |
+---------+-----------+----------+-------------+------------+
|   12345 | Jack      | Sprat    | ABC         |         50 |
|   12346 | John      | Doe      | ABC         |         45 |
|   12347 | Betty     | Jones    | ABC         |        100 |
|   12351 | Peter     | Pope     | ABC        |         70 |
|   12346 | John      | Doe      | NMO        |         60 |
|   12345 | Jack      | Sprat    | XYZ        |         50 |
|   12349 | Sam       | Smith    | XYZ        |         50 |
|   12350 | Ann       | Smith    | XYZ        |        100 |
|   12351 | Peter     | Pope     | XYZ        |         30 |
+---------+-----------+----------+------------+------------+
```

Writing the full name of the table for each non-unique column name can make our SQL quite long and harder to read (especially when you wish to do a few comparisons), so many databases offer an alias feature, whereby column names can be given shorter aliases. This shortcut allows you to neaten your SQL and specify (for clarity) where you expect each column to come from even when they are unique.

```
SELECT s.StaffID, FirstName, LastName, ProjectName
FROM   Staff s, Projects p
WHERE  s.StaffID = p.StaffID AND
       Allocation = 100;

# This gives us:
+---------+-----------+----------+-------------+
| StaffID | FirstName | LastName | ProjectName |
+---------+-----------+----------+-------------+
|   12347 | Betty     | Jones    | ABC         |
|   12350 | Ann       | Smith    | XYZ         |
+---------+-----------+----------+-------------+
```

Some databases offer extended SELECT statements that allow joins to be automatically performed on like-named fields across tables. These can simplify the writing of SELECT statements considerably.

Primary and Foreign keys

Until now we've assumed that `StaffID`s are unique per staff member and that the `StaffID`s in our other tables match the `StaffID`s in the `Staff` table. This behaviour suggests a primary and foreign key relationship. By this we mean that in the `Staff` table the `StaffID` column uniquely identifies each record, so that even if we have two staff members with the same first and last names we know that they are different people. `StaffID` is the *primary key* of the `Staff` table.

As we *reference* the primary key of the `Staff` table in our `StaffPhone` tables we can say that the `StaffPhone` `StaffID` column contains a *foreign key* from the `Staff` table. The `StaffID` column does not uniquely identify the records in the `StaffPhone` table as it is possible for staff members to have zero, one or more phone numbers listed here.

The *primary key* of the `StaffPhone` table is created by referencing both the `StaffID` and the `PhoneNumber` column. While each `PhoneNumber` is currently distinct it may not be safe to assume that to always be the case.

Specifying primary and foreign keys when constructing database tables allows the database to ensure that no duplicate records are created and that *referential integrity* is maintained. Referential integrity means that if we delete a staff member from the database we should also delete their entries in each other table which uses the `StaffID` as a foreign key. Likewise, we could not insert a record into `StaffPhone` unless the person already existed in `Staff`.

Some databases do this better than others, and some don't support a concept of foreign keys at all.

Ordering data

Just like Perl hashes, databases guarantee no internal order. So while you may get data out in approximately the same order that you inserted it, this may not always be the case. Very commonly, we'll want to fetch our records in a particular order, not just the order the database would normally return them in. As a result, SQL allows you to use a `ORDER BY` instruction to specify ordering. For example consider the output of the following command:

```
SELECT FirstName, LastName, PhoneNumber
FROM Staff a, StaffPhone p
WHERE a.StaffID = p.StaffID AND
      PhoneNumber LIKE '03%';
```

This returns:

```
+-----------+----------+--------------+
| FirstName | LastName | PhoneNumber  |
+-----------+----------+--------------+
| Jack      | Sprat    | 03 9333 3333 |
| John      | Doe      | 03 8444 4444 |
| Betty     | Jones    | 03 5555 5555 |
+-----------+----------+--------------+
```

This isn't a very useful ordering for our information. Let's change it to sort by `LastName`:

```
SELECT FirstName, LastName, PhoneNumber
FROM Staff a, StaffPhone p
WHERE a.StaffID = p.StaffID AND
      PhoneNumber LIKE '03%'
ORDER BY LastName;

# This gives us:
+-----------+----------+--------------+
| FirstName | LastName | PhoneNumber  |
+-----------+----------+--------------+
| John      | Doe      | 03 8444 4444 |
| Betty     | Jones    | 03 5555 5555 |
| Jack      | Sprat    | 03 9333 3333 |
+-----------+----------+--------------+
```

Reverse sorting can be achieved by specifying `DESC` (descending) after the column name you plan to sort on. `ASC` (ascending) is the default and will be assumed.

Sorting by multiple columns is also possible, just specify the columns in the order you wish them to sort by. For example `LastName, FirstName, City`.

The `CREATE` statement

Now we know how to pull data out of database, let's talk about how we created those tables in the first place. There are many different column types, and some databases offer a richer set than others. Basic types that most databases provide are as follows:

Table B-3. Some database types

Type	Description
CHAR(*length*)	A string of characters no longer than *length*. If the string is shorter than *length* it is padded out with whitespace.
VARCHAR(*length*)	A string of characters no longer than *length*. If the string is shorter than *length* it is stored at that length.
INT	A *normal sized* integer.
DATE	A date (database specific)
BOOL	Occasionally represented by a TINYINT or otherwise, this value should provide the options for true/false flags in records.

The general case for the CREATE case is as follows:

```
CREATE TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)]
    [table_options] [select_statement]
```

There are many table creation options and it is best to refer to your database documentation. In our database, we can create the tables we've used above by the following:

```
CREATE TABLE Staff(
        StaffID         INT             AUTO_INCREMENT PRIMARY KEY,
        FirstName       VARCHAR(15)     NOT NULL,
        LastName        VARCHAR(15)     NOT NULL,
        Address         VARCHAR(30),
        City            VARCHAR(15),
        State           VARCHAR(3),
        Position        VARCHAR(20)     NOT NULL,
        Wage            INT             NOT NULL
);

CREATE TABLE Projects (
        StaffID         INT         NOT NULL REFERENCES Staff(StaffID),
        ProjectName     VARCHAR(20) NOT NULL,
        Allocation      INT         NOT NULL
);

CREATE TABLE StaffPhone (
        StaffID         INT     NOT NULL REFERENCES Staff(StaffID),
        PhoneNumber     VARCHAR(15) NOT NULL
);
```

You'll notice that we've taken a few shortcuts here. In order to ensure that each StaffID is unique we get the database to assign the StaffID at record creation time and then increment its value read for the next staff member. We've informed Projects and StaffPhone that their StaffID references Staff's StaffID (and is therefore a foreign key. Different databases do this differently so this may not work on your database.

By specifying certain fields as NOT NULL we ask the database to refuse changes which set those fields as NULL and any record addition that does not specify non-NULL values for those fields. We do this because it doesn't make sense to create a staff member who doesn't have a name, or to add a record in the Projects table which doesn't include a ProjectName.

The INSERT statement

Now that we know how to CREATE a database table and SELECT items from it, we need to know how to INSERT data into it. The general case for an INSERT statement is as follows:

```
INSERT INTO tbl_name [(col_name,...)]
       VALUES (expression,...)
```

For example, to insert a staff member into Staff we can do the following:

```
INSERT INTO Staff (FirstName, LastName, Address, City, State, Position, Wage)
VALUES ('Bob', 'Jane', '123 High St', 'Mallacouda', 'Vic', 'Devel', 90);
```

We refrain from inserting our StaffID as we know that the database will fill this in for us. Alternately we could provide the value 0 (zero).

When we add values into a table in the same order as the columns are defined without skipping any columns (including StaffID) we can leave off the column names in the INSERT statement. However, this can introduce bugs should the ordering or number of fields in our tables change, so it's considered good practice to always explicitly name our fields.

Selecting our data out after this INSERT gives us:

```
+---------+-----------+----------+--------------+-----------+-------+
| StaffID | FirstName | LastName | Address      | City      | State |
+---------+-----------+----------+--------------+-----------+-------+
|   12345 | Jack      | Sprat    | 2 Pine Rd    | Melbourne | Vic   |
|   12346 | John      | Doe      | 43 James Crt | Melbourne | Vic   |
|   12347 | Betty     | Jones    | 22 Fishing St | Melton    | Vic   |
|   12352 | Bob       | Jane     | 123 High St  | Mallacouda | Vic   |
|   12349 | Sam       | Smith    | 55 Queens Ave | Sydney    | NSW   |
|   12350 | Ann       | Smith    | 10 Albert St | Brisbane  | QLD   |
|   12351 | Peter     | Pope     | 254 King St  | Sydney    | NSW   |
+---------+-----------+----------+--------------+-----------+-------+
```

This is a good example of how databases don't promise an internal order. Our new entry is put where the hole from StaffID 12348 was.

The LAST_INSERT_ID() function

When we insert data into our Staff table the database automatically fills in the StaffID for us. This is great when we wish to merely add to that table, but not so good when we want to use the newly created StaffID to add further data to other tables.

To get around this, some databases offer a LAST_INSERT_ID() function which returns the value of the last AUTO INCREMENT. For example:

```
INSERT INTO StaffPhone (StaffID, PhoneNumber)
       VALUES (LAST_INSERT_ID(), '09 1234 1234');
```

This will add the given phone number to the same StaffID as assigned to Bob Jane by the previous INSERT.

The UPDATE statement

Once data is in the database you can do a few things with it. You can pull it out for display or calculation using a SELECT statement, you can DELETE it (which we'll cover next) or you can UPDATE it.

The general case for an UPDATE statement is as follows:

```
UPDATE tbl_name
    SET col_name1=expr1 [, col_name2=expr2 ...]
    [WHERE where_definition]
```

Let's pretend that Jack Sprat has moved house. We can update his record as follows:

```
UPDATE Staff
SET Address = '14 Cranberry St',
    City = 'Bendigo'
WHERE StaffID = 12345;
```

Notice that we only need to update the fields in the record which have changed. To update a whole class of things we can do something like the following:

```
UPDATE Staff
SET Wage = 170000
WHERE StaffPosition = 'Manager';
```

This would give all managers a cut in their pay, which may prove to be unpopular.

Make sure you include your WHERE conditions in your UPDATE statements. A UPDATE without a WHERE statement will apply that change to everything in your table.

The DELETE statement

Databases wouldn't be much use if you couldn't DELETE records when you're done with them. The general case for a DELETE statement is:

```
DELETE FROM tbl_name
       [WHERE where_definition]
```

If Betty Jones can no longer be reached at 0423 222 222 we can delete this record:

```
DELETE FROM StaffPhone
WHERE StaffID = '12347' AND
      PhoneNumber = '0423 222 222';
```

Make sure you include your WHERE conditions in your DELETE statements. A DELETE without a WHERE statement will delete everything from that table.

More SQL

There's a lot more to learn about things you can do in SQL. Make sure you take the time to read up on indexes as these are essential in assisting your database run quickly and smoothly. Once you move beyond simple queries you'll find it essential to have a reasonable understanding of `GROUP BY`, `HAVING` and the aggregate options. If your database supports sub-queries, you'll find that you're able to push more work on to it rather than having to do it in your code. Finally `UNION` and *outer joins* are very handy things to understand.

The Cheetah book covers SQL more fully on pages 58 - 75.

Chapter summary

- The `SELECT` statement allows us to pull data out from the database.
- We can limit the amount of data we get out from a `SELECT` statement by using conditional selections.
- We can join tables to gain further information in our `SELECT` statements.
- To order data we can use the `ORDER BY` instruction.
- The `CREATE` statement allows us to create tables to store our data within.
- The `INSERT` statement allows us to add data to a table.
- The `UPDATE` statement allows us to change parts of our data in a row.
- The `DELETE` statement allows us to delete rows from the database that we don't need any longer.

Appendix C. Remote connections and persistence

In this chapter...

In this chapter we will discuss connecting to databases remotely, setting access controls on remote connections and using encrypted connections. We will also cover persistent connections to databases by caching database handles and the use of `Apache::DBI`.

Establishing a remote connection

It is not always possible, or even desirable, to have your programs run on the same machine as the database application which you reference. In these cases you need some way to connect to the remote server. Many databases (and therefore their database drivers) support remote connection natively. For the other databases you need another strategy. The `DBD::Proxy` driver can be used to connect to any remote database and provides extra features such as encryption, compression and access controls.

Connecting with the database driver

Many databases, like MySQL support remote connections natively. To use this feature you pass the hostname and port to the database driver through the `DSN` in your call to the `connect` method.

```
my $database = "dbiX";
my $hostname = "example.perltraining.com.au";
my $port = 3306;

my $dsn = "dbi:mysql:database=$database;".
          "host=$host;port=$port";

my $dbh = DBI->connect($dsn, $username, $password, { AutoCommit => 1 })
        or die "Failed to connect to database: $DBI::errstr";
```

The order of the elements after the driver in the `DSN` doesn't matter. The port number can be omitted if the database server is running on its default port.

To discover whether your database driver can handle remote connections natively read its documentation at **perldoc DBD::*drivername***. For example the driver documentation for the `mysql` driver can be found at **perldoc DBD::mysql**.

Using DBD::Proxy

The Cheetah book covers the `DBD::Proxy` in more detail on pages 178 - 186.

Before connecting to a remote database with `DBD::Proxy` you must ensure that a Proxy server is running on the remote machine. Connecting to a remote database with `DBD::Proxy` is very similar to connecting remotely via a database driver. The biggest difference is that you have to provide a `dsn` option in your `DSN`. This is the `DSN` that the Proxy server will use to connect to your database on the remote server.

If your connection to the database would look like the following if your program was running on the machine locally:

```
my $database = "dbiX";

my $dsn = "dbi:mysql:database=$database";

my $dbh = DBI->connect($dsn, $username, $password, { AutoCommit => 1 })
        or die "Failed to connect to database: $DBI::errstr";
```

then your call to `connect` to the remote server should look like this:

```
my $database = "dbiX";
my $hostname = "example.perltraining.com.au";
my $port = 3306;

my $host_dsn = "dbi:mysql:database=$database";
my $dsn = "dbi:Proxy:hostname=$hostname;port=$port;".
          "dsn=$host_dsn";

my $dbh = DBI->connect($dsn, $username, $password, { AutoCommit => 1 })
        or die "Failed to connect to database: $DBI::errstr";
```

To find out more about `DBD::Proxy` read its documentation at **perldoc DBD::Proxy**.

Exercise

`DBD::Proxy` can be used to connect to local proxy servers. Your instructor will inform you of the hostname and port to provide in your connection.

1. Connect to the proxy and run a simple select command.

Running the proxy server

The `DBD::Proxy` driver requires a DBI Proxy server to be running on the remote machine. This Proxy server can be created by using the `DBI::ProxyServer` module. This module runs as a daemon on the machine with the database application and is implemented as a `RPC::PlServer` application.

`DBI::ProxyServer` provides a driver program `dbiproxy` which can be used as is, or modified to meet your needs.

`DBI::ProxyServer` takes a number of configuration options and must be given a value for `localport`. We will discuss some of these options in greater detail in the following sections.

To learn more about `DBI::ProxyServer` read its documentation at **perldoc DBI::ProxyServer**.

Access control

By careful choice of configuration options for `DBI::ProxyServer` you can restrict which machines can connect to your Proxy server, which users have access and even what SQL statements can be executed. This configuration must be specified in the configuration file. The name of the configuration file should be passed to `dbiproxy` on startup.

This access control goes in the the `clients` list. This list is an array of hashes with each hash covering one set of permissions. The array is searched from top to bottom until access is either permitted or denied.

The permission hashes have four primary keys:

1. `mask`: a regular expression which is matched against the hostname or IP address of the remote client
2. `accept`: whether to allow or deny access if these criteria are met
3. `users`: a list of usernames to verify
4. `sql`: a hash of which SQL statements are covered by this criteria.

Any of the `mask`, `users` and `sql` keys may be omitted. A missing `mask` implies all hosts, a missing `users` list implies all users, a missing `sql` hash implies all SQL.

```
'clients' => [
        {
                'mask' => '^example\.perltraining\.com\.au$',
                'accept' => 1,
                'users' => [ 'dbi1', 'dbi2', 'dbi3', 'dbi4', 'dbi5', 'dbi6' ],
                'sql' => {
                        'select_all_staff' => 'SELECT * FROM Staff',
                        'select_all_phone' => 'SELECT * FROM StaffPhone',
                        'select_all' => 'SELECT *
                                        FROM Staff s,
                                             Projects proj,
                                             StaffPhone ph
                                        WHERE s.StaffID = proj.StaffID AND
                                              s.StaffID =   ph.StaffID',
                        'select_staff_wage' => 'SELECT *
                                        FROM Staff s,
                                             Projects proj,
                                        WHERE s.StaffID = proj.StaffID AND
                                              s.StaffID = ?',
                }
        },
        {
                'mask' => 'perltraining\.com\.au$',
                'accept' => 1,
                'users' => [ 'pjf', 'jarich' ],
        },
        {
                accept => 0,
        }
],
```

The above access list allows `pjf` and `jarich` to connect from any machine in the `perltraining.com.au` network and execute any SQL statement. Users `dbi1` through to `dbi6` are *only* allowed if they connect from `example.perltraining.com.au` and may only run the four select statements specified. Finally we specify that any situation not covered by the previous controls is not accepted.

By specifying the SQL allowed to be executed we remove the need for the client to craft the SQL statements. Instead the client must call these SQL statements by the name we've given them in the `sql` hash. For example:

```
my $sth = $dbh->prepare("select_staff_wage");

$sth->execute($staffid);
```

Exercise

Your instructor will inform you of your username and password for this exercise.

1. Modify your previous program to use your new username and password. Try one of the SQL queries mentioned in the above access control hash.
2. What happens when you try a query that is not included?

Encrypted connections

Connecting to a remote database comes with all the usual network security issues. While you have some control over who has access to snoop on your own machine, you don't (usually) have the same control over who can snoop on your network traffic. As a result it's a good idea to encrypt connections to remote databases when you are dealing with sensitive data. Remember that client names, phone numbers and addresses can be very sensitive data in many situations.

The proxy driver supports two phase encryption. The cipher/key encryption is used during the login and authorisation phase. Once the client is authorised the driver will change to usercipher/userkey encryption. The cipher/key pair is a *host* based secret which is usually less secure than the usercipher/userkey secret. The usercipher/userkey is the user's private secret. The usercipher/userkey part is optional; if it is not present the less secure cipher/key encryption will be used throughout the connection.

Many of the cryptography modules available on CPAN can be used to provide the ciphers for the host and user encryption. The cryptography module must provide the following methods: `blocksize`, `encrypt` and `decrypt`. The choice of cipher is important and you should evaluate which is appropriate to your needs. In these notes we use `Crypt::IDEA` and `Crypt::Blowfish`.

To use encryption we need to set it up on both the Proxy server side and when making the connection using `DBD::Proxy`.

Client setup for encrypted connections

To encrypt connections we need to ensure that our cryptography modules are installed on the client machine as well as `RPC::PlClient`. Once this is done all we need to do is pass some extra parameters to the `connect` method as part of our `DSN`.

```
use DBI;
use Crypt::IDEA;

my $database = "dbiX";
my $hostname = "example.perltraining.com.au";
my $port = 3306;

my $cipher = "Crypt::IDEA";     # Host cipher

# Get the host key out of some file
open KEY, "<", "/etc/sharedkey.txt" or die "$!";
chomp(my $key = <KEY>);
close KEY;

my $usercipher = "Crypt::IDEA";   # or a different module if we wish

# Get my user key out of some file
open USERKEY, "<", "/home/$username/keys/training.key" or die "$!";
chomp(my $userkey = <USERKEY>);
close USERKEY;

my $host_dsn = "dbi:mysql:database=$database";
my $dsn = "dbi:Proxy:hostname=$hostname;port=$port;".
          "dsn=$host_dsn".
          "cipher=$cipher;key=$key;".
          "usercipher=$usercipher;userkey=$userkey";

my $dbh = DBI->connect($dsn, $username, $password, { AutoCommit => 1 })
        or die "Failed to connect to database: $DBI::errstr";
```

Exercises

Your instructor will tell you your username and password for this exercise. You’ll also be told where to find the shared host key and your private shared key.

1. Connect to the proxy using the host key and your private key.
2. As above, execute one of the SELECT queries provided to you.

Server setup for encrypted connections

To receive encrypted connections we need to make sure that the server has the appropriate cryptography modules installed as well as RPC::PlServer. Once this is done configuration is done in two parts: we need to add the cipher key and value to the clients access control list and we need to add the user keys and ciphers for second-stage encryption on a per user basis.

The DBI::ProxyServer configuration file is a Perl script. This means that at the top of the file you can include arbitrary Perl source. This is the ideal place to load in our first-stage cipher keys. Then we use the pre-loaded keys in our clients list.

Configuring for first-stage encryption

Using a portion of our configuration file from earlier we can configure for first-stage encryption as follows:

```
use Crypt::IDEA;

# Get the shared key for all PTA hosts
open KEY, "<", "/etc/sharedkey.txt" or die "$!";
chomp(my $key = <KEY>);
close KEY;

{
        # other configuration options

        ’clients’ => [
                {
                        ’mask’ => ’^example\.perltraining\.com\.au$’,
                        ’accept’ => 1,
                        ’users’ => [ ’dbi1’, ’dbi2’, ’dbi3’, ’dbi4’, ’dbi5’, ’dbi6’ ],
                        ’sql’ => {
                                ’select_all_staff’ => ’SELECT * FROM dbiX’,
                                .....
                        }
                        ’cipher’ => Crypt::IDEA->new(pack(’H*’, $key)),
                },
                {
                        ’mask’ => ’perltraining\.com\.au$’,
                        ’accept’ => 1,
                        ’users’ => [ ’pjf’, ’jarich’ ],
                        ’cipher’ => Crypt::IDEA->new(pack(’H*’, $key)),
                },
                {
                        accept => 0,
                }
        ],
}
```

This now insists that if any of our users attempt to connect to the database they need to send the same connection key as that stored in `/etc/sharedkey.txt`.

☞ It’s important to make sure that the host keys are kept up to date between the host and the client, otherwise encryption (and therefore connections) cannot take place. This key is a shared secret and as such should be kept in a manner that prevents disclosure to third parties, such as reading it from restricted file. It is good practice to provide different shared keys for each host.

Configuring for second-stage encryption

Configuration for second-stage encryption is done on a per-user basis. This allows different users to use different ciphers if they wish. We do this by expanding the ’users’ part of the access control criterion.

As we may potentially have many users mentioned in our access control list we can define a subroutine to fetch the user keys for us. Focusing on our `jarich` and `pjf` permissions we have:

```
use Crypt::IDEA;
use Crypt::Blowfish;

sub getKey {
        my ($keyname, $cipher) = @_;

        # Get the shared key out of the file
        open KEY, "<", "/etc/$keyname" or die "$!";
        chomp(my $key = <KEY>);
        close KEY;

        return $cipher->new(pack("H*", $key));
}


{
        # other configuration options

        ’clients’ => [
                {
                        ’mask’ => ’perltraining\.com\.au$’,
                        ’accept’ => 1,
                        ’users’ => [
                                {
                                        name => ’pjf’,
                                        cipher => getKey("pjf",
                                                 "Crypt::Blowfish"),
                                },
                                {
                                        name => ’jarich’,
                                        cipher => getKey("jarich",
                                                "Crypt::IDEA"),
                                }
                        ],
                        ’cipher’ => getKey("/etc/sharedkey.txt",
                                        "Crypt::IDEA"),
                },
                ....
        ],
}
```

This allows `jarich` and `pjf` to take advantage of second-stage encryption. `pjf` is expected to attempt to connect using the `Crypt::IDEA` for the first-stage encryption and to follow that with `Crypt::Blowfish` for the second-stage encryption.

☞ Just as the host keys are shared, the user keys are also shared. It is best practice to ensure that each and every user has their own private (shared) key.

Persistent connections

In some situations, usually to do with CGI programs, the programs using the database are many short-lived processes. In such cases the setup time for each connection to the database may form a noticeable part of the program running time. Since the connections to the database are often for the same user it would be ideal if some sort of persistence could be kept between program invocations such that the next program can reuse the database handle rather than reconnecting.

There are three primary ways of achieving this aim.

1. Using FastCGI
2. Using DBI’s `connect_cached` method with `DBD::Proxy`
3. Using `Apache::DBI`

FastCGI

FastCGI (provided by the `FCGI` module) allows CGI scripts to be persistent. Once the scripts are persistent the database handle becomes so too, and the problem is solved.

connect_cached and DBD::Proxy

DBI provides a useful method called `connect_cached`. This method is like `connect` except that it also stores the database handle in a hash associated with the given parameters. When another call is made to `connect_cached` with the same parameters the stored handle is returned.

The `connect_cached` method is only useful if there is a persistent process to keep the cached handles between program invocations. This can be managed by using the `DBD::Proxy` driver. Connections to databases can use the `DBD::Proxy` driver even if the database is local.

```
my $driver = "mysql";
my $database = "dbiX";
my $hostname = "localhost";
my $port = 9999;

my $dsn = "dbi:$driver:database=$database";

my $dbh = DBI->connect_cached("dbi:Proxy:hostname=$hostname;port=$port;dsn=$dsn",
                $username, $password, { AutoCommit => 1 })
        or die "Failed to connect to database: $DBI::errstr";
```

Caching connections can cause problems such as too many connections, so it should be used with care. Make sure you are certain that the database connection time is at fault before you prematurely optimise connection times away.

Using this method to cache connections can lead to problems when the same database handle is provided to more than one process. `connect_cached` is not smart enough to lock database handles which are in use.

Apache::DBI

Similarly to FastCGI, the `Apache::DBI` module gets passed the persistent database handle problems by relying on the `mod_perl` environment. `mod_perl` provides persistence to CGI scripts running within the `Apache` web server.

Attempting to connect to the database with the same parameters as a previous call results in `Apache::DBI` returning the previously generated database handle. However, if multiple scripts all ask for a database handle with the same parameters, `Apache::DBI` will create as many new connections as necessary to provide each process with its own, unique database connection. These will then be cached for later use.

Evaluating a caching strategy

The very first thing you have to do in this evaluation is ask yourself why you need database handle persistence. In many cases you won't need it.

Both the `Apache::DBI` and `connect_cached` strategies are very similar. The biggest difference is that while `connect_cached` is the most simple of the two, `Apache::DBI` handles multiple processes asking for database handles with the same signatures properly.

Chapter summary

- Many databases support remote connections natively. In these cases you can use your database's `DBD` driver to connect.
- If our database cannot handle remote connections then we can use `DBD::Proxy`.
- Using `DBD::Proxy` may also be desirable if you wish to encrypt your connection.
- Connection with `DBD::Proxy` requires the addition of a `dsn` keyword to the `DSN`.
- Using `DBD::Proxy` requires that a proxy server be running on the machine with the database. `DBI::ProxyServer` can be used to implement this.
- To configure access control you add the appropriate restrictions in the `clients` list.
- Encryption can be set up as host based and user based. In both cases encryption keys must be present on both machines.
- Persistent connections can be made by using FastCGI, `DBD::Proxy` and `connect_cached` or `Apache::DBI`.
- In most cases persistent connections are not required.

Appendix D. Managing Passwords

In this chapter...

In this chapter we'll discuss ways to avoid including usernames and password in our database program.

Supplying a password

So far we've assumed that the database usernames and passwords have been defined in our program. While this may be acceptable in some circumstances, there are many cases where this is not. It's good practice to have different passwords for your development, testing, and production environments. It's also rare that interaction with the database will be restricted to a single program; hard-coded passwords means that should the password change, all the programs need change with it. In an ideal world, your code shouldn't change just because your configuration has.

In the next few sections we'll cover a couple of ways to take your password out of your programs.

Environment variables

If you pass `DBI`'s `connect` method undefined values it will look in the program's environment to find the missing values. In this way you can explicitly provide the values to `DBI` that will not change between development stages and allow `DBI` to find all other required values from the program's environment. This can have the added effect that the same script invoked by two different users can use two different username/password combinations and even two different databases.

Some of the environment variables that `DBI` will use are:

Table D-1. DBI environment variables

Environment Variable	Value
`$ENV{DBI_USER}`	`$username`, the second argument to `connect`.
`$ENV{DBI_PASS}`	`$passwd`, the third argument to `connect`.
`$ENV{DBI_DSN}`	`$dsn`, the whole first argument to `connect`. This variable should have the format such as `dbi:Driver:databasename`.
`$ENV{DBI_AUTOPROXY}`	Setting this environment variable causes `DBI` to automatically proxy all database connections. This variable must contain the host and port part of the `DSN` that `DBD::Proxy` requires. `DBI` will then reroute the connection request via the proxy driver modifying the supplied `DSN` to include the host and port number. Additional parameters to the proxy module may also be added to this environment variable. For more information about `DBD::Proxy` and proxying database connections, see the chapter on remote database connections.

These variables can be used by providing undefined values in their place in the `connect` method. For example:

```
my $dbh = DBI->connect(undef, undef, undef, { AutoCommit => 1 });
```

If you choose to use any `DBI` environment variables instead of specifying them in your script make sure that you clearly document this. Missing environment variables are not treated as a cause for fatal error but you'll probably discover your connection failing a lot. It can also cause confusion when your program works for one user and not for everyone else.

Placing your password in an environment variable may pose security problems on some operating systems. Some operating systems allow other users to view the environment of your running process and therefore would allow other users to discover your database password. This behaviour may also be version dependent on your operating system. Many operating systems are moving away from providing non-privileged users with environment information about processes they don't own.

If you cannot be certain of the security of your environment variables then it is best not to use this feature of `DBI`.

Setting values such as your username and password in the environment can have further reaching consequences than you may intend. All programs called by you have access to read your environment variables. This means that you're providing your username and password to everything you call, not just your database programs.

Exercise

To set environment variables you can either set them in your environment:

```
export DBI_USER=dbiX # Using Bash.
setenv DBI_USER dbiX # Using csh/tcsh.
SET DBI_USER=dbiX # Under Win32.
```

or in your code:

```
$ENV{ DBI_USER } = "dbiX";      # The %ENV hash is a Perl special variable
```

1. Change an existing program that connects to the database to use the environment variables to provide the username and password. Check that it still connects and works as expected.

Using a configuration file

There are many good modules in Perl which provide easy access to configuration files. Some of these include `Config::Tiny`, `Config::General`, `Tie::Config` and `Config::Vars`. By storing your username, password, database name and even driver in a configuration file you limit the amount of work needing to be done when these things change.

While a full discussion of configuration files is out of the scope of this course, here is one way this can be achieved:

```
#---- Config file -----
<database>
    host   = example.perltraining.com.au
    user   = dbiX
    dbpass = verysecret
    dbname = dbiX
    driver = mysql
</database>

#--- Program -----
#!/usr/bin/perl -w
use strict;
use DBI;
use Config::General;

my $conf     = Config::General->new("config.txt");
my %config   = $conf->getall();
my %database = %{$config{database}};

my $dsn      = "dbi:$database{driver}:database=$database{dbname}";
my $username = $database{user};
my $passwd   = $database{dbpass};

my $dbh = DBI->connect($dsn, $username, $passwd, { AutoCommit => 1 })
        or die DBI->errstr;
```

You read more about `Config::General` on CPAN.

Exercise

1. The above script is in `exercises/config.pl`. Edit `exercises/config.txt` to match your database login details and run the script to make sure you can connect.

Prompting the user

When working with an interactive application, we have the option of not storing a password at all. Instead, we can prompt the user for the appropriate credentials when required. This is most appropriate when a program is to be used by multiple users.

When prompting the user for passwords, it's usually considered polite not to echo the password back while the user types it, to avoid it being accidently revealed to anyone. The code below demonstrates how we can replace password characters with asterisks when the user enters their password.

```
#!/usr/bin/perl -w
use strict;
use warnings;
use Term::Screen::ReadLine;

# Create a screen object
my $screen = Term::Screen::ReadLine->new();
```

```
# Clear the screen
$screen->clrscr;

# Ask for username
$screen->at(0,0)->puts("Username: ");
my $username = $screen->readline(ROW => 0, COL=>11);

# Ask for password, replace character presses with stars
$screen->at(1,0)->puts("Password: ");
my $password = $screen->readline(ROW => 1, COL=>11, PASSWORD => 1);

# Move to new line down and then remove our screen object
$screen->at(2,0);
undef $screen;

my $dbh = DBI->connect($dsn, $username, $password,
        {
                AutoCommit=>1
        }
) or die DBI->errstr;
```

Exercise

1. Change one of your programs to prompt for the username and password when run. Run the script to make sure that you can connect and execute the SQL.

Chapter summary

- Getting our username and password from an external source reduces program maintenance and improves security.
- `DBI` will look in certain environment variables if a password is not provided to the `connect` method.
- Storing our values in a configuration file offers an easy and portable way to store this information.
- For interactive programs prompting the user for their username and password allows multiple users to use the same script as themselves.

Appendix E. Binding variables

In this chapter...

In this chapter we will discuss binding parameters to statements and binding data to output columns.

Binding parameters to placeholders

Typically when we `prepare` and `execute` a statement our code looks like the following:

```
my $sth = $dbh->prepare("UPDATE Staff SET Wage = ? WHERE StaffID = ?")
        or die $dbh->errstr;
$sth->execute($wage, $staffid) or die $dbh->errstr;
```

Now in a simple case like this passing two arguments to `execute` is easy. However if our SQL generation resulted in a need to pass in a hundred bind values to `execute` this might become more difficult.

Instead of passing arguments to `execute` we can instead use the `bind_param` method to bind our variables to the correct positions. Each placeholder is numbered according to its position in the query.

```
my $sth = $dbh->prepare("UPDATE Staff SET Wage = ? WHERE StaffID = ?")
          or die $dbh->errstr;

$sth->bind_param( 1, $wage ); # Binds to the first placeholder
$sth->bind_param( 2, $staffid); # Binds to the second placeholder

$sth->execute or die $dbh->errstr;
```

Unfortunately, while this makes it easier to distinguish the meanings of our binding variables, if we do change our SQL to include an extra placeholder we must remember to re-number all the later bind parameters:

```
my $sth = $dbh->prepare("UPDATE Staff
                         SET Wage = ?, Position = ?
                         WHERE StaffID = ?")
        or die $dbh->errstr;

$sth->bind_param( 1, $wage );
$sth->bind_param( 2, $position );
$sth->bind_param( 3, $staffid);

$sth->execute or die $dbh->errstr;
```

The `bind_param` method cannot be used together with passing values to `execute`. If you pass bind values to `execute` all values set by calling `bind_param` will be ignored.

☞ The `bind_param` method takes a copy of the value passed into it. This means that if you change the value of the variable, `$wage` for example, after binding, but before execution, it is the previous value of `$wage` that will be used.

Exercise

1. Write a program that asks the user for a pair of wages. Select the first and last names of the staff who have wages between these figures from the database. Use `bind_param` to pass these values to the database.

Specifying the types of our bound variables

Perl's fundamental type is the scalar. Scalars can contain numbers or strings although this distinction can be quite fuzzy at times. When data is passed to `DBI`, `DBI` has to guess from it whether the database expects it to be a string or a number. Strings must be quoted when being passed to the database, but numbers need not be quoted.

If we treat the data like a number, by using it in numerical equations or comparisons, Perl knows that it's probably a number. If we treat the data like a string, by concatenating strings to it or using string comparisons, Perl knows that it's probably a string. Sometimes `DBI` gets the type wrong, perhaps because we've treated the data as both a number and a string. When using bind values passed to `execute` we can force the context we want by doing something like the following:

```
my $sth = $dbh->prepare("UPDATE Staff
                         SET Wage = ?, Position = ?
                         WHERE StaffID = ?")
        or die $dbh->errstr;

$sth->execute($wage+0, $position."", $staffid+0) or die $dbh->errstr;
```

Alternately by binding our variables using `bind_param` we can pass a hint to the database driver as to what type we expect the value to be. Typically the driver only cares whether the placeholder should be bound as a number or string:

```
use DBI qw(:sql_types);

...

$sth->bind_param( 1, $wage, { TYPE => SQL_INTEGER } );
$sth->bind_param( 2, $position, { TYPE => SQL_VARCHAR } );
```

Notice that in the above we pass in an option to `DBI` when we load the module. This says to `DBI` that we want to have access to the `sql_types` constants and allows us to use `SQL_INTEGER` and `SQL_VARCHAR` rather than the numbers they represent (4 and 12). As well as making our code more readable this means that if the actual values of these constants change in the future, our code will still work.

As the most common database hint passed to `bind_param` is the data type, `DBI` provides us with a short-cut:

```
use DBI qw(:sql_types);

...

$sth->bind_param( 1, $wage, SQL_INTEGER );
$sth->bind_param( 2, $position, SQL_VARCHAR );
```

The data type for a placeholder cannot be changed after the first call to `bind_param`. That is, if you prepare a call, bind a number of values to the statement, execute it, and then change the values for a second execution, you cannot change the data types the database is expecting. This should make sense, as the database types should not have changed between the two execute calls either.

Binding variables to output

Typically when we fetch data from a `SELECT` statement the code looks something like this:

```
while( @row = $sth->fetchrow_array) {
        my ($staffid, $firstname, $lastname, $address, $city, $state) = @row;

        print "$firstname $lastname lives at $address\n";
}
```

An alternative syntax allows us to get the database to give us the data in the variables we want without our having to copy it later. Binding columns is the most efficient way to fetch data.

```
my ( $staffid, $firstname, $lastname, $address, $city, $state);

my $sth = prepare("SELECT StaffId, FirstName, LastName, Address, City,
                          State
                   FROM Staff
                   WHERE City = ?");

$sth->execute(’Melbourne’);

$sth->bind_col(1, \$staffid);
$sth->bind_col(2, \$firstname);
$sth->bind_col(3, \$lastname);
$sth->bind_col(4, \$address);
$sth->bind_col(5, \$city);
$sth->bind_col(6, \$state);

while( $sth->fetchrow_arrayref() ) {
        print "$firstname $lastname lives at $address\n";
}
```

Like `bind_param`, `bind_col` takes a position and a reference to the variable to bind. The position refers to the position in the selected fields for which this variable should match. If you wish to ignore the data in a given column you can choose not to bind a variable to that column number.

It is important to bind the columns *after* the call to `execute` rather than before. This is to ensure that Perl has enough information to associate the output columns with the Perl variables.

Should you be selecting a large number of values from a statement you may find the plural form `bind_columns` easier. This is covered further below.

Bound variables can be changed between calls to execute and a variable can be unbound by passing `bind_col` the `undef` value in its place. Bound variables remain bound for as long as the statement handle exists, including successive calls to `execute`.

The binding is performed at a low level using Perl aliasing. This means that whenever rows are fetched from the database the variable is updated simply because it refers to the memory location that that fetched value now occupies. This makes using bound variables very efficient.

Some database drivers will allow you to bind columns prior to calling `execute` on that statement handle the first time. For maximum portability between drivers, however, this is not recommended.

Data types for column binding

Just as we can specify data types for parameter binding, data types can also be requested for bound columns. For example you may wish to specify that a date column comes out in the standard format for `SQL_DATETIME` which is `YYYY-MM-DD HH:MM:SS` rather than the database's native formatting. You'd do this with the following:

```
$sth->bind_col(1, \$date, { TYPE => SQL_DATETIME });
        # or using the short-cut
$sth->bind_col(1, \$date, SQL_DATETIME );
```

Setting the column output type affects the output type for that column for all subsequent calls to the statement handle. Unbinding `$date` will not change this.

You can specify data types for columns without mentioning any bind variable. Do this by passing in `undef`:

```
$sth->bind_col(1, undef, SQL_DATETIME );
```

Binding to all columns

We can bind values to all of the columns in a `SELECT` by using the `bind_columns` method. This takes a list of references to scalar values and binds them to each column of the result in turn. This method will die if the number of references does not match the number of fields.

```
my ( $staffid, $firstname, $lastname, $address, $city, $state);

my $sth = prepare("SELECT StaffId, FirstName, LastName, Address, City,
                          State
                   FROM Staff
                   WHERE City = ?");

$sth->execute('Melbourne');

$sth->bind_columns( \$staffid, \$firstname, \$lastname, \$address, \$city,
                    \$state );

# Alternately this can be written:
# $sth->bind_columns( \($staffid, $firstname, $lastname, $address, $city,
#                      $state) );

while( $sth->fetchrow_arrayref() ) {
        print "$firstname $lastname lives at $address\n";
}
```

Exercise

1. Modify your previous script to bind the output columns to variables. Use either `bind_col` or `bind_columns`.

Avoiding premature optimisation

Binding values to placeholders and variables to columns is very rarely necessary. Although binding variables to columns is the most efficient way to fetch data from the database it is not always the most convenient.

`DBI` has been written to be as fast as possible. Before rewriting your program to bind values and variables make sure that you properly profile your script to ensure that such an optimisation will be productive.

The `DBI` documentation includes some hints on how to test whether it is `DBI` or your program that is being slow.

Chapter summary

- We can bind the values of variables to placeholders with the `bind_param` method.
- `bind_param` allows us to specify the types of our bind values if the database driver might get it wrong.
- We can bind variables to the database fields output by using the `bind_col` and `bind_columns` methods.
- Binding variables to database fields is the most efficient way to fetch data from the database.
- In many cases these abilities are not required.

Colophon

```
Said the Lama to the Llama:
    u       t i m e                       Found:
                                          ------
    c                   n       t r o s   alarm
                                          binmode
    f   s o c k e t   i                   cate
                                          chop
    i               s         s e         continue
                                          exec
    r   c o n t i n u e     m i           exists
                                          fileno
  k s m s e e             a t n           getgrnam
                                          getpwnam
  n t r t m t k i l l   n n               gmtime
                                          goto
  i   a s i a         r u                 import
                                          int
  l   l i t c   o   g h t g n e l         kill
                                          length
      a x m   t   t         o             link
                                          map
        e g o   e         n               next
                                          oct
          g   g   o     e   o             redo
                                          send
  i m p o r t   c     l   d               sin
                                          socket
              t     i g e t p w n a m     sort
                                          time
                  f   r         s e n d   ucfirst
                                          untie
```

The `perlmonk` code that appears on the cover of this book was written by Frank Booth. When executed, it solves the output of the `find-a-func` puzzle, written by Stephen B. Jenkins.

The `perlmonk` code can be found in its native habitat on the PerlMonks website (http://perlmonks.org/index.pl?node_id=123321). The `find-a-func` code can also be found on PerlMonks (http://perlmonks.org/index.pl?node_id=108730).

Frank Booth is the patron saint of red cabbage, and more information about him can be found on his homenode (http://perlmonks.org/index.pl?node=frankus).