An Introduction to Perl
Perl Modules and DBI Database Access

IT Computing Services
June 2002
Version 1.0
Overview

• As Tom requested...

• Database access with Perl DBI, \textit{but first}...

• How to use DBI via Perl modules

• We also only talk about modules from the perspective of a consumer and not a producer. That is, this talk is not about writing Perl modules!

• This is only an introduction. There are cheatsheets/handouts at \url{http://support.itcs.northwestern.edu/ses/} that discuss Modules and DBI stuff differently.

• Next time, we'll talk get back to sorts.
Again, I challenge you all to give me some data to demo a sort with. \textit{Thanks!}
Extra Functionality in Perl

- Perl has the potential to do lots of interesting things. See [http://www.cpan.org/](http://www.cpan.org/)

- Modules are Perl's way of packaging code. That's how you get new third-party functionality.

- And a lot of third-party code tends to be implemented as objects.
Some Terminology

- objects are a collection of code and data

- attributes - formerly known as variables

- methods - formerly known as functions
• In many object-oriented programming languages nowadays, code (and their calls) act as an interface into the internal data structures and so programmers can change the internal representation anyway they want. Code acts as an interface.

At least that's the dream... All attributes in Perl are public.
Making a Module Available to Your Perl Script

• The way you tell the Perl interpreter to use a particular module is with `use` near the top of your program, after the first line of the Perl script.

```perl
#!/opt/local/bin/perl
use DBI;
```
Instantiating an Object

• To create a particular instance of an object, many objects have a `new()` method, but consult your module's documentation for the object's constructor.

For example, to open a new FTP connection using the `Net::FTP` module:

```perl
$ftp = Net::FTP->new("spike.itcs.northwestern.edu");
```

The new FTP object is created and its location in memory has been returned to be placed in a scalar variable.

*Scalars hold references (memory locations) in addition to numbers and strings!*
Using Modules' Methods and Attributes

• If you have a reference to an object, you can inspect or change its attributes:

```php
print ($ftp->Timeout . "\n");
$ftp->Timeout=300;
```

or invoke its methods:

```php
$ftp->put("/tmp/blah", "/home/barryc/blah");
$ftp->quit();
```
Diversion: Methods w/o Object References

- Sometimes you'll even see a module's method referred to in this type of notation:

  ```
  CGI::head()
  ```

  especially you don't have a reference to an object. Consult the module's documentation.
A Tale of Two Modules

• **DBI - Database Independent**
  The module that contains the database-independent *(or generic)* code for your Perl script to use. Serves as an interface for ...

• **DBD - Database Driver**
  The module that contains the database-specific *(or actual)* code used to access the database. Invoked indirectly thru DBI.

• Make sure your system administrator has installed both DBI and DBD (for the database-type you want to access) !!!
DBI/DBD Architecture

- from "Programming the Perl DBI" by By Alligator Descartes & Tim Bunce
  or http://www.oreilly.com/catalog/perldb/chapter/ch04.html

Your Perl script uses ... the generic DBI module which uses ... the database-specific module which accesses ... the database which stores the information.
Examples & The Table First

• Here's the table that I created in some of the examples:

  ```sql
  SQL> create table computers ( 
  2    name  varchar(8) not null, 
  3    os_rel integer 
  4  ) 
  5  ;
  
  Table created.
  
  SQL> describe computers;
  
  Name                      Null? Type
  --------------- -------- -------------------------------
  NAME        NOT NULL VARCHAR2(8) 
  OS_REL      NUMBER(38)
  
  SQL>
  ```
• Recall the following operations/phases in reading from a file?

```perl
open (INFILE, "<blah") ||
  die ("Can't open blah\n");
while (<INFILE>) {
  $input=$_;
  print $input;
}
close (INFILE);
```

Reading data from a database is very similar!
Connecting to a Database

- Let’s do a simple connection to the database first:

```perl
#!/usr/local/bin/perl
use DBI;

$dbh = DBI->connect("dbi:Oracle:sesdemo", "system", "missedem") ||
  die ("can't connect to Oracle database: $DBI::errstr\n");
$rc = $dbh->disconnect();print "$rc\n";
exit $rc;
```

Notice that the database connection object is "stored" in the "handle" `$_dbh` and is used to reference that particular connection.
Diversion: Recall error checking with `die()` and `warn()`

- Recall we use `die()` to print errors to standard error (STDERR) and abort the program.

  ```
  open (INFILE, "<blah") || die ("Can't open blah");
  ```

- The function `warn()` also reports errors to STDERR, but program execution continues.

  ```
  $rc= open (INFILE, "<blah");
  if ($rc!=0) {
     warn("Can't open blah");
     ... continue with error recovery ...
  }
  ```
Built-In Error Checking Instead

- DBI's built-in error checking can be turned on with an optional fourth parameter to \texttt{connect()}:

```perl
$dbh =
    DBI->connect("dbi:Oracle:sesdemo", "system", "missedem",
    { PrintError=>0,
      RaiseError=>1
    });
```

Don't report errors via \texttt{warn()}

Do report errors via \texttt{die()}

Note that the squigley-braces are needed!
• BTW, that fourth argument was a hash. In a similar way we did last month with a normal array:
  
  ```
  $flintstone[0]="Fred";
  $flintstone[1]="Barney";
  ```
  
  We could define the list by enumerating all the elements with:
  
  ```
  @flintstone=("Fred", "Barney");
  ```

• So, equivalent the fourth argument on the previous page is:
  
  ```
  $options{"PrintError"}=0;
  $options{"RaiseError"}=1;
  ```
  
  ```
  $dbh= DBI->connect("dbi:Oracle:sesdemo", "system", "missedem", 
                  %options);
  ```

  Also note that the hash here has a name `%options`, the one on the previous page was "anonymous".
Selecting Data

- After connecting to the database, you can execute SQL statements, and for each statement, a statement "handle" is needed... very much like we did with file handles before.

```perl
#!/usr/local/bin/perl
use DBI;

$dbh = DBI->connect("dbi:Oracle:sesdemo", "system", "missedem",
                    { PrintError=>0, RaiseError=>1 });

$stmt = $dbh->prepare("SELECT owner, table_name FROM DBA_TABLES");
```

... statement "handle" creation of `$sth` based on database "handle" `$dbh`
• The statement handle is used then to request that the statement get executed on the database and ...

• A **while** loop is used to fetch the rows of data one row at a time. When the array `@row` is empty, the loop exits.
Selecting Data Summary

- #!/usr/local/bin/perl
  use DBI;

  $dbh = DBI->connect("dbi:Oracle:sesdemo",
                      "system", "missedem",
                      { PrintError=>0,
                        RaiseError=>1 });

  $sth = $dbh->prepare(
    "SELECT owner, table_name FROM DBA_TABLES" );
  $sth->execute();

  while (@row=$sth->fetchrow_array()) {
    print ("Row: @row\n");
  }

  $dbh->disconnect();
Using Bind Parameters

• Here is an example that uses "bind parameters":

```php
$dbh = ...;

$stmt = $dbh->prepare(
    "SELECT owner, table_name FROM DBA_TABLES WHERE owner = ?",
);
$stmt->bind_param(1, "SYSTEM");

$stmt->execute();

while (@row=$stmt->fetchrow_array()) {
    print ("Row: @row\n");
}

$dbh->disconnect();
```

• BTW, the third *optional* parameter to `bind_param()` specifies the datatype, and the default type is `SQL_VARCHAR`.
Diversion:
"Importing a module's symbols"

• BTW, one more thing about modules ...

It might get tiring to say/type the full name to an object attribute or object method. Who likes to type a lot? And typing out the full name gets old fast.

You can tell Perl that whenever you say `SQL_VARCHAR`, you mean `DBI::SQL_VARCHAR`.

The process of doing this is known as "importing a module's symbols".
Symbols 2: How To Do It ...

• To do that, place the symbols you want to import as a list of arguments to the module name.
  
  use DBI ("SQL_VARCHAR", "SQL_INTEGER");

Some programmers use `qw()`: 
  
  use DBI qw(:sql_types);

Recall that `qw()` means quoted word from the area of lists in Perl. `qw()` treats everything in between the parentheses as a list of strings that are separated by spaces. Remember, no commas with `qw()`. 

Also, many packages have special arguments to import all the symbols or a subset of the symbols. For instance, many packages understand :all, but see the documentation of the module for all the details.
Insertion of Data and Good Situation for Bind Parameters

• Suppose we start off our program like this:

```perl
#!/usr/local/bin/perl
use DBI qw(:sql_types);
$name[0]="pigpen"; $os_rel[0]="11";
$dbh= DBI->connect("dbi:Oracle:sesdemo", "system",
                   "missedem",
                   {PrintError=>0, RaiseError=>1 })
...
Insertion of Data and Good Situation for Bind Parameters 2

• Now let's prepare and execute our SQL INSERT statement:

```php
$sth = $dbh->prepare('INSERT INTO computers (name, os_rel) VALUES (?,?)');
$sth->bind_param(1, 'dummy', SQL_VARCHAR);
$sth->bind_param(2, 42, SQL_INTEGER);
$i = 0;
while ($i < 2) {
    $sth->execute($name[$i], $os_rel[$i]);
    $i++;
}
$dbh->disconnect();
```

Note dummy values replaced with (1) and (2) at execute()!

• This gives us the effect of binding the data types once and then binding the data at the `execute()`!
Why Bother with `bind_param()`?

What is this a picture of?
Why Bother with `bind_param()`?

- You don't have to bind parameters, but if you don't, every time you run a query like it, your SQL will be parsed and a new execution plan will be created.

  **This can lead to an unhappy DBA!**

  If there are SQL statements that are the same but with different "constants", then using bind statements would be very good.

- Also, Perl is a type-less language. There are times you'll want to be explicit what datatype your values are and DBI will take care of the conversion for you.
Transactions

- Like before, suppose we start off our program like this:

```perl
#!/usr/local/bin/perl
use DBI qw(:sql_types);

$name[0]="pigpen";
$name[1]="rerun";

$os_rel[0]="11";
$os_rel[1]="11";

$dbh = DBI->connect("dbi:Oracle:sesdemo", "system",
                   "missedem",
                   {PrintError=>0, RaiseError=>1});
...
```
Transactions 2

• And we turned off Auto Commits to the database:

...  

```perl
print "AutoCommit: " . $dbh->{AutoCommit} . "\n";
# turn off AutoCommit
$dbh->{AutoCommit}=0;    # turn off AutoCommit
print "AutoCommit: " . $dbh->{AutoCommit} . "\n";
```

...  

Remember **true** is a non-zero number or non-null string. Remember **false** is a zero number or null string.

You may see no value when it is **false**!
Transactions 3

• Now let's prepare and execute our SQL INSERT statement:

```php
$sth = $dbh->prepare(
    "INSERT INTO computers ( name, os_rel ) VALUES (?,?)"; 
$sth->bind_param(1, "dummy", SQL_VARCHAR); 
$sth->bind_param(2, 42, SQL_INTEGER); 

$i=0; 
while ($i<2) {
    $sth->execute($name[$i], $os_rel[$i]); 
    $i++; 
}
```

• Now we can either abort the changes or commit the changes:

```php
$dbh->rollback();    # abort the changes to the database
# $dbh->commit();    # commit the changes to the database
```

```php
$dbh->disconnect();
```
disconnect() from database w/o commit() or rollback()

• From *Programming the Perl DBI*, pg. 158

• The transaction effect of explicitly disconnecting from a database while *AutoCommit* is disabled is, sadly, undefined. Some database systems, such as Oracle and Ingres, will automatically commit any outstanding changes.

• However, other database systems, such as Informix, will rollback any outstanding changes. Because of this, applications not using *AutoCommit* should ALWAYS explicitly call commit() or rollback() before calling disconnect().
die() or crash without commit() or rollback()

- From *Programming the Perl DBI*, pg. 158

- So what happens if you don't explicitly call disconnect(), or don't have the chance to because the program exists after a die? Well, because DBI handles are object references, we can be sure that Perl itself will call the **DESTROY** method for us on each handle if the program exits, ...

- The actual implementation of the **DESTROY** method is in the hands of the driver author. If the database handle is still connected, then it **SHOULD** automatically call rollback() (unless **AutoCommit** is enabled) before calling disconnect(). Calling rollback in **DESTROY** is critical. If the driver doesn't, then a program aborting due to a die() part way through a transaction may actually "accidentally" commit the incomplete transaction! Fortunately, all the drivers that we're aware of that support transactions do the right thing.
Executing *Quick-and-Dirty* SQL

- What if you find doing a `prepare()` and `execute()` cumbersome for some statements you'll do only once? There's `do()`

```perl
#!/usr/local/bin/perl
use DBI;
$dbh = DBI->connect("dbi:Oracle:sesdemo", "system", "missedem",
                    { PrintError=>0, RaiseError=>1 })

$rows = $dbh->do ("DELETE FROM computers WHERE name like '%e%'");
$dbh->disconnect();
```
Drawbacks with \texttt{do ()}

• From \textit{Programming the Perl DBI}, pg. 129

• To reiterate...
  ... the \texttt{do ()} method supplied by the DBI makes executing non-\texttt{SELECT} statements much simpler than repeatedly preparing and executing statements. This is achieved by simplifying wrapping the prepare and execute stages into one composite method.

• There is a drawback to doing this, however: performance. If you invoked \texttt{do ()} repeatedly to insert a huge number of rows into a table, you could be preparing a statement handle many times more than is required, especially if the statement contained placeholder variables.
quote()

• A handy tool to remember. It correctly quotes values as literal strings in the manner that is correct for your database. Good for portability!

• $name="%e\%";
  $rows= $dbh->do ("DELETE FROM computers WHERE name like " .
  $dbh->quote($name) )

• BTW, this does not satisfy the performance problem with \texttt{do()} mentioned in the previous slide.
Questions?

Thank you!