Writing MySQL Programs with Ruby

Paul DuBois
paul@kitebird.com

MySQL User Conference
April 12, 2003

1. Presentation Outline

- Introduction
  - Administrative Stuff
  - Purpose of this presentation
  - Overview of Ruby
- Ruby and MySQL
  - Overview of Ruby's MySQL interfaces
  - Installing the software
  - Connecting to the server, issuing queries
  - Transaction support
  - Metadata
- Sample Applications
  - Server monitoring
  - Ruby and MySQL on the web
  - Producing XML from MySQL data

2. Administrative Stuff

Hey, go get certified. :-)

3. Ground Rules for This Presentation

- Feel free to ask questions...
- But no difficult ones!

4. Warning

I'll be proceeding rather quickly through a fair amount of material, so...

- For a copy of this presentation, visit:
  http://www.kitebird.com/presentations/
- For more in-depth discussion of Ruby and MySQL:
  http://www.kitebird.com/articles/

5. Why a MySQL Presentation About Ruby?

To benefit you:

- So you know *what* you can do with MySQL in Ruby
- So you know *how* to do it

To benefit Ruby:

- To increase Ruby's exposure in the MySQL Community:
  - Ruby is big in Japan
  - Not so big elsewhere, e.g., in the U.S.

6. Is There Interest in MySQL and Ruby?

You be the judge.

In January, 2003, I wrote articles on the two APIs discussed in this presentation. Number of downloads as of April 8, 2003:

- Article 1: 1157 downloads
- Article 2: 821 downloads

Compare that with:

- Java JDBC article: 2053 downloads
- PHP PEAR article: 527 downloads
- Python DB-API article: 454 downloads
- *MySQL* Chapter 6 (C API): 2028 downloads

7. What You Can Expect to Learn Here

- If you don't know any Ruby:

  You'll learn a little about what the language looks like and how it works
- If you know some Ruby but haven't used it for database programming:

  You'll learn how to use it to access MySQL
- If you know some Ruby and have used it for database programming:

  What are you doing here?!?

8. Characteristics of the Ruby Language

(this is the obligatory buzzword slide)

- scripting language
- interpreted (primarily)
- extensible
- pure object-oriented language (everything's an object)
- weakly typed
- single inheritance (but mixins are allowed)
- true closures
- iterators
- multithreaded (even on Windows)
- mark and sweep garbage collection

9. Quick Overview of Ruby

- Purpose: So you know a little about what the language looks like and how it works
- Note: I'm leaving out a lot!

10. Data Types

Scalars: numbers, strings, nil

```
x = 1
x = "hello"
x = nil
```

(no surprises there, but note the weak typing)

Indexed types: arrays, hashes

```
x = [1, 2, ["a", "b", "c"], 4.5]
x = { "IA" => "Iowa", "OH" => "Ohio", "UT" => "Utah" }
```

11. Type Conversion

Conversion methods:

```
x.to_s    # to string
x.to_i    # to integer
x.to_f    # to float
```

Conversion often is required in contexts needing a particular type of value:

```
1 / 3         -> 0
1.to_f / 3    -> 0.333333
```

12. Control Structures

```
if i.nil?
    puts "i is nil"
elsif i < 10
    puts "i is less than 10"
else
    puts "i is 10 or greater"
end


while i < 10
    puts i
    i += 1
end
```

Shorthand control structures:

```
i = "I'm nil" if i == nil
i = i + 1 while i < 10
```

13. Blocks and Iterators

```
x = ['red', 'yellow', 'blue']

x.each do |val|
    puts val
end

x.each { |val|
    puts val
}
```

- **each** for an array produces values
- **each** for a hash produces hash keys and values
  (No need to use key to look up value)
- Iterator methods can be applied to many kinds of objects. They tend to replace **while** loops.

14. Exceptions

Exceptions are raised when errors occur:

- Similar to Java's **try/catch/finally**
- Similar to Python's **try/except**

Syntax:

```
begin
    ... stuff that may fail ...
rescue [exception-type => var ]
    ... error handling code...
ensure
    ... cleanup code...
end
```

- **rescue** and **ensure** are optional
- There may be more than one **rescue** clause (e.g., to handle different exception types)

15. Printing

```
puts "hello, world"
str = "hello, world"
puts str
h, w = "hello", "world"
puts h + ", " + w
printf "%s, %s\n", h, w
puts "#{h}, #{w}"
```

16. End of Ruby Summary

Okay, you're now a Ruby expert. :-)

17. Ruby Interfaces for MySQL

You have a choice of two interfaces:

- Lower level: Ruby MySQL module
- Higher level: Ruby DBI module

18. Ruby Interfaces for MySQL (cont.)

Lower level: Ruby MySQL module:

- Written in C as a Ruby extension library
- Acts as wrapper around MySQL C API
- Used much like C API
- Database-dependent (MySQL-specific)

19. Ruby Interfaces for MySQL (cont.)

Higher level: Ruby DBI module:

- Written in Ruby
- Somewhat like Perl DBI (or JDBC, DB-API, ...)
- Two-level architecture:
  - Database-independent (abstract) level
  - Database-dependent driver level
- Works with MySQL (driver uses Ruby MySQL module)
- Other database drivers are available: PostgreSQL, InterBase, ODBC, Oracle...

20. Module Commonalities

The MySQL and DBI modules have many things in common.

Both have methods for:

- Connecting to database server
- Disconnecting from database server
- Issuing queries
- Fetching result sets
- Obtaining query metadata
- Handling special characters

21. Module Differences

The MySQL and DBI modules have differences, too.

The MySQL module provides:

- Direct access to row count
- Access to other MySQL-specific information not exposed by DBI

The DBI module provides:

- More flexibility in issuing queries
- More flexibility in fetching rows
- More flexibility in accessing row contents
- Placeholders and parameter binding
- Transaction abstraction

22. Using What You Already Know

- Experience with the C API or PHP will help you use the Ruby MySQL module
- Experience with high-level interfaces such as Perl DBI, Java JDBC, Python DB-API, or PHP PEAR will help you use the Ruby DBI module

23. Adapting What You Already Know

You can write code that looks the way you might write it in other languages. For example, a loop:

```
i = 0
while i < myarray.size do
    puts myarray[i]
    i += 1
end
```

Eventually, you'll begin to adopt a more Ruby-like way of doing things:

- Iterators
- Code blocks

```
myarray.each do |val|
    puts val
end
```

24. Platform Support for MySQL with Ruby

- Unix
- Windows
  (*if* you have a Unix-like environment, e.g., Cygwin)

25. Installing the MySQL Module on Unix

- Visit Tomita Masahiro's site to get the distribution:
  http://www.tmtm.org/en/mysql/ruby/
- Unpack the distribution, change location into the resulting directory:
  % **tar zxf mysql-ruby-2.4.4.tar.gz**
  % **cd mysql-ruby-2.4.4**
- Build and install the distribution:
  % **ruby extconf.rb --with-mysql-config**
  % **make**
  % **make install**

26. Installing the MySQL Module on Windows

You need a Unix-like environment to work with.

Therefore:

- Install Cygwin
- Use the Unix installation instructions

27. Sample MySQL Module Script

The script performs these operations:

- Connect to the MySQL server
- Create and populate a table
- Retrieve and display the table contents
- Disconnect from the server
- Handle errors

28. Sample MySQL Module Script (cont.)

Script framework (connect, disconnect, handle errors):

```
require "mysql"

begin
    # connect to the MySQL server
    dbh = Mysql.real_connect("localhost",
                             "testuser", "testpass", "test")

    # ... run queries ...

rescue MysqlError => e
    puts "Error code: #{e.errno}"
    puts "Error message: #{e.error}"
ensure
    # disconnect from server
    dbh.close if dbh
end
```

29. Sample MySQL Module Script (cont.)

Use the database handle to create and populate a table:

```
dbh.query("DROP TABLE IF EXISTS ucctalk")
dbh.query("CREATE TABLE ucctalk
            (speaker CHAR(30), title CHAR(60))")
dbh.query("
    INSERT INTO ucctalk (speaker,title)
    VALUES
    ('Jeremy Zawodny', 'Optimizing MySQL'),
    ('Sanja Byelkin', 'Sub-Queries in MySQL'),
    ('Tim Bunce', 'Advanced Perl DBI')
")
puts "#{dbh.affected_rows} rows were inserted"
```

30. Sample MySQL Module Script (cont.)

Use the database handle to run a retrieval and produce a result set object. Use the object to access the retrieved data, then release the result set:

```
res = dbh.query("SELECT speaker, title FROM ucctalk")
while row = res.fetch_row do
    puts row.join(", ")
end
puts "#{res.num_rows} rows were selected"
res.free
```

31. MySQL Module Classes

- **Mysql**: The main class, for connecting and issuing queries
- **MysqlRes**: For result sets
- **MysqlField**: For column metadata
- **MysqlError**: For exceptions

Other than **MysqlError**, these classes correspond to C API structs: **MYSQL**, **MYSQL_RES**, **MYSQL_FIELD**.

32. MySQL Module Methods

You can do pretty much everything with the following methods:

- **Mysql.real_connect**: connect to server
- **dbh.close**: disconnect from server
- **dbh.query**: issue query/generate result set
- **res.fetch_row**: fetch row of result set as array
- **res.free**: free result set

Notice the similarity to the C API function names?

Other row-fetching methods:

- **res.fetch_hash**: fetch row of result set as hash
- **res.each/each_hash**: row iterators

33. MySQL Module Methods (cont.)

Other useful methods:

- **dbh.escape_string/quote**: escape special characters
- **dbh.affected_rows**: number of rows affected by update
- **res.num_rows**: number of rows returned by retrieval
- **res.num_fields**: number of columns returned by retrieval
- **res.fetch_fields**: column metadata

Error information: obtain from exception object

34. Installing Ruby DBI

- Make sure you're installed the Ruby MySQL module first
- Visit the Ruby DBI site to get the distribution:
  http://ruby-dbi.sourceforge.net/
- Unpack the distribution, change location into the resulting directory:
  % **tar zxf ruby-dbi-all-0.0.18.tar.gz**
  % **cd ruby-dbi-all**
- Configure, build, and install the distribution:
  % **ruby setup.rb config --with=dbi,dbd_mysql**
  % **ruby setup.rb setup**
  % **ruby setup.rb install**
- Note: For the most recent source, use the CVS repository instead

35. Sample DBI Module Script

The script performs the same operations as the MySQL module script:

- Connect to the MySQL server
- Create and populate a table
- Retrieve and display the table contents
- Disconnect from the server
- Handle errors

36. Sample DBI Module Script (cont.)

Script framework (connect, disconnect, handle errors):

```
require "dbi"

begin
    # connect to the MySQL server
    dbh = DBI.connect("dbi:Mysql:test:localhost",
                      "testuser", "testpass")

    # ... run queries ...

rescue DBI::DatabaseError => e
    puts "Error code: #{e.err}"
    puts "Error message: #{e.errstr}"
ensure
    # disconnect from server
    dbh.disconnect if dbh
end
```

37. Sample DBI Module Script (cont.)

Use the database handle to create and populate a table:

```
dbh.do("DROP TABLE IF EXISTS ucctalk")
dbh.do("CREATE TABLE ucctalk
        (speaker CHAR(30), title CHAR(60))")
nrows = dbh.do("
    INSERT INTO ucctalk (speaker,title)
    VALUES
    ('Jeremy Zawodny', 'Optimizing MySQL'),
    ('Sanja Byelkin', 'Sub-Queries in MySQL'),
    ('Tim Bunce', 'Advanced Perl DBI')
")
puts "#{nrows} rows were inserted"
```

38. Sample DBI Module Script (cont.)

Use the database handle to run a retrieval and produce a statement handle. Use the statement handle to access the retrieved data, then free the result set:

```
sth = dbh.execute("SELECT speaker, title FROM ucctalk")
nrows = 0
sth.fetch_array do |row|
    puts row.join(", ")
    nrows += 1
end
sth.finish
puts "#{nrows} rows were selected"
```

Note that you determine the row count by counting rows as they are fetched.

39. DBI Classes

- **DBI::DatabaseHandle**: connection object
- **DBI::StatementHandle**: statement object
- **DBI::Row**: object containing row contents
- **DBI::ColumnInfo**: object containing column metadata
- Exception classes, e.g., **DBI::DatabaseError**

40. DBI Module Methods

You can do pretty much everything with the following methods:

- **DBI.connect**: connect to server
- **dbh.disconnect**: disconnect from server
- **dbh.do**: perform statement
- **dbh.prepare**: prepare statement for execution
- **sth.execute**: execute statement/generate result set
- Row-fetching methods (various forms)
- **res.finish**: release result set

41. DBI Module Methods (cont.)

Convenience methods:

- **dbh.quote**: escape special characters
- **dbh.select_one**: retrieve first row as array
- **dbh.select_all**: retrieve all rows, as array of **DBI::Row** objects
- **sth.fetch_all**: retrieve all rows, as array of **DBI::Row** objects

42. Row-Fetching Methods

- **fetch_array**: fetch row as array
- **fetch_hash**: fetch row as hash
- **fetch**: fetch row as **DBI::Row** object
  - Can also be used as iterator
  - **each** is a synonymous iterator
- All methods return nil if no more rows available

There is no hash version of **fetch** because **DBI::Row** values can be accessed either by numeric or named indexes

Example:

```
sth.fetch do |row|
    puts row.join(", ")
end
```

43. Column-Access Methods

Columns for rows returned as arrays or hashes can be accessed only by numeric index or column name:

- **row[0], row[1], ...**
- **row["speaker"], row["title"], ...**

44. Column-Access Methods (cont.)

DBI::Row objects have several column-access methods.

By numeric index:

- **row[0], row[1], ...**
- **row.by_index(0), row.by_index(1), ...**

By column name:

- **row["speaker"], row["title"], ...**
- **row.by_field("speaker"), row.by_field("title"), ...**

By using the **each_with_name** iterator::

```
row.each_with_name do |col_val, col_name|
    ...
end
```

45. Column-Access Caveats

Values returned by the MySQL driver are not always strings

- Values are coerced to specific types:
  - integer -> FixNum type
  - float -> Float type
  - string -> String type
  - NULL is returned as nil (NilClass type)
- Coercion is not universal among DBI drivers
- PostgreSQL driver performs coercion; others may not

46. Column-Access Caveats (cont.)

- As a result of coercion, you may need to re-convert values, depending on the context in which you use them.
- DATETIME and TIMESTAMP results may not look like you expect:

```
query = "SELECT DATE_ADD('2003-01-01 00:00:00',INTERVAL 0 DAY)"
puts dbh.select_one(query)[0]

Result: 2003-1-2 0:0:0.0
```

47. Methods That Take Code Blocks

Some methods take code blocks. This is useful because handle cleanup is automatic.

Example:

```
dbh.execute("SELECT speaker, title FROM ucctalk") do |sth|
    sth.fetch_array do |row|
        puts row.join(", ")
    end
end
```

Notice that no **sth.finish** call is needed to release the result set.

48. Parameter Binding

Use parameter binding to handle quoting issues for special characters and NULL values:

- Use **?** as the placeholder character in query strings
- To bind data values to placeholders, pass them to the appropriate method
- To bind NULL, use nil
- Methods that bind values: **do**, **execute**, **select_one**, **select_all**

Example:

```
dbh.do("INSERT INTO ucctalk (speaker,title) VALUES (?,?)",
       "Heikki Tuuri", "The Wonders of InnoDB")
```

Alternatively, use **bind_param** to bind values explicitly to prepared statement.

49. Transaction Support

MySQL module:

- No direct transaction support
- Issue transaction-related SQL statements directly (BEGIN, COMMIT, ROLLBACK)

DBI module:

- DBI level has a transaction abstraction:
  - The **AutoCommit** attribute controls auto-commit mode
  - The **commit** and **rollback** methods terminate transactions
  - The **transaction** method handles commit and rollback for you (but you must disable auto-commit mode yourself)
- Caveat: MySQL DBD does not currently support the abstraction except in the CVS repository

50. Using the DBI Transaction Abstraction (I)

Perform a transaction by invoking **commit** and **rollback** explicitly:

```
dbh['AutoCommit'] = false
begin
    dbh.do("UPDATE account SET balance = balance - 50
            WHERE name = 'bill'")
    dbh.do("UPDATE account SET balance = balance + 50
            WHERE name = 'bob'")
    dbh.commit
rescue
    puts "transaction failed"
    dbh.rollback
end
```

51. Using the DBI Transaction Abstraction (II)

Perform a transaction with the **transaction** method:

```
dbh['AutoCommit'] = false
dbh.transaction do |dbh|
    dbh.do("UPDATE account SET balance = balance - 50
            WHERE name = 'bill'")
    dbh.do("UPDATE account SET balance = balance + 50
            WHERE name = 'bob'")
end
```

52. Handling Query Metadata

Availability of metadata depends on statement type:

- For statements that return no result set (updates):
  - Count of rows affected
- For statements that return a result set (retrievals):
  - Number of columns
  - Number of rows
  - Per-column information (name, type, ...)

53. Metadata for Updates (MySQL module)

Row count using database handle:

```
dbh.query(query_string)
nrows = dbh.affected_rows
```

54. Metadata for Retrievals (MySQL module)

Get a result set object:

```
res = dbh.query(query_string)
# access query metadata here, using res
res.free
# metadata no longer available at this point
```

While the result set object is valid, use it to obtain metadata.

55. Metadata for Retrievals (MySQL module)

- Use result set to get row and column counts:

```
nrows = res.num_rows
ncols = res.num_fields
```

- Per-column information is available in **MysqlField** objects:

```
res.fetch_fields.each do |info|
    # access metadata using info
end
```

- **MysqlField** methods include:

```
info.name     # Column name
info.table    # Table name
info.def      # Default value
(etc.)
```

The methods correspond to C API **MYSQL_FIELD** members.

56. Metadata for Updates (DBI module)

The Row Processed Count (RPC) can be obtained two ways, depending on how you execute the update:

- Use a database handle:

```
nrows = dbh.do(query_string)
```

- Use a statement handle:

```
sth = dbh.prepare(query_string)
sth.execute
nrows = sth.rows
```

57. Metadata for Retrievals (DBI module)

Use a statement handle like this:

```
dbh.execute(query_string) do |sth|
    # access query metadata here, using sth
end
# metadata no longer available at this point
```

Or like this:

```
sth = dbh.prepare(query_string)
sth.execute
# access query metadata here, using sth
sth.finish
# metadata no longer available at this point
```

While the statement handle is valid, use it to obtain metadata.

58. Metadata for Retrievals (DBI module)

Unlike the MySQL module, with DBI the row and column counts are not available directly.

To get the row count, count the rows yourself:

- Count as you fetch
- Fetch into data structure, then see how many elements it has (**dbh.select_all**, **sth.fetch_all**)
- Do not use **sth.rows**

To get the column count, count the number of column names:

```
ncols = sth.column_names.size
```

59. Metadata for Retrievals (DBI module)

Per-column information is available from the **column_info** method:

```
sth.column_info.each do |info|
    # access metadata using info
end
```

column_info methods include:

```
info.name         # Column name
info.precision    # Precision
info.scale        # Scale
(etc.)
```

60. Some Sample Applications

A few ways to use Ruby with MySQL:

- Monitoring the MySQL server
- Using Ruby to connect MySQL to the web
- Generating XML from MySQL data

61. Monitoring the MySQL Server

Task: Show selected status variables, both as current value and as change from previous value.

Logic of the script is as follows:

- Grab current values
- Print status, comparing to previous values
- Wait
- Loop

62. Monitoring the MySQL Server (cont.)

Script framework:

```
# list of status variables to monitor
stat_vars = [
    "Connections",
    "Questions",
    "Uptime"
]

begin
    dbh = DBI.connect("dbi:Mysql:test:localhost",
                      "testuser", "testpass")
    cur_stat = get_server_status(dbh)
    while true do
        old_stat = cur_stat
        cur_stat = get_server_status(dbh)
        display_status_change(stat_vars, old_stat, cur_stat)
        sleep 10
    end
rescue DBI::DatabaseError => e
    puts "Error code: #{e.err}"
    puts "Error message: #{e.errstr}"
ensure
    dbh.disconnect if dbh
end
```

63. Monitoring the MySQL Server (cont.)

Getting current server status:

```
def get_server_status(dbh)
    stat_hash = Hash.new
    dbh.select_all("SHOW STATUS").each do |name, value|
        stat_hash[name] = value
    end
    stat_hash
end
```

Printing the status:

```
def display_status_change(stat_vars, old_stat, cur_stat)
    puts Time.now.strftime("%Y-%m-%d %H:%M:%S")
    stat_vars.each do |var_name|
        old_val = old_stat[var_name].to_i
        cur_val = cur_stat[var_name].to_i
        diff = cur_val - old_val
        printf "%15s: %8d %8d\n", var_name, cur_val, diff
    end
    puts "-" * 20
end
```

64. Using Ruby on the Web

Some Ruby support for web programming:

- CGI module
- mod_ruby (analogous to mod_perl)
- eruby (embedded Ruby)

65. Using the Ruby CGI Module

Access the module and create a CGI object:

```
require "cgi"
cgi = CGI.new("html3")
```

The CGI object provides:

- Methods for generating HTML (reminiscent of CGI.pm):

```
cgi.p { "This is a paragraph." }
cgi.ul { cgi.li { "Item 1" } + cgi.li { "Item 2" } }
```

- Access to input parameters:

```
cgi['param_name']
```

66. Ruby CGI Script

Task: Select and display baseball players born on a given day of the year.

The table contains rows that look like this:

```
mysql> SELECT * FROM ballplayer LIMIT 5;
+----------+-----------+-----------+------------+----------+
| lastname | firstname | birthyear | birthmonth | birthday |
+----------+-----------+-----------+------------+----------+
| AARON    | HANK      |      1934 |          2 |        5 |
| AARON    | TOMMIE    |      1939 |          8 |        5 |
| AASE     | DON       |      1954 |          9 |        8 |
| ABAD     | ANDY      |      1972 |          8 |       25 |
| ABADIE   | JOHN      |      1854 |         11 |        4 |
+----------+-----------+-----------+------------+----------+
```

67. Ruby CGI Script (cont.)

Initial setup and query processing

```
require "dbi"
require "cgi"

cgi = CGI.new("html3")
month = cgi['month']
day = cgi['day']

dbh = DBI.connect("dbi:Mysql:test:localhost",
                  "testuser", "testpass")
query_string =
"SELECT
    CONCAT(firstname,' ',lastname,' (',birthyear,')') AS name
FROM ballplayer
WHERE birthmonth = ?  AND birthday = ?
ORDER BY lastname, firstname"
rows = dbh.select_all(query_string, month, day)
```

68. Ruby CGI Script (cont.)

Generate the page:

```
# generate list of names and birth years
list = ""
rows.each do |row|
    list += cgi.li { CGI.escapeHTML(row[0]) }
end

# produce web page
cgi.out {
  cgi.html {
    cgi.head {
      cgi.title { "Baseball Player Birthdays" }
    } +
    cgi.body {
      cgi.p { "Baseball player birthdays for #{month}/#{day}:" }
      cgi.ul { list }
    }
  }
}
```

69. Producing XML from MySQL Data

Task: Pull out baseball players born on a given day of the year, and convert the result to XML.

To get the data, do this:

```
query_string =
"SELECT * FROM ballplayer
WHERE birthmonth = ? AND birthday = ?
ORDER BY lastname, firstname"

rows = []
begin
    dbh = DBI.connect("dbi:Mysql:test:localhost",
                      "testuser", "testpass")
    rows = dbh.select_all(query_string, month, day)
rescue DBI::DatabaseError => e
    puts "Error code: #{e.err}"
    puts "Error message: #{e.errstr}"
ensure
    dbh.disconnect if dbh
end
```

70. Producing XML from MySQL Data (cont.)

The XML produced from the result set should look something like this:

```
<?xml version="1.0"?>
<ballplayers>
  <ballplayer>
    <lastname>BARTHOLOMEW</lastname>
    <firstname>LES</firstname>
    <birthyear>1903</birthyear>
    <birthmonth>4</birthmonth>
    <birthday>4</birthday>
  </ballplayer>
  <ballplayer>
    <lastname>BOKINA</lastname>
    <firstname>JOE</firstname>
    <birthyear>1910</birthyear>
    <birthmonth>4</birthmonth>
    <birthday>4</birthday>
  </ballplayer>
  ...
</ballplayers>
```

71. Producing XML from MySQL Data (cont.)

To produce XML from the result set, you have several choices:

- Generate markup yourself
- Use **XML::Utils::XMLFormatter** from the DBI module
- Use an XML module (e.g., REXML)

72. Generating XML Directly

Print the element tags and content "manually":

```
str = "<?xml version=\"1.0\"?>\n"
str += "<ballplayers>\n"
rows.each do |row|
    str += "  <ballplayer>\n"
    row.each_with_name do |val, name|
        str += "    <#{name}>#{row[name]}</#{name}>\n"
    end
    str += "  </ballplayer>\n"
end
str += "</ballplayers>\n"
puts str
```

Problem: This code doesn't encode special characters.

73. Generating XML with XMLFormatter

The DBI module includes some handy utilities.

One of them is **XMLFormatter**, which makes it easy to turn a set of rows into XML:

```
DBI::Utils::XMLFormatter.table(rows,
                               "ballplayers",
                               "ballplayer")
```

Drawback of XMLFormatter: writes only to streams

74. Generating XML with the REXML Module

The REXML module provides more flexibility, at the cost of a little more work:

```
require "rexml/document"
include REXML

doc = Document.new("<ballplayers/>")
rows.each do |row|
    row_elt = Element.new("ballplayer")
    row.each_with_name do |val, name|
        col_elt = Element.new(name)
        col_elt.text = val.to_s
        row_elt.elements << col_elt
    end
    doc.root.elements << row_elt
end
doc.write($stdout,0)
```

REXML can write to streams or strings. For example, to generate the output into a string, do this:

```
str = ""
doc.write(str,0)
```

75. Other Stuff to Know About

- **irb**: interactive Ruby; useful for experimenting and one-liners
- **sqlsh.rb**: command-line DBI shell (similar to Perl **dbish**)

76. Want More Information?

- Main Ruby site:
  http://www.ruby-lang.org/en/
- Ruby Application Archive:
  http://raa.ruby-lang.org/
- Ruby/MySQL Articles:
  http://www.kitebird.com/articles/
- Ruby MySQL module:
  http://www.tmtm.org/en/mysql/ruby/
- Ruby DBI module:
  http://ruby-dbi.sourceforge.net/
- Ruby REXML module:
  http://www.germane-software.com/software/rexml/

77. The End

Thanks for attending!