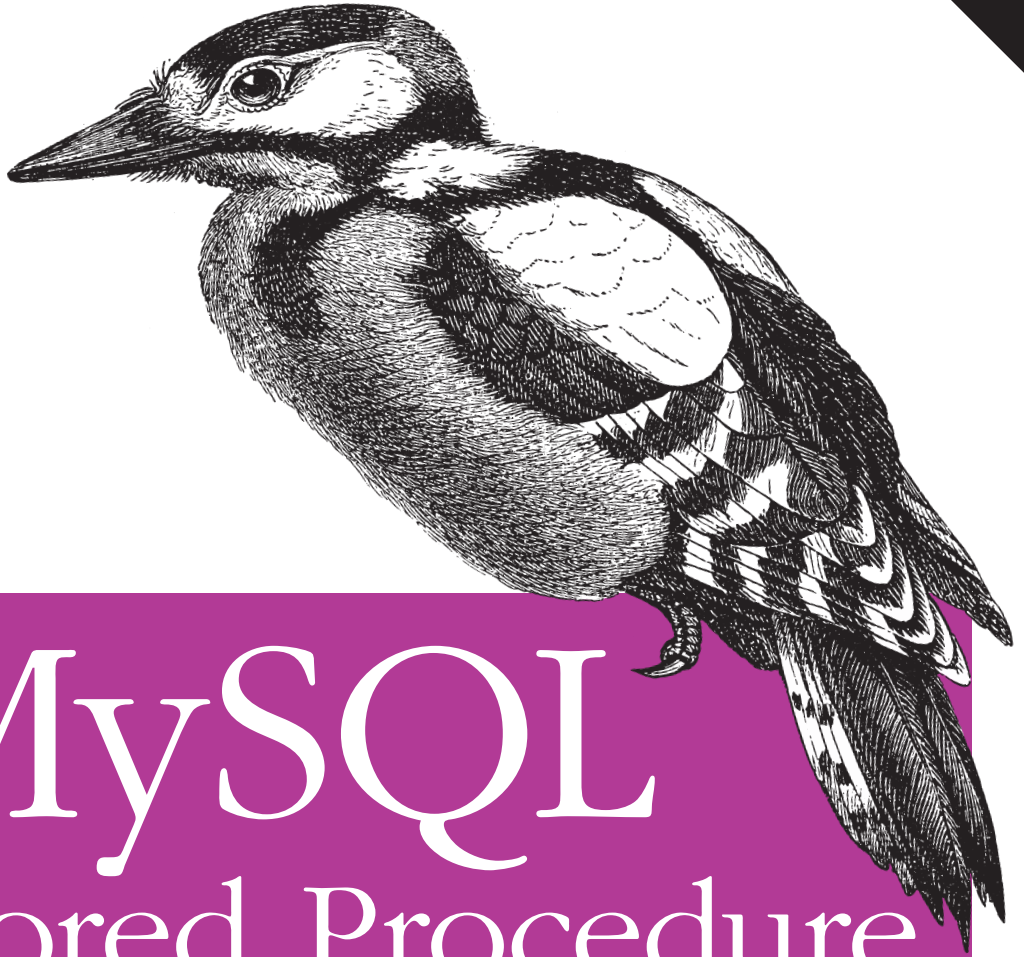


*Building High-Performance Web Applications
with PHP, Perl, Python, Java & .NET*

Covers MySQL 5



MySQL

Stored Procedure

Programming

O'REILLY®

*Guy Harrison
with Steven Feuerstein*

MySQL Stored Procedure Programming

by Guy Harrison with Steven Feuerstein

Copyright © 2006 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Deborah Russell

Production Editor: Adam Witwer

Production Services: Argosy Publishing

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrators: Robert Romano, Jessamyn Read,
and Lesley Borash

Printing History:

March 2006: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *MySQL Stored Procedure Programming*, the image of a middle spotted woodpecker, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-10089-2

[M]

This excerpt is protected by copyright law. It is your responsibility to obtain permissions necessary for any proposed use of this material. Please direct your inquiries to permissions@oreilly.com.

Error Handling

The perfect programmer, living in a perfect world, would always write programs that anticipate every possible circumstance. Those programs would either always work correctly, or fail “gracefully” by providing comprehensive diagnostic information to the support team and very readable messages to the user.

For a certain class of applications—software supporting life support systems or the space shuttle, for instance—this level of perfection is actually a part of the requirements, because any unexpected failure of the software would be catastrophic. However, in the world of business applications, we usually make certain assumptions about our execution environment—we assume the MySQL server will be running, that our tables have not been dropped, that the host machine is not on fire, and so on. If any of these conditions occurs, then we accept that our application will fail. In many other circumstances, we can and should anticipate potential failures and write code to manage those situations. This is where exception handling comes into play.

When a stored program encounters an error condition, execution ceases and an error is returned to the calling application. That’s the default behavior. What if we need a different kind of behavior? What if, for example, we want to trap that error, log it, or report on it, and then continue execution of our application? For that kind of control, we need to define exception handlers in our programs.

When developing MySQL stored programs, a very common scenario—fetching to the end of a result set—also requires that we define an *exception handler*.

In this chapter we explain how to create various types of exception handlers and how to improve the readability of error handling by using “named” conditions. We also identify several gaps in exception-handling functionality in MySQL 5, and explore ways of compensating for these omissions.

Introduction to Error Handling

Let’s begin by looking at several examples of stored program error handling.

A Simple First Example

Consider a simple stored procedure that creates a location record, as shown in Example 6-1.

Example 6-1. Simple stored procedure without error handling

```
CREATE PROCEDURE sp_add_location
    (in_location  VARCHAR(30),
     in_address1  VARCHAR(30),
     in_address2  VARCHAR(30),
     zipcode      VARCHAR(10))
    MODIFIES SQL DATA
BEGIN
    INSERT INTO locations
        (location,address1,address2,zipcode)
    VALUES
        (in_location,in_address1,in_address2,zipcode);
END$$
```

This procedure works fine when the location does not already exist, as shown in the following output:

```
mysql> CALL sp_add_location('Guys place','30 Blakely Drv',
                             'Irvine CA','92618-20');
```

```
Query OK, 1 row affected, 1 warning (0.44 sec)
```

However, if we try to insert a department that already exists, MySQL raises an error:

```
mysql> CALL sp_add_location('Guys place','30 Blakely Drv',
                             'Irvine CA','92618-20');
```

```
ERROR 1062 (23000): Duplicate entry 'Guys place' for key 1
```

If the stored procedure is called by an external program such as PHP, we could *probably* get away with leaving this program “as is.” PHP, and other external programs, can detect such error conditions and then take appropriate action. If the stored procedure is called from another stored procedure, however, we risk causing the entire procedure call stack to abort. That may not be what we want.

Since we can anticipate that MySQL error 1062 could be raised by this procedure, we can write code to handle that specific error code. Example 6-2 demonstrates this technique. Rather than allow the exception to propagate out of the procedure unhandled (causing failures in the calling program), the stored procedure traps the exception, sets a status flag, and returns that status information to the calling program.

The calling program can then decide if this failure warrants termination or if it should continue execution.

Example 6-2. Simple stored procedure with error handling

```
CREATE PROCEDURE sp_add_location
    (in_location    VARCHAR(30),
     in_address1    VARCHAR(30),
     in_address2    VARCHAR(30),
     zipcode        VARCHAR(10),
     OUT out_status VARCHAR(30))
    MODIFIES SQL DATA
BEGIN
    DECLARE CONTINUE HANDLER FOR 1062
        SET out_status='Duplicate Entry';

    SET out_status='OK';
    INSERT INTO locations
        (location,address1,address2,zipcode)
    VALUES
        (in_location,in_address1,in_address2,zipcode);
END;
```

We'll review in detail the syntax of the HANDLER clause later in this chapter. For now, it is enough to understand that the DECLARE CONTINUE HANDLER statement tells MySQL that “if you encounter MySQL error 1062 (duplicate entry for key), then *continue* execution but set the variable `p_status` to 'Duplicate Entry'.”

As expected, this implementation does not return an error to the calling program, and we can examine the status variable to see if the stored procedure execution was successful. In Example 6-3 we show a stored procedure that creates new department records. This procedure calls our previous procedure to add a new location. If the location already exists, the stored procedure generates a warning and continues. Without the exception handling in `sp_add_location`, this procedure would terminate when the unhandled exception is raised.

Example 6-3. Calling a stored procedure that has an error handler

```
CREATE PROCEDURE sp_add_department
    (in_department_name VARCHAR(30),
     in_manager_id       INT,
     in_location         VARCHAR(30),
     in_address1         VARCHAR(30),
     in_address2         VARCHAR(30),
     in_zipcode          VARCHAR(10)
    )
    MODIFIES SQL DATA
BEGIN
    DECLARE l_status VARCHAR(20);

    CALL sp_add_location(in_location,in_address1,in_address2,
                        in_zipcode, l_status);
    IF l_status='Duplicate Entry' THEN
        SELECT CONCAT('Warning: using existing definition for location ',
                    in_location) AS warning;
    END IF;
END;
```

Example 6-3. Calling a stored procedure that has an error handler (continued)

```
END IF;

INSERT INTO departments (manager_id,department_name,location)
VALUES(in_manager_id,in_department_name,in_location);

END;
```

Handling Last Row Conditions

One of the most common operations in a MySQL stored program involves fetching one or more rows of data. You can do this in a stored program through the use of a cursor (explained in Chapter 5). However, MySQL (and the ANSI standard) considers an attempt to fetch past the last row of the cursor an error. Therefore, you almost always need to catch that particular error when looping through the results from a cursor.

Consider the simple cursor loop shown in Example 6-4. At first glance, you might worry that we might inadvertently have created an infinite loop, since we have not coded any way to leave the `dept_loop` loop.

Example 6-4. Cursor loop without a NOT FOUND handler

```
CREATE PROCEDURE sp_fetch_forever()
READS SQL DATA
BEGIN
    DECLARE l_dept_id INT;
    DECLARE c_dept CURSOR FOR
        SELECT department_id
        FROM departments;

    OPEN c_dept;
    dept_cursor: LOOP
        FETCH c_dept INTO l_dept_id;
    END LOOP dept_cursor;
    CLOSE c_dept;
END
```

Bravely, we run this program and find that the seemingly infinite loop fails as soon as we attempt to fetch beyond the final row in the result set:

```
mysql> CALL sp_fetch_forever();
ERROR 1329 (02000): No data to FETCH
```

Since we likely want to do something with the data after we've fetched it, we cannot let this exception propagate out of the procedure unhandled. So we will add a declaration for a `CONTINUE HANDLER` in the procedure, setting a flag to indicate that the last row has been fetched. This technique is shown in Example 6-5.

Example 6-5. Cursor loop with a NOT FOUND handler

```
1 CREATE PROCEDURE sp_not_found()
2     READS SQL DATA
3 BEGIN
4     DECLARE l_last_row INT DEFAULT 0;
5     DECLARE l_dept_id INT;
6     DECLARE c_dept CURSOR FOR
7         SELECT department_id
8             FROM departments;
9     /* handler to set l_last_row=1 if a cursor returns no more rows */
10    DECLARE CONTINUE HANDLER FOR NOT FOUND SET l_last_row=1;
11
12    OPEN c_dept;
13    dept_cursor: LOOP
14        FETCH c_dept INTO l_dept_id;
15        IF (l_last_row=1) THEN
16            LEAVE dept_cursor;
17        END IF;
18        /* Do something with the data*/
19
20    END LOOP dept_cursor;
21    CLOSE c_dept;
22
23 END;
```

In plain English, the handler on line 10 says “When a fetch from a cursor returns no more rows, continue execution, but set the variable `l_last_row` to 1.” After retrieving each row, we check the `l_last_row` variable and exit from the cursor loop if the last row is returned. Without this handler, our cursor loop will fetch too many times and raise an exception.

Now that you have seen two simple examples of declaring handlers for error situations that you can anticipate, let’s explore this functionality in more detail.

Condition Handlers

A *condition handler* defines the actions that the stored program is to take when a specified event—such as a warning or an error—occurs.

Here is the syntax of the `DECLARE HANDLER` command:

```
DECLARE {CONTINUE | EXIT} HANDLER FOR
    {SQLSTATE sqlstate_code | MySQL error code | condition_name}
    handler_actions
```

Note that handlers must be defined after any variable or cursor declarations, which makes sense, since the handlers frequently access local variables or perform actions on cursors (such as closing them). They must also be declared before any executable statements. Chapter 4 includes more details on the rules governing the positioning of statements within a block.

The handler declaration has three main clauses;

- Handler type (CONTINUE, EXIT)
- Handler condition (SQLSTATE, MySQL error code, named condition)
- Handler actions

Let's look at each of these clauses in turn.

Types of Handlers

Condition handlers can be one of two types:

EXIT

When an EXIT handler fires, the currently executing block is terminated. If this block is the main block for the stored program, the procedure terminates, and control is returned to the procedure or external program that invoked the procedure. If the block is enclosed within an outer block inside of the same stored program, control is returned to that outer block.

CONTINUE

With a CONTINUE handler, execution continues with the statement following the one that caused the error to occur.

In either case, any statements defined within the handler (the *handler actions*) are run before either the EXIT or CONTINUE takes place.

Let's look at examples of both types of handlers. Example 6-6 shows a stored procedure that creates a department record and attempts to gracefully handle the situation in which the specified department already exists.

Example 6-6. Example of an EXIT handler

```
1 CREATE PROCEDURE add_department
2     (in_dept_name VARCHAR(30),
3     in_location VARCHAR(30),
4     in_manager_id INT)
5     MODIFIES SQL DATA
6 BEGIN
7     DECLARE duplicate_key INT DEFAULT 0;
8     BEGIN
9         DECLARE EXIT HANDLER FOR 1062 /* Duplicate key*/ SET duplicate_key=1;
10
11         INSERT INTO departments (department_name,location,manager_id)
12         VALUES(in_dept_name,in_location,in_manager_id);
13
14         SELECT CONCAT('Department ',in_dept_name,' created') as "Result";
15     END;
16
17     IF duplicate_key=1 THEN
18         SELECT CONCAT('Failed to insert ',in_dept_name,
```

Example 6-6. Example of an EXIT handler (continued)

```
19             ': duplicate key') as "Result";
20     END IF;
21 END$$
```

Let's examine the logic for Example 6-6:

Lines(s)	Explanation
7	Declare a status variable that will record the status of our insert attempt.
8-15	This BEGIN-END block encloses the INSERT statement that will attempt to create the department row. The block includes the EXIT handler that will terminate the block if a 1062 error occurs.
11	Attempt to insert our row—if we get a duplicate key error, the handler will set the variable and terminate the block.
14	This line executes only if the EXIT handler did not fire, and reports success to the user. If the handler fired, then the block was terminated and this line would never be executed.
17	Execution will then continue on this line, where we check the value of the variable and—if the handler has fired—advise the user that the insert was unsuccessful.

Following is the output from this stored procedure for both unsuccessful and successful execution:

```
MySQL> CALL add_department('OPTIMIZER RESEARCH','SEATTLE',4) //
```

```
+-----+
| Result                               |
+-----+
| Failed to insert OPTIMIZER RESEARCH: duplicate key |
+-----+
1 row in set (0.02 sec)
```

```
MySQL> CALL add_department('CUSTOMER SATISFACTION','DAVIS',4);
```

```
+-----+
| Result                               |
+-----+
| Department CUSTOMER SATISFACTION created |
+-----+
1 row in set (0.00 sec)
```

Example 6-7 provides an example of the same functionality implemented with a CONTINUE handler. In this example, when the handler fires, execution continues with the statement immediately following the INSERT statement. This IF statement checks to see if the handler has fired, and if it has, it displays the failure message. Otherwise, the success message is displayed.

Example 6-7. Example of a CONTINUE handler

```
CREATE PROCEDURE add_department
    (in_dept_name VARCHAR(30),
     in_location VARCHAR(30),
```

Example 6-7. Example of a CONTINUE handler (continued)

```
        in_manager_id INT)
MODIFIES SQL DATA
BEGIN
    DECLARE duplicate_key INT DEFAULT 0;

    DECLARE CONTINUE HANDLER FOR 1062 /* Duplicate key*/
        SET duplicate_key=1;

    INSERT INTO departments (department_name,location,manager_id)
    VALUES(in_dept_name,in_location,in_manager_id);

    IF duplicate_key=1 THEN
        SELECT CONCAT('Failed to insert ',in_dept_name,
            ': duplicate key') as "Result";
    ELSE
        SELECT CONCAT('Department ',in_dept_name,' created') as "Result";
    END IF;
END$$
```

EXIT or CONTINUE?

The choice between creating an EXIT handler and creating a CONTINUE handler is based primarily on program flow-of-control considerations.

An EXIT handler will exit from the block in which it is declared, which precludes the possibility that any other statements in the block (or the entire procedure) might be executed. This type of handler is most suitable for catastrophic errors that do not allow for any form of continued processing.

A CONTINUE handler allows subsequent statements to be executed. Generally, you will detect that the handler has fired (through some form of status variable set in the handler) and determine the most appropriate course of action. This type of handler is most suitable when you have some alternative processing that you will execute if the exception occurs.

Handler Conditions

The handler condition defines the circumstances under which the handler will be invoked. The circumstance is always associated with an error condition, but you have three choices as to how you define that error:

- As a MySQL error code.
- As an ANSI-standard SQLSTATE code.
- As a named condition. You may define your own named conditions (described in the later section “Named Conditions”) or use one of the built-in conditions SQLEXCEPTION, SQLWARNING, and NOT FOUND.

MySQL has its own set of error codes that are unique to the MySQL server. A handler condition that refers to a numeric code without qualification is referring to a MySQL error code. For instance, the following handler will fire when MySQL error code 1062 (duplicate key value) is encountered:

```
DECLARE CONTINUE HANDLER FOR 1062 SET duplicate_key=1;
```

SQLSTATE error codes are defined by the ANSI standard and are database-independent, meaning that they will have the same value regardless of the underlying database. So, for instance, Oracle, SQL Server, DB2, and MySQL will always report the same SQLSTATE value (23000) when a duplicate key value error is encountered. Every MySQL error code has an associated SQLSTATE code, but the relationship is not one-to-one; some SQLSTATE codes are associated with many MySQL codes; HY000 is a general-purpose SQLSTATE code that is raised for MySQL codes that have no specific associated SQLSTATE code.

The following handler will fire when SQLSTATE 23000 (duplicate key value) is encountered:

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET duplicate_key=1;
```

SQLSTATE or MySQL Error Code?

In theory, using the SQLSTATE codes will make your code more portable to other database platforms and might therefore seem to be the best choice. However, there are a number of reasons to use MySQL error codes rather than SQLSTATE codes when writing MySQL stored programs:

- In reality, it is unlikely that you will move your stored programs to another RDBMS. The Oracle and SQL Server stored program languages are totally incompatible with MySQL. The DB2 stored program language is somewhat compatible (both are based on the SQL:2003 standard). It is very likely, however, that you will use MySQL-specific syntax as you write your application, which will prevent your stored code from being portable.
- Not all MySQL error codes have SQLSTATE equivalents. Although every MySQL error code is associated with some SQLSTATE error code, often it will be a general-purpose SQLSTATE that is not specific (such as HY000). Therefore, you will almost certainly have to code some handlers that refer directly to MySQL error codes. You'll probably find that the advantages of using a consistent handler format will outweigh the theoretical portability advantage of SQLSTATE error codes.

We will, for the most part, use MySQL error codes in this book.

When the MySQL client encounters an error, it will report both the MySQL error code and the associated SQLSTATE code, as in the following output:

```
mysql> CALL nosuch_sp();
```

```
ERROR 1305 (42000): PROCEDURE sqltune.nosuch_sp does not exist
```

In this case, the MySQL error code is 1305 and the SQLSTATE code is 42000.

Table 6-1 lists some of the error codes you might expect to encounter in a MySQL stored program together with their SQLSTATE equivalents. Note, again, that many MySQL error codes map to the same SQLSTATE code (many map to HY000, for instance), which is why you may wish to sacrifice portability and use MySQL error codes—rather than SQLSTATE codes—in your error handlers.

Table 6-1. Some common MySQL error codes and SQLSTATE codes

MySQL error code	SQLSTATE code	Error message
1011	HY000	Error on delete of '%s' (errno: %d)
1021	HY000	Disk full (%s); waiting for someone to free some space...
1022	23000	Can't write; duplicate key in table '%s'
1027	HY000	'%s' is locked against change
1036	HY000	Table '%s' is read only
1048	23000	Column '%s' cannot be null
1062	23000	Duplicate entry '%s' for key %d
1099	HY000	Table '%s' was locked with a READ lock and can't be updated
1100	HY000	Table '%s' was not locked with LOCK TABLES
1104	42000	The SELECT would examine more than MAX_JOIN_SIZE rows; check your WHERE and use SET SQL_BIG_SELECTS=1 or SET SQL_MAX_JOIN_SIZE=# if the SELECT is okay
1106	42000	Incorrect parameters to procedure '%s'
1114	HY000	The table '%s' is full
1150	HY000	Delayed insert thread couldn't get requested lock for table %s
1165	HY000	INSERT DELAYED can't be used with table '%s' because it is locked with LOCK TABLES
1242	21000	Subquery returns more than 1 row
1263	22004	Column set to default value; NULL supplied to NOT NULL column '%s' at row %ld
1264	22003	Out of range value adjusted for column '%s' at row %ld
1265	1000	Data truncated for column '%s' at row %ld
1312	0A000	SELECT in a stored program must have INTO
1317	70100	Query execution was interrupted
1319	42000	Undefined CONDITION: %s
1325	24000	Cursor is already open

Table 6-1. Some common MySQL error codes and SQLSTATE codes (continued)

MySQL error code	SQLSTATE code	Error message
1326	24000	Cursor is not open
1328	HY000	Incorrect number of FETCH variables
1329	2000	No data to FETCH
1336	42000	USE is not allowed in a stored program
1337	42000	Variable or condition declaration after cursor or handler declaration
1338	42000	Cursor declaration after handler declaration
1339	20000	Case not found for CASE statement
1348	HY000	Column '%s' is not updatable
1357	HY000	Can't drop a %s from within another stored routine
1358	HY000	GOTO is not allowed in a stored program handler
1362	HY000	Updating of %s row is not allowed in %s trigger
1363	HY000	There is no %s row in %s trigger

You can find a complete and up-to-date list of error codes in Appendix B of the MySQL reference manual, available online at <http://dev.mysql.com/doc/>.

Handler Examples

Here are some examples of handler declarations:

- If any error condition arises (other than a NOT FOUND), continue execution after setting `l_error=1`:

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SET l_error=1;
```

- If any error condition arises (other than a NOT FOUND), exit the current block or stored program after issuing a ROLLBACK statement and issuing an error message:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    ROLLBACK;
    SELECT 'Error occurred - terminating';
END;
```

- If MySQL error 1062 (duplicate key value) is encountered, continue execution after executing the SELECT statement (which generates a message for the calling program):

```
DECLARE CONTINUE HANDLER FOR 1062
SELECT 'Duplicate key in index';
```

- If SQLSTATE 23000 (duplicate key value) is encountered, continue execution after executing the SELECT statement (which generates a message for the calling program):

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000'
SELECT 'Duplicate key in index';
```

- When a cursor fetch or SQL retrieves no values, continue execution after setting `l_done=1`:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET l_done=1;
```

- Same as the previous example, except specified using a `SQLSTATE` variable rather than a named condition:

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
    SET l_done=1;
```

- Same as the previous two examples, except specified using a MySQL error code variable rather than a named condition or `SQLSTATE` variable:

```
DECLARE CONTINUE HANDLER FOR 1329
    SET l_done=1;
```

Handler Precedence

As we've described, MySQL lets you define handler conditions in terms of a MySQL error code, a `SQLSTATE` error, or a named condition such as `SQLWARNING`. It is possible, therefore, that you could define several handlers in a stored program that would *all* be eligible to fire when a specific error occurred. Yet only one handler can fire in response to an error, and MySQL has clearly defined rules that determine the precedence of handlers in such a situation.

To understand the problem, consider the code fragment in Example 6-8. We have declared three different handlers, each of which would be eligible to execute if a duplicate key value error occurs. Which handler will execute? The answer is that the *most specific* handler will execute.

Example 6-8. Overlapping condition handlers

```
DECLARE EXIT HANDLER FOR 1062 SELECT 'MySQL error 1062 encountered';
DECLARE EXIT HANDLER FOR SQLWARNING SELECT 'SQLWarning encountered';
DECLARE EXIT HANDLER FOR SQLSTATE '23000' SELECT 'SQLSTATE 23000';
```

```
INSERT INTO departments VALUES (1, 'Department of Fred',22,'House of Fred');
```

Handlers based on MySQL error codes are the most specific type of handler, since an error condition will always correspond to a single MySQL error code. `SQLSTATE` codes can sometimes map to many MySQL error codes, so they are less specific. General conditions such as `SQLWARNING` and `SQLWARNING` are not at all specific. Therefore, a MySQL error code takes precedence over a `SQLSTATE` exception, which, in turn, takes precedence over a `SQLWARNING` condition.



If multiple exception handlers are eligible to fire upon an error, the most specific handler will be invoked. This means that a MySQL error code handler fires before a `SQLSTATE` handler, which, in turn, fires before a `SQLWARNING` handler.

This strictly defined precedence allows us to define a general-purpose handler for unexpected conditions, while creating a specific handler for those circumstances that we can easily anticipate. So, for instance, in Example 6-9, the first handler will be invoked if something catastrophic happens (perhaps a jealous colleague drops your database tables), while the second will fire in the more likely event that someone tries to create a duplicate row within your database.

Example 6-9. Example of overlapping condition handling

```
DECLARE EXIT HANDLER FOR 1062
    SELECT 'Attempt to create a duplicate entry occurred';
DECLARE EXIT HANDLER FOR SQLEXCEPTION
    SELECT 'Unexpected error occurred -
           make sure Fred did not drop your tables again';
```

Note, however, that we generally don't advise creating `SQLEXCEPTION` handlers until MySQL implements the `SIGNAL` statement; see “Missing SQL:2003 Features” later in this chapter.

Scope of Condition Handlers

The *scope* of a handler determines which statements within the stored program are covered by the handler. In essence, the scope of a handler is the same as for a stored program variable: the handler applies to all statements in the block in which it is defined, including any statements in nested blocks. Furthermore, handlers in a stored program also cover statements that execute in any stored program that might be called by the first program, unless that program declares its own handler.

For instance, in Example 6-10 the handler will be invoked when the `INSERT` statement executes (because it violates a `NOT NULL` constraint). The handler fires because the `INSERT` statement is contained within the same block as the handler—even though the `INSERT` statement is in a nested block.

Example 6-10. Handler scope includes statements within BEGIN-END blocks

```
DECLARE CONTINUE HANDLER FOR 1048 SELECT 'Attempt to insert a null value';
BEGIN
    INSERT INTO departments (department_name,manager_id,location)
        VALUES (NULL,1,'Wouldn't you like to know?');
END;
```

However, in Example 6-11 the handler will not be invoked—the scope of the handler is limited to the nested block, and the `INSERT` statement occurs outside that block.

Example 6-11. Handlers within a nested block do not cover statements in enclosing blocks

```
BEGIN
  BEGIN
    DECLARE CONTINUE HANDLER FOR 1216 select
      'Foreign key constraint violated';
  END;
  INSERT INTO departments (department_name,manager_id,location)
    VALUES ('Elbonian HR','Catbert','Catbertia');
END;
```

Handler scope extends to any stored procedures or functions that are invoked within the handler scope. This means that if one stored program calls another, a handler in the calling program can trap errors that occur in the program that has been called. So, for instance, in Example 6-12, the handler in `calling_procedure()` traps the null value exception that occurs in `sub_procedure()`.

Example 6-12. A handler can catch conditions raised in called procedures

```
CREATE PROCEDURE calling_procedure()
BEGIN
  DECLARE EXIT HANDLER FOR 1048 SELECT 'Attempt to insert a null value';
  CALL sub_procedure();
END;
```

Query OK, 0 rows affected (0.00 sec)

```
-----
CREATE PROCEDURE sub_procedure()
BEGIN
  INSERT INTO departments (department_name,manager_id,location)
    VALUES (NULL,1,'Wouldn't you like to know');
  SELECT 'Row inserted';
END;
```

Query OK, 0 rows affected (0.00 sec)

```
CALL calling_procedure();
```

```
+-----+
| Attempt to insert a null value |
+-----+
| Attempt to insert a null value |
+-----+
1 row in set (0.01 sec)
```

Query OK, 0 rows affected (0.01 sec)

Of course, a handler in a procedure will override the scope of a handler that exists in a calling procedure. Only one handler can ever be activated in response to a specific error condition.

Named Conditions

So far, our examples have used conditions based on MySQL error codes, `SQLSTATE` codes, or predefined named conditions (`SQLException`, `SQLWarning`, `NOT FOUND`). These handlers do the job required, but they do not result in particularly readable code, since they rely on the hardcoding of literal error numbers. Unless you memorize all or most of the MySQL error codes and `SQLSTATE` codes (and expect everyone maintaining your code to do the same), you are going to have to refer to a manual to understand exactly what error a handler is trying to catch.

You can improve the readability of your handlers by defining a condition declaration, which associates a MySQL error code or `SQLSTATE` code with a meaningful name that you can then use in your handler declarations. The syntax for a condition declaration is:

```
DECLARE condition_name CONDITION FOR {SQLSTATE sqlstate_code | MySQL_error_code};
```

Once we have declared our condition name, we can use it in our code instead of a MySQL error code or `SQLSTATE` code. So instead of the following declaration:

```
DECLARE CONTINUE HANDLER FOR 1216 MySQL_statements;
```

we could use the following more readable declaration:

```
DECLARE foreign_key_error CONDITION FOR 1216;
```

```
DECLARE CONTINUE HANDLER FOR foreign_key_error MySQL_statements;
```



Create named conditions using condition declarations, and use these named conditions in your handlers to improve the readability and maintainability of your stored program code.

Missing SQL:2003 Features

The SQL:2003 specification includes a few useful features that—at the time of writing—are not currently implemented in the MySQL stored program language. The absence of these features certainly limits your ability to handle unexpected conditions, but we expect that they will be implemented in MySQL server 5.2. Specifically:

- There is no way to examine the current MySQL error code or `SQLSTATE` code. This means that in an exception handler based on a generic condition such as `SQLException`, you have no way of knowing what error just occurred.
- You cannot raise an exception of your own to indicate an application-specific error or to re-signal an exception after first catching the exception and examining its context.

We'll describe these situations in the following sections and suggest ways to deal with them.

Directly Accessing SQLCODE or SQLSTATE

Implementing a general-purpose exception handler would be a good practice, except that if you cannot reveal the reason why the exception occurred, you make debugging your stored programs difficult or impossible. For instance, consider Example 6-13.

Example 6-13. General-purpose—but mostly useless—condition handler

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
    SET l_status=-1;
    SET l_message='Some sort of error detected somewhere in the application';
END;
```

Receiving an error message like this is not much help—in fact, there is almost nothing more frustrating than receiving such an error message when trying to debug an application. Obscuring the actual cause of the error makes the condition handler worse than useless in most circumstances.

The SQL:2003 specification allows for direct access to the values of SQLCODE (the “vendor”—in this case MySQL—error code) and the SQLSTATE code. If we had access to these codes, we could produce a far more helpful message such as shown in Example 6-14.

Example 6-14. A more useful—but not supported—form of condition handler

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
    SET l_status=-1;
    SET l_message='Error '||sqlcode||' encountered';
END;
```

We can partially emulate the existence of a SQLCODE or SQLSTATE variable by defining a more comprehensive set of condition handlers that create appropriate SQLCODE variables when they are fired. The general approach would look like Example 6-15.

Example 6-15. Using multiple condition handlers to expose the actual error code

```
DECLARE sqlcode INT DEFAULT 0;
DECLARE status_message VARCHAR(50);

DECLARE CONTINUE HANDLER FOR duplicate_key
BEGIN
    SET sqlcode=1052;
    SET status_message='Duplicate key error';
END;

DECLARE CONTINUE HANDLER FOR foreign_key_violated
BEGIN
    SET sqlcode=1216;
```

Example 6-15. Using multiple condition handlers to expose the actual error code (continued)

```
    SET status_message='Foreign key violated';
END;

DECLARE CONTINUE HANDLER FOR NOT FOUND
BEGIN
    SET sqlcode=1329;
    SET status_message='No record found';
END;
```

In most circumstances, it is best not to define a `SQLEXCEPTION` handler, because without the ability to display the `SQLSTATE` or `SQLSTATE`, it is better to let the exception occur and allow the calling application to have full access to the error codes and messages concerned.



Until MySQL implements a `SQLSTATE` or `SQLSTATE` variable, avoid creating a general-purpose `SQLEXCEPTION` handler. Instead, create handlers for individual error conditions that generate appropriate messages and status codes.

Creating Your Own Exceptions with the `SIGNAL` Statement

So far in this chapter, we have talked about how you can handle errors raised by MySQL as it executes SQL statements within the stored program. In addition to these system-raised exceptions, however, you will surely have to deal with errors that are specific to an application's domain of requirements and rules. If that rule is violated in your code, you may want to raise your *own* error and communicate this problem back to the user. The SQL:2003 specification provides the `SIGNAL` statement for this purpose.

The `SIGNAL` statement allows you to raise your own error conditions. Unfortunately, at the time of writing, the `SIGNAL` statement is not implemented within the MySQL stored program language (it is currently scheduled for MySQL 5.2).

You can't use the `SIGNAL` statement in MySQL 5.0, but we are going to describe it here, in case you are using a later version of MySQL in which the statement has been implemented. Visit this book's web site (see the Preface for details) to check on the status of this and other enhancements to the MySQL stored program language.

So let's say that we are creating a stored procedure to process employee date-of-birth changes, as shown in Example 6-16. Our company never employs people under the age of 16, so we put a check in the stored procedure to ensure that the updated date of birth is more than 16 years ago (the `curdate()` function returns the current timestamp).

Example 6-16. Example stored procedure with date-of-birth validation

```
CREATE PROCEDURE sp_update_employee_dob
    (p_employee_id INT, p_dob DATE, OUT p_status varchar(30))
BEGIN
    IF DATE_SUB(curdate(), INTERVAL 16 YEAR) <p_dob THEN
        SET p_status='Employee must be 16 years or older';
    ELSE
        UPDATE employees
            SET date_of_birth=p_dob
            WHERE employee_id=p_employee_id;
        SET p_status='Ok';
    END IF;
END;
```

This implementation will work, but it has a few disadvantages. The most significant problem is that if the procedure is called from another program, the procedure will return success (at least, it will not raise an error) even if the update was actually rejected. Of course, the calling program could detect this by examining the `p_status` variable, but there is a good chance that the program will assume that the procedure succeeded since the procedure call itself does not raise an exception.

We have designed the procedure so that it depends on the diligence of the programmer calling the procedure to check the value of the returning status argument. It is all too tempting and easy to assume that everything went fine, since there was no error.

To illustrate, if we try to set an employee's date of birth to the current date from the MySQL command line, everything seems OK:

```
mysql> CALL sp_update_employee_dob(1,now(),@status);
Query OK, 0 rows affected (0.01 sec)
```

It is only if we examine the status variable that we realize that the update did not complete:

```
mysql> SELECT @status;
+-----+
| @status |
+-----+
| Employee must be 16 years or older |
+-----+
1 row in set (0.00 sec)
```

This stored procedure would be more robust, and less likely to allow errors to slip by, if it actually raised an error condition when the date of birth was invalid. The ANSI SQL:2003 SIGNAL statement allows you to do this:

SIGNAL takes the following form:

```
SIGNAL SQLSTATE sqlstate_code|condition_name [SET MESSAGE_TEXT=string_or_variable];
```

You can create your own SQLSTATE codes (there are some rules for the numbers you are allowed to use) or use an existing SQLSTATE code or named condition. When

MySQL implements SIGNAL, you will probably be allowed to use a MySQL error code (within designated ranges) as well.

When the SIGNAL statement is executed, a database error condition is raised that acts in exactly the same way as an error that might be raised by an invalid SQL statement or a constraint violation. This error could be returned to the calling program or could be trapped by a handler in this or another stored program. If SIGNAL were available to us, we might write the employee date-of-birth birth procedure, as shown in Example 6-17.

Example 6-17. Using the SIGNAL statement (expected to be implemented in MySQL 5.2)

```
CREATE PROCEDURE sp_update_employee_dob
  (p_employee_id int, p_dob date)
BEGIN
  DECLARE employee_is_too_young CONDITION FOR SQLSTATE '99001';

  IF DATE_SUB(curdate(), INTERVAL 16 YEAR) <P_DOB THEN
    SIGNAL employee_is_too_young
      SET MESSAGE_TEST='Employee must be 16 years or older';
  ELSE
    UPDATE employees
      SET date_of_birth=p_dob
      WHERE employee_id=p_employee_id;
  END IF;
END;
```

If we ran this new procedure from the MySQL command line (when MySQL implements SIGNAL), we would expect the following output:

```
mysql> CALL sp_update_employee(1,now());
ERROR 90001 (99001): Employee must be 16 years or older
```

Using SIGNAL, we could make it completely obvious to the user or calling program that the stored program execution failed.

Emulating the SIGNAL Statement

The absence of the SIGNAL statement makes some stored program logic awkward, and in some cases demands that calling applications examine OUT variables, rather than SQL return codes, to check the results of some operations.

There is, however, a way to force an error to occur and pass some diagnostic information back to the calling application. You can, in other words, emulate SIGNAL in MySQL 5.0, but we warn you: this solution is not pretty!

Where we would otherwise want to use the SIGNAL statement to return an error to the calling application, we can instead issue a SQL statement that will fail—and fail in such a way that we can embed our error message within the standard error message.

The best way to do this is to issue a SQL statement that attempts to reference a non-existent table or column. The name of the nonexistent column or table can include the error message itself, which will be useful because the name of the column or table is included in the error message.

Example 6-18 shows how we can do this. We try to select a nonexistent column name from a table and we make the nonexistent column name comprise our error message. Note that in order for a string to be interpreted as a column name, it must be enclosed by backquotes (these are the quote characters normally found on your keyboard to the left of the 1 key).

Example 6-18. Using a nonexistent column name to force an error to the calling program

```
CREATE PROCEDURE sp_update_employee_dob2
    (p_employee_id INT, p_dob DATE)
BEGIN
    IF datediff(curdate(),p_dob)<(16*365) THEN
        UPDATE `Error: employee_is_too_young; Employee must be 16 years or older`
            SET x=1;
    ELSE
        UPDATE employees
            SET date_of_birth=p_dob
            WHERE employee_id=p_dob;
    END IF;
END;
```

If we try to run the stored procedure from the MySQL command line, passing in an invalid date of birth, we get a somewhat informative error message:

```
MySQL> CALL sp_update_employee_dob2(2,now()) ;

ERROR 1054 (42S22): Unknown column 'Error: employee_is_too_young; Employee must be 16
years or older' in 'field list'
```

The error code is somewhat garbled, and the error code is not in itself accurate, but at least we have managed to signal to the calling application that the procedure did not execute successfully and we have at least provided some helpful information.

We can somewhat improve the reliability of our error handling—and also prepare for a future in which the SIGNAL statement is implemented—by creating a generic procedure to implement our SIGNAL workaround. Example 6-19 shows a procedure that accepts an error message and then constructs dynamic SQL that includes that message within an invalid table name error.

Example 6-19. Standard procedure to emulate SIGNAL

```
CREATE PROCEDURE `my_signal`(in_errortext VARCHAR(255))
BEGIN
    SET @sql=CONCAT('UPDATE `',
        in_errortext,
        '` SET x=1');
END;
```

Example 6-19. Standard procedure to emulate SIGNAL (continued)

```
PREPARE my_signal_stmt FROM @sql;
EXECUTE my_signal_stmt;
DEALLOCATE PREPARE my_signal_stmt;
END$$
```

We could now implement our employee date-of-birth update routine to call this routine, as shown in Example 6-20.

Example 6-20. Using our SIGNAL emulation procedure to raise an error

```
CREATE PROCEDURE sp_update_employee_dob2(p_employee_id INT, p_dob DATE)
BEGIN
    IF datediff(curdate(),p_dob)<(16*365) THEN
        CALL my_signal('Error: employee_is_too_young; Employee must be 16
            years or older');
    ELSE
        UPDATE employees
            SET date_of_birth=p_dob
            WHERE employee_id=p_employee_id;
    END IF;
END$$
```

Not only does this routine result in cleaner code that is easier to maintain, but when MySQL does implement SIGNAL, we will only need to update our code in a single procedure.

Putting It All Together

We have now covered in detail the error-handling features of MySQL. We'll finish up this discussion by offering an example that puts all of these features together. We will take a simple stored procedure that contains no exception handling and apply the concepts from this chapter to ensure that it will not raise any unhandled exceptions for all problems that we can reasonably anticipate.

The example stored procedure creates a new departments row. It takes the names of the new department, the manager of the department, and the department's location. It retrieves the appropriate `employee_id` from the `employees` table using the manager's name. Example 6-21 shows the version of the stored procedure without exception handling.

Example 6-21. Stored procedure without error handling

```
CREATE PROCEDURE sp_add_department
(p_department_name    VARCHAR(30),
 p_manager_surname   VARCHAR(30),
 p_manager_firstname VARCHAR(30),
```

Example 6-21. Stored procedure without error handling (continued)

```
    p_location          VARCHAR(30),
    out p_sqlcode       INT,
    out p_status_message VARCHAR(100))
MODIFIES SQL DATA
BEGIN

    DECLARE l_manager_id INT;
    DECLARE csr_mgr_id cursor for
        SELECT employee_id
           FROM employees
          WHERE surname=UPPER(p_manager_surname)
             AND firstname=UPPER(p_manager_firstname);

    OPEN csr_mgr_id;
    FETCH csr_mgr_id INTO l_manager_id;

    INSERT INTO departments (department_name,manager_id,location)
    VALUES(UPPER(p_department_name),l_manager_id,UPPER(p_location));

    CLOSE csr_mgr_id;
END$$
```

This program reflects the typical development process for many of us: we concentrate on implementing the required functionality (the “positive”) and generally pay little attention to (or more likely, want to avoid thinking about) what could possibly go wrong. The end result is a stored program that contains no error handling.

So either before you write the program (ideally) or after the first iteration is done, you should sit down and list out all the errors that might be raised by MySQL when the program is run.

Here are several of the failure points of this stored procedure:

- If the manager’s name is incorrect, we will fail to find a matching manager in the employees table. We will then attempt to insert a NULL value for the `MANAGER_ID` column, which will violate its NOT NULL constraint.
- If the location argument does not match a location in the locations table, the foreign key constraint between the two tables will be violated.
- If we specify a `department_name` that already exists, we will violate the unique constraint on the `department_name`.

The code in Example 6-22 demonstrates these failure scenarios.

Example 6-22. Some of the errors generated by a stored procedure without error handling

```
mysql> CALL sp_add_department
      ('Optimizer Research','Yan','Bianca','Berkshire',@p_sqlcode,@p_status_message)
```

ERROR 1062 (23000): Duplicate entry 'OPTIMIZER RESEARCH' for key 2

Example 6-22. Some of the errors generated by a stored procedure without error handling (continued)

```
mysql> CALL sp_add_department
('Optimizer Research','Yan','Binca','Berkshire',@p_sqlcode,@p_status_message);
```

ERROR 1048 (23000): Column 'MANAGER_ID' cannot be null

```
mysql> CALL sp_add_department('Advanced Research','Yan','Bianca','Bercshire',@p_
sqlcode,@p_status_message)
```

ERROR 1216 (23000): Cannot add or update a child row: a foreign key constraint fails

The good news is that MySQL detects these problems and will not allow bad data to be placed into the table. If this stored procedure will be called only by the host language, such as PHP or Java, we could declare ourselves done. If, on the other hand, this program might be called from another MySQL stored program, then we need to handle the errors and return status information so that the calling stored program can take appropriate action. Example 6-23 shows a version of the stored procedure that handles all the errors shown in Example 6-22.

Example 6-23. Stored procedure with error handling

```
1 CREATE PROCEDURE sp_add_department2
2     (p_department_name      VARCHAR(30),
3     p_manager_surname      VARCHAR(30),
4     p_manager_firstname    VARCHAR(30),
5     p_location             VARCHAR(30),
6     OUT p_sqlcode          INT,
7     OUT p_status_message   VARCHAR(100))
8 BEGIN
9
10 /* START Declare Conditions */
11
12 DECLARE duplicate_key CONDITION FOR 1062;
13 DECLARE foreign_key_violated CONDITION FOR 1216;
14
15 /* END Declare Conditions */
16
17 /* START Declare variables and cursors */
18
19 DECLARE l_manager_id      INT;
20
21 DECLARE csr_mgr_id CURSOR FOR
22     SELECT employee_id
23     FROM employees
24     WHERE surname=UPPER(p_manager_surname)
25     AND firstname=UPPER(p_manager_firstname);
26
27 /* END Declare variables and cursors */
28
29 /* START Declare Exception Handlers */
30
31 DECLARE CONTINUE HANDLER FOR duplicate_key
```

Example 6-23. Stored procedure with error handling (continued)

```
32 BEGIN
33     SET p_sqlcode=1052;
34     SET p_status_message='Duplicate key error';
35 END;
36
37 DECLARE CONTINUE HANDLER FOR foreign_key_violated
38 BEGIN
39     SET p_sqlcode=1216;
40     SET p_status_message='Foreign key violated';
41 END;
42
43 DECLARE CONTINUE HANDLER FOR not FOUND
44 BEGIN
45     SET p_sqlcode=1329;
46     SET p_status_message='No record found';
47 END;
48
49 /* END Declare Exception Handlers */
50
51 /* START Execution */
52
53 SET p_sqlcode=0;
54 OPEN csr_mgr_id;
55 FETCH csr_mgr_id INTO l_manager_id;
56
57 IF p_sqlcode<>0 THEN /* Failed to get manager id*/
58     SET p_status_message=CONCAT(p_status_message,' when fetching manager id');
59 ELSE
60     /* Got manager id, we can try and insert */
61     INSERT INTO departments (department_name,manager_id,location)
62     VALUES(UPPER(p_department_name),l_manager_id,UPPER(p_location));
63     IF p_sqlcode<>0 THEN/* Failed to insert new department */
64         SET p_status_message=CONCAT(p_status_message,
65         ' when inserting new department');
66     END IF;
67 END IF;
68
69 CLOSE csr_mgr_id;
70
71 / * END Execution */
72
73 END
```

Let's go through Example 6-23 and review the error-handling code we have added.

Line(s)	Significance
12 and 13	Create condition declarations for duplicate key (1062) and foreign key (1216) errors. As we noted earlier, these declarations are not strictly necessary, but they improve the readability of the condition handlers we will declare later.
31-48	Define handlers for each of the exceptions we think might occur. The condition names match those we defined in lines 10 and 11. We didn't have to create a NOT FOUND condition, since this is a predefined condition name. Each handler sets an appropriate value for the output status variables p_sqlcode and p_status_message.

Line(s)	Significance
57	On this line we check the value of the <code>p_sqlcode</code> variable following our fetch from the cursor that retrieves the manager's <code>employee_id</code> . If <code>p_sqlcode</code> is not 0, then we know that one of our exception handlers has fired. We add some context information to the message—identifying the statement we were executing—and avoid attempting to execute the insert into the <code>departments</code> table.
53	Check the value of the <code>p_sqlcode</code> variable following our insert operation. Again, if the value is nonzero, we know that an error has occurred, and we add some context information to the error message. At line 53, we don't know what error has occurred—it could be either the foreign key or the unique index constraint. The handler itself controls the error message returned to the user, and so we could add handling for more error conditions by adding additional handlers without having to amend this section of code.

Running the stored procedure from the MySQL command line shows us that all the exceptions are now correctly handled. Example 6-24 shows the output generated by various invalid inputs.

Example 6-24. Output from stored procedure with exception handling

```
mysql> CALL sp_add_department2('Optimizer Research','Yan','Bianca','Berkshire',
@p_sqlcode,@p_status_message)
```

```
Query OK, 0 rows affected (0.17 sec)
```

```
mysql> SELECT @p_sqlcode,@p_status_message
```

```
+-----+-----+
| @p_sqlcode | @p_status_message |
+-----+-----+
| 1052      | Duplicate key error when inserting new department |
+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> CALL sp_add_department2('Optimizer Research','Yan','Binca','Berkshire',
@p_sqlcode,@p_status_message)
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @p_sqlcode,@p_status_message
```

```
+-----+-----+
| @p_sqlcode | @p_status_message |
+-----+-----+
| 1329      | No record found when fetching manager id |
+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> call sp_add_department2('Advanced Research','Yan','Bianca','Bercshire',
@p_sqlcode,@p_status_message)
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
mysql> SELECT @p_sqlcode,@p_status_message
```

Example 6-24. Output from stored procedure with exception handling (continued)

```
+-----+-----+
| @p_sqlcode | @p_status_message |
+-----+-----+
| 1216       | Foreign key violated when inserting new department |
+-----+-----+
1 row in set (0.00 sec)
```

Handling Stored Program Errors in the Calling Application

Throughout this chapter, we’ve often talked about “returning the error to the calling application.” In our examples we have used the MySQL command-line client to represent the calling application since this client is common to all environments and readers, and it allows you (and us) to quickly test out the stored program.

In production environments, however, the calling application will not be the MySQL command-line program, but possibly a web-based application using PHP, Perl, Java, Python, or .NET (C# or Visual Basic) to interface with the MySQL stored program. In Chapters 12 through 17, we look in detail at how to invoke stored programs from a variety of languages. We also cover various techniques for retrieving status and error messages from these languages. However, since we’re on the topic of error handling, let’s briefly look at how we can process errors generated by a stored program called from each of these languages.

PHP

PHP provides a variety of ways of interacting with MySQL. There are four major interfaces available:

PEAR (PHP Extension and Application Repository)

The PEAR repository includes a standard, database-independent module called PEAR DB that can be used to interface with almost any relational database.

mysql

PHP includes a MySQL-specific interface inventively called the `mysql` extension.

mysqli

Commencing with PHP 5, a new interface—`mysqli`—was introduced (according to the developer, the “i” stands for “any one of: improved, interface, ingenious, incompatible, or incomplete”). This interface provides better support for new features of MySQL.

PDO (PHP Data Objects)

PDO, a new interface with PHP 5.1, provides a PHP 5N compatible, object-oriented, database-independent interface.

The `mysqli` and PDO interfaces provide the best support for MySQL stored programs and other new features of MySQL 5.0.

In Chapter 13, we show in detail how to use stored programs with each of the major PHP interfaces and provide examples of both procedural and nonprocedural styles. For now, let's look at a simple example showing how to process errors using the object-oriented variant of the `mysqli` interface.

In Example 6-25, a simple stored procedure—one without OUT parameters or result sets—is executed on line 8. If the method call returns failure, we can examine various properties of the database connection object (`$dbh` in this example). `$dbh->errno` contains the MySQL error code, `$dbh->error` contains the error message, and `$dbh->sqlstate` contains the SQLSTATE code.

Example 6-25. Error handling in the PHP 5 `mysqli` interface

```
1 $dbh = new mysqli($hostname, $username, $password, $database);
2 /* check connection */
3 if (mysqli_connect_errno()) {
4     printf("Connect failed: %s\n", mysqli_connect_error());
5     exit();
6 }
7
8 if ($dbh->query("call error_test_proc(1)") /*execute stored procedure*/
9 {
10    printf("Stored procedure execution succeeded");
11 }
12 else // Stored procedure failed - show error
13 {
14     printf("<p>Stored procedure error: MySQL error %d (SQLSTATE %s)\n %s\n",
15           $dbh->errno,$dbh->sqlstate,$dbh->error);
16 }
```

Perl

The Perl DBI interface provides a consistent interface to various relational databases. The error-handling techniques for Perl are very similar to those of PHP.

DBI objects—such as database and statement handles—include the following properties:

Err

Contains the database-specific return code (in our case, the MySQL error code).

Errstr

Contains the full message text.

State

Contains the SQLSTATE variable. However, the SQLSTATE variable usually includes only a generic success or failure code.

Each of these items can be referenced as a method or a property, so, for instance, you can reference the last MySQL error code for the connect handle `$dbh` as either `$dbh::err` or `$dbh->err`.

Example 6-26 shows a simple Perl code fragment that executes a stored procedure and checks the error status. On line 5 we execute a simple stored procedure (one without parameters or result sets). If the stored procedure call fails, we interrogate the error methods from the database handle.

Example 6-26. Error handling in Perl DBI

```
1  $dbh = DBI->connect("DBI:mysql:$database:$host:$port",
2                      "$user", "$password",
3                      { PrintError => 0}) || die $DBI::errstr;
4
5  if ($dbh->do("call error_test_proc(1)"))
6  {
7      printf("Stored procedure execution succeeded\n");
8  }
9  else
10 {
11     printf("Error executing stored procedure: MySQL error %d (SQLSTATE %s)\n %s\n",
12           $dbh->err, $dbh->state, $dbh->errstr);
13 }
```

Java/JDBC

MySQL provides a Java JDBC 3.0 driver—MySQL Connector/J—that allows Java programs to interact with a MySQL server.

Like most modern object-oriented languages, Java uses structured exception handling to allow for flexible and efficient interception and handling of runtime errors. Rather than check the error status of every database call, we enclose our JDBC statements within a try block. If any of these statements causes a `SQLException` error, then the catch handler will be invoked to handle the error.

The catch handler has access to a `SQLException` object that provides various methods and properties for diagnosing and interpreting the error. Of most interest to us are these three methods:

```
getErrorCode()
    Returns the MySQL-specific error code
getSQLState()
    Returns the ANSI-standard SQLSTATE code
getMessage()
    Returns the full text of the error message
```

Example 6-27 shows an example of invoking a simple stored procedure that involves no OUT parameters or result sets. On line 8 we create a statement object, and on line 9

we use the execute method of that object to execute the stored procedure. If an error occurs, the catch block on line 11 is invoked, and the relevant methods of the SQLException object are used to display the details of the error.

Example 6-27. Stored procedure error handling in Java/JDBC

```
1 try {
2     Class.forName("com.mysql.jdbc.Driver").newInstance();
3
4     String connectionString="jdbc:mysql://" + hostname + "/" + database + "?user=" +
5         username + "&password=" + password;
6     System.out.println(connectionString);
7     Connection conn = DriverManager.getConnection(connectionString);
8     Statement stmt=conn.createStatement();
9     stmt.execute("call error_test_proc(1)");
10 }
11 catch(SQLException SQLEx) {
12     System.out.println("MySQL error: "+SQLEx.getErrorCode()+
13         " SQLSTATE:" +SQLEx.getSQLState());
14     System.out.println(SQLEx.getMessage());
15 }
```

Python

Python can connect to MySQL using the MySQLdb extension. This extension generates Python exceptions if any MySQL errors are raised during execution. We enclose our calls to MySQL in a try block and catch any errors in an except block.

Example 6-28 shows how we can connect to MySQL and execute a stored procedure in Python. Line 1 commences the try block, which contains our calls to MySQL. On line 2 we connect to MySQL. On line 7 we create a cursor (SQL statement handle), and on line 8 we execute a stored procedure call.

Example 6-28. Stored procedure error handling in Python

```
1     try:
2         conn = MySQLdb.connect (host = 'localhost',
3             user = 'root',
4             passwd = 'secret',
5             db = 'prod',
6             port=3306)
7         cursor1=conn.cursor()
8         cursor1.execute("CALL error_test_proc()")
9         cursor1.close()
10
11     except MySQLdb.Error, e:
12         print "Mysql Error %d: %s" % (e.args[0], e.args[1])
```

If any of these calls generates a MySQL error condition, we jump to the except block on line 11. The MySQLdb.Error object (aliased here as e) contains two elements: element 0 is the MySQL error code, and element 1 is the MySQL error message.

C# .NET

MySQL provides an ADO.NET connector—MySQL Connector/Net—that allows any .NET-compatible language to interact with a MySQL server.

In this chapter we provide a short example of handling stored procedure errors from a C# program. More details are provided in Chapter 17.

As in Java, C# provides an exception-handling model that relieves the developer of the necessity of checking for error conditions after every statement execution. Instead, commands to be executed are included within a try block. If an error occurs for any of these statements, execution switches to the catch block, in which appropriate error handling can be implemented.

Example 6-29 shows an example of error handling for a simple stored procedure (one without output parameters or result sets) in C#. A statement object for the stored procedure is created on line 15, and the statement is executed on line 17. If a `MySqlException` (essentially any MySQL error) occurs, the error handler defined on line 19 is invoked.

Example 6-29. Error handling in C#/ADO.NET

```
1  MySqlConnection myConnection;
2  myConnection = new MySqlConnection();
3  myConnection.ConnectionString = "database="+database+";server="+server+
4                                ";user id="+user+";password="+password;
5  try {
6      myConnection.Open();
7  }
8  catch (MySqlException MyException) {
9      Console.WriteLine("Connection error: MySQL code: "+MyException.Number
10                       +" "+ MyException.Message);
11 }
12
13 try {
14
15     MySqlCommand myCommand = new MySqlCommand("call error_test_proc(1)",
16                                               myConnection);
17     myCommand.ExecuteNonQuery();
18 }
19 catch (MySqlException MyException) {
20     Console.WriteLine("Stored procedure error: MySQL code: " + MyException.Number
21                       + " " + MyException.Message);
22 }
```

catch blocks have access to a `MySqlException` object; this object includes `Message` and `Number` properties, which contain the MySQL error message and error number, respectively.

Visual Basic .NET

The process for handling stored program errors in Visual Basic .NET (VB.NET) is practically identical to that of C#.

Example 6-30 shows an example of error handling for a simple stored procedure (one without output parameters or result sets) in VB.NET. A statement object for the stored procedure is created on lines 16 and 17, and the statement is executed on line 18. If a `MySqlException` (essentially any MySQL error) occurs, the error handler defined in lines 20-24 is invoked.

Example 6-30. Stored procedure error handling in VB.NET

```
1 Dim myConnectionString As String = "Database=" & myDatabase & _
2   " ;Data Source=" & myHost & _
3   ";User Id=" & myUserId & ";Password=" & myPassword
4
5 Dim myConnection As New MySqlConnection(myConnectionString)
6
7 Try
8   myConnection.Open()
9 Catch MyException As MySqlException
10    Console.WriteLine("Connection error: MySQL code: " & MyException.Number & _
11      " " + MyException.Message)
12 End Try
13
14 Try
15
16     Dim myCommand As New MySqlCommand("call error_test_proc(1)")
17     myCommand.Connection = myConnection
18     myCommand.ExecuteNonQuery()
19
20 Catch MyException As MySqlException
21     Console.WriteLine("Stored procedure error: MySQL code: " & _
22       MyException.Number & " " & _
23       MyException.Message)
24 End Try
```

Catch blocks have access to a `MySqlException` object; this object includes `Message` and `Number` properties, which contain the MySQL error message and error number, respectively.

Conclusion

In this chapter we examined the MySQL error handlers that allow you to catch error conditions and take appropriate corrective actions. Without error handlers, your stored programs will abort whenever they encounter SQL errors, returning control to the calling program. While this might be acceptable for some simple stored programs, it is more likely that you will want to trap and handle errors within the stored

program environment, especially if you plan to call one stored program from another. In addition, you need to declare handlers for cursor loops so that an error is not thrown when the last row is retrieved from the cursor.

Handlers can be constructed to catch all errors, although this is currently not best practice in MySQL, since you do not have access to an error code variable that would allow you to differentiate between possible error conditions or to report an appropriate diagnostic to the calling program. Instead, you should declare individual handlers for error conditions that can reasonably be anticipated. When an unexpected error occurs, it is best to let the stored program abort so that the calling program has access to the error codes and messages.

Handlers can be constructed that catch either ANSI-standard `SQLSTATE` codes or MySQL-specific error codes. Using the `SQLSTATE` codes leads to more portable code, but because specific `SQLSTATE` codes are not available for all MySQL error conditions, you should feel free to construct handlers against MySQL-specific error conditions.

To improve the readability of your code, you will normally want to declare named conditions against the error codes you are handling, so that the intention of your handlers is clear. It is far easier to understand a handler that traps `DUPLICATE_KEY_VALUE` than one that checks for MySQL error code 1062.

At the time of writing, some critical SQL:2003 error-handling functionality has yet to be implemented in MySQL, most notably the ability to directly access the `SQLSTATE` or `SQLCODE` variables, as well as the ability to raise an error condition using the `SIGNAL` statement. In the absence of a `SQLSTATE` or `SQLCODE` variable, it is good practice for you to define handlers against all error conditions that can reasonably be anticipated that populate a `SQLCODE`-like variable that you can use within your program code to detect errors and take appropriate action. We expect MySQL to add these “missing” features in version 5.2—you should check to see if they have been implemented in the time since this book was written (see the book’s web site for details). Note also that it is currently possible to provide a workaround (though a somewhat awkward one) for the missing `SIGNAL` statement if you find that it is absolutely necessary in your programs.