CHAPTER 4

Creating the Product Catalog: Part 1

After learning about the three-tier architecture and implementing a bit of your web site's main page, you're ready to start creating the TShirtShop product catalog.

Because the product catalog is composed of many components, you'll create it over two chapters. In this chapter, you'll create the first data table, implement access methods in the middle tier, and learn how to deal with the data tier. By the end of this chapter, you'll finally have something dynamically generated on your web page. In Chapter 5, you'll finish building the product catalog by adding support for categories, product lists, a product details page, and more!

The main topics we'll cover in this chapter are

- Analyzing the structure of the product catalog and the functionality it should support
- Creating the database structures for the catalog and the data tier of the catalog
- Implementing the business tier objects required to make the catalog run
- Implementing a functional user interface for the product catalog

Showing Your Visitors What You've Got

One of the essential features required in any e-store is to allow the visitor to easily browse through the products. Just imagine what Amazon.com would be like without its excellent product catalog!

Whether your visitors are looking for something specific or just browsing, it's important to make sure their experiences with your site are pleasant ones. After all, you want your visitors to find what they are looking for as easily and painlessly as possible. This is why you'll want to add search functionality to the site and also find a clever way of structuring products into categories so they can be quickly and intuitively accessed.

Depending on the size of the store, it might be enough to group products under a number of categories, but if there are a lot of products, you'll need to find even more ways to categorize and structure the product catalog.

Determining the structure of the catalog is one of the first tasks to accomplish in this chapter. Keep in mind that, in a professional approach, these details would have been established before starting to code when building the requirements document for the project. However, for the purposes of this book, we prefer to deal with things one at a time.

After the structure of the catalog is established, you'll start writing the code that makes the catalog work as planned.

What Does a Product Catalog Look Like?

Today's web surfers are more demanding than they used to be. They expect to find information quickly on whatever product or service they have in mind, and if they don't find it, they are likely to go to the competition before giving the site a second chance. Of course, you don't want this to happen to *your* visitors, so you need to structure the catalog to make it as intuitive and helpful as possible.

Because the e-store will start with around 100 products and will probably have many more in the future, it's not enough to just group them in categories. The store also has a number of departments, and each department will contain a number of categories. Each category can then have any number of products attached to it.

■ **Note** Later in the book, you'll also create the administrative part of the web site, often referred to as the *control panel*, which allows the client to update department, category, and product data. Until then, you'll manually fill in the database with data (or you can "cheat" by using the SQL scripts provided in the Source Code/Download section of the Apress web site at `http://www.apress.com`, as you'll see).

Another particularly important detail that you need to think about is whether a category can exist in more than one department and whether a product can exist in more than one category. As you might suspect, this is the kind of decision that has implications on the way you code the product catalog, so you need to consult your client on this matter.

For the TShirtShop product catalog, each category can exist in only one department, but a product can exist in more than one category. For example, in our catalog, the product Kat Over New Moon will appear in both Animal and Christmas categories. This decision will have implications in the way you'll design the database, and we'll highlight those implications when we get there.

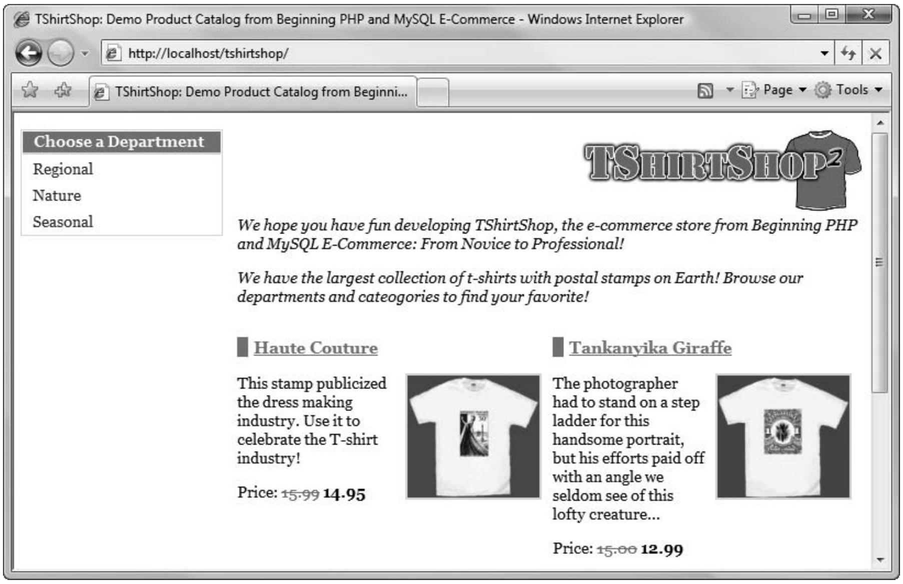
Finally, apart from having the products grouped in categories, you also want to have featured products. For this web site, a product can be featured either on the front page or in the department pages. The next section shows a few screenshots that explain this.

Previewing the Product Catalog

Although you'll have the fully functional product catalog finished by the end of Chapter 5, taking a look at it right now will give you a better idea about where you're heading. In Figure 4-1, you can see the TShirtShop front page and two of its featured products.

Note the departments list in the upper-left corner of the page. The list of departments is dynamically generated with data gathered from the database; you'll implement the list of departments in this chapter.

When site visitors click a department in the departments list, they go to the main page of the specified department. This replaces the store's list of catalog-featured products with a page containing information specific to the selected department—including the list of featured products for that department. In Figure 4-2, you see the page that will appear when the Seasonal department is clicked.


TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
TSHIRTSHOP²

Choose a Department
Regional
Nature
Seasonal

We hope you have fun developing TShirtShop, the e-commerce store from Beginning PHP and MySQL E-Commerce: From Novice to Professional!

We have the largest collection of t-shirts with postal stamps on Earth! Browse our departments and cateogories to find your favorite!

Haute Couture

This stamp publicized the dress making industry. Use it to celebrate the T-shirt industry!

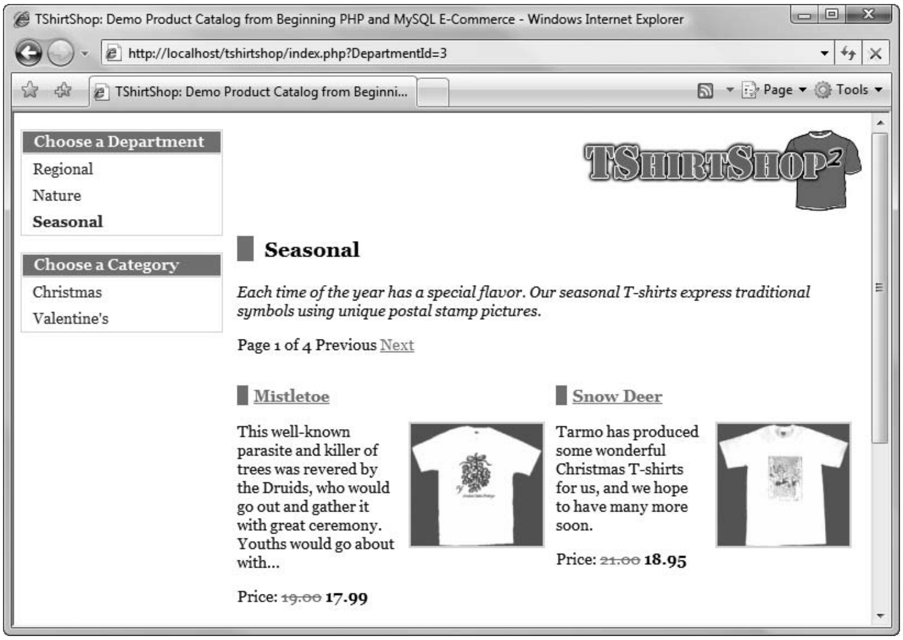
Price: ~~15.99~~ **14.95**

Tankanyika Giraffe

The photographer had to stand on a step ladder for this handsome portrait, but his efforts paid off with an angle we seldom see of this lofty creature...

Price: ~~15.00~~ **12.99**

Figure 4-1. *TShirtShop front page and two of its featured products*


TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/index.php?DepartmentId=3
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
TSHIRTSHOP²

Choose a Department
Regional
Nature
Seasonal

Choose a Category
Christmas
Valentine's

Seasonal

Each time of the year has a special flavor. Our seasonal T-shirts express traditional symbols using unique postal stamp pictures.

Page 1 of 4 Previous Next

Mistletoe

This well-known parasite and killer of trees was revered by the Druids, who would go out and gather it with great ceremony. Youths would go about with...

Price: ~~19.00~~ **17.99**

Snow Deer

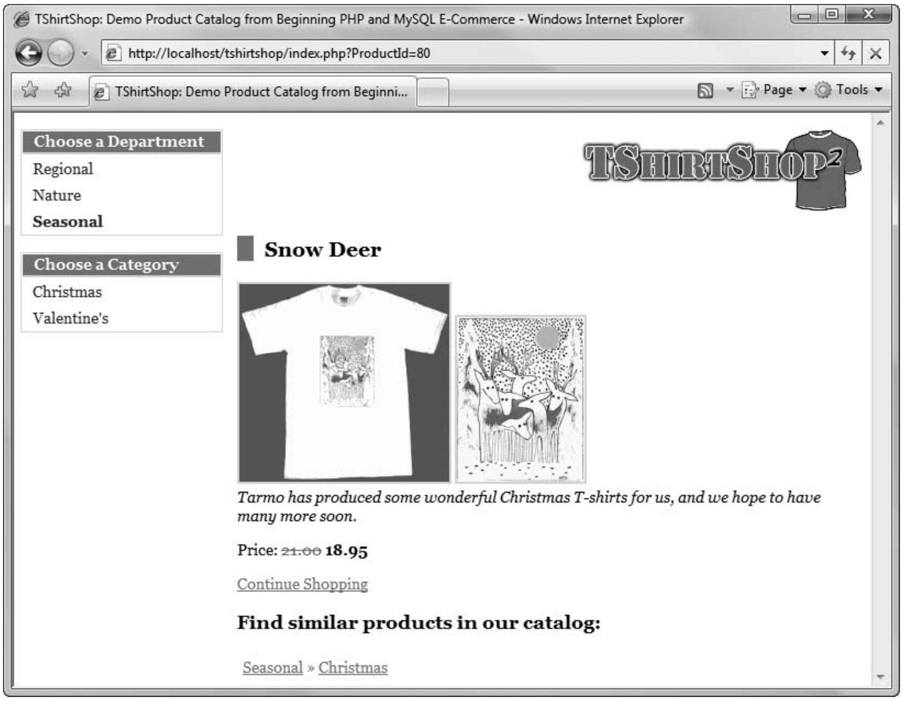
Tarmo has produced some wonderful Christmas T-shirts for us, and we hope to have many more soon.

Price: ~~21.00~~ **18.95**

Figure 4-2. *Visiting the Seasonal department*

Under the list of departments, you can now see the list of categories that belong to the selected department. On the right side of the screen, you can see the name of the selected department, its description, and its featured products. When a particular page must display a larger number of products than a predefined value, the products will be split into more subpages, and a pager shows up to allow the navigation between these pages. You can see this pager in Figure 4-2.

We decided to list only the featured products in the department page, and let the visitors browse all the products by navigating to category pages. On a department page, the text above the list of products is the description for the selected department, which means you'll need to store both a name and a description for each department in the database. When selecting a category from the categories list, all of its products are listed, along with the category title and description. Clicking a product's image or title takes you to a product details page, which you can see in Figure 4-3. The department and category boxes must retain their state when a product is selected; this is a navigational aid for the visitor. A Continue Shopping link also shows up, helping the visitor go back to the page he or she was visiting prior to selecting a product.


TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/index.php?ProductId=80
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
Choose a Department
Regional
Nature
Seasonal
Choose a Category
Christmas
Valentine's
TSHIRTSHOP2
Snow Deer
Continue Shopping
Find similar products in our catalog:
Seasonal » Christmas

Tarmo has produced some wonderful Christmas T-shirts for us, and we hope to have many more soon.

Price: ~~21.00~~ **18.95**

Figure 4-3. *Visiting a product details page*

When a category is selected, all its products are listed—you no longer see featured products. Note that the description text also changes. This time, this is the description of the selected category.

Roadmap for This Chapter

As you can see, the product catalog, although not very complicated, has more parts that need to be covered. In this chapter, you'll only create the departments list (see Figure 4-4).

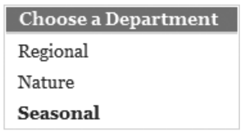


Figure 4-4. *The departments list*

The departments list will be the first dynamically generated data in your site (the names of the departments will be extracted from the database).

In this chapter, you'll implement just the departments list part of the web site. After you understand how this list works, you'll be able to quickly implement the other components of the product catalog in Chapter 5.

In Chapter 2, we discussed the three-tiered architecture that you'll use to implement the web application. The product catalog part of the site is no exception to the rule, and its components (including the departments list) will be spread over the three logical layers. Figure 4-5 previews what you'll create in this chapter at each tier to achieve a functional departments list.

So far, you've only played a bit with the presentation and business tiers in Chapter 3. Now, when building the catalog, you'll finally meet the final tier and work further with the `tshirtshop` database (depending on whom you ask, the data store may or may not be considered an integral part of the three-tiered architecture).

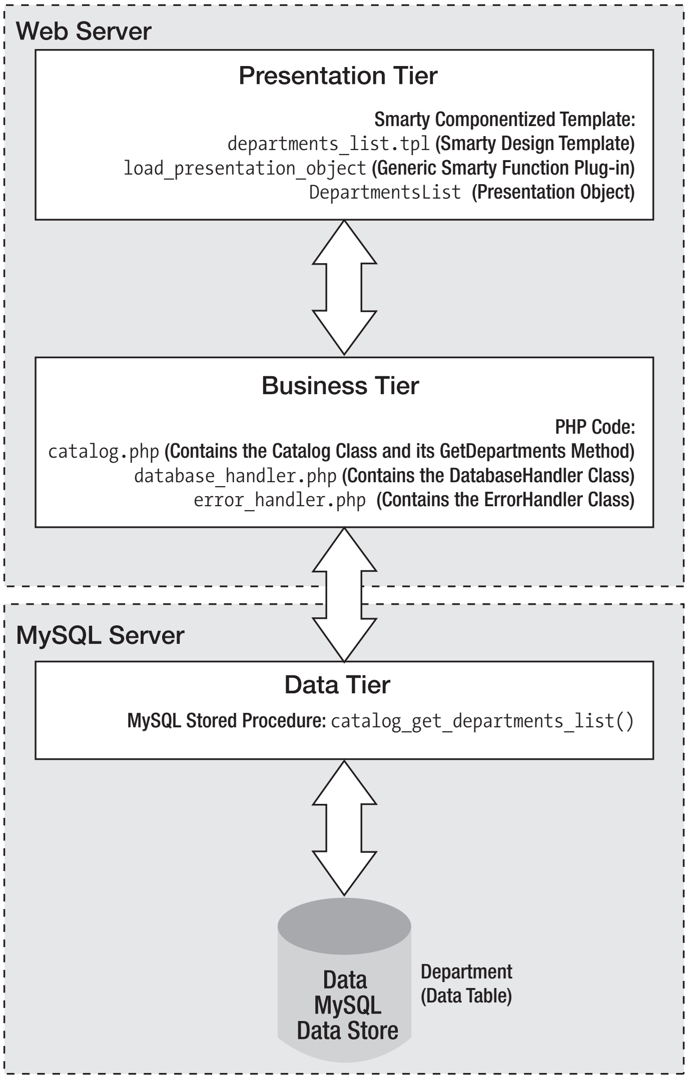

Web Server
Presentation Tier
Smarty Componentized Template:
departments_list.tpl (Smarty Design Template)
load_presentation_object (Generic Smarty Function Plug-in)
DepartmentsList (Presentation Object)
Business Tier
PHP Code:
catalog.php (Contains the Catalog Class and its GetDepartments Method)
database_handler.php (Contains the DatabaseHandler Class)
error_handler.php (Contains the ErrorHandler Class)
MySQL Server
Data Tier
MySQL Stored Procedure: catalog_get_departments_list()
Data MySQL Data Store
Department (Data Table)

Figure 4-5. *The components of the departments list*

These are the main steps you'll take toward having your own dynamically generated department list. Note that you start with the database and make your way to the presentation tier:

1. Create the `department` table in the database. This table will store data regarding the store's departments. Before adding this table, you'll learn the basic concepts of working with relational databases.

2. Write a MySQL stored procedure named `catalog_get_departments_list`, which returns the IDs and names of the departments from the `department` table. PHP scripts will call this stored procedure to generate the departments list for your visitor. MySQL stored procedures are logically located in the data tier of your application. At this step, you'll learn how to speak to your relational database using SQL.

3. Create the `DatabaseHandler` class, which will be your helper class that performs common database interaction operations. `DatabaseHandler` is a wrapper class for some PDO methods and includes consistent error-handling techniques that deal with database-related errors.
4. Create the business tier components of the departments list (the `Catalog` class and its `GetDepartments()` method). You'll see how to communicate with the database, through the `DatabaseHandler` helper class, to retrieve the necessary data.
5. Implement the `departments_list` Smarty componentized template, the `load_presentation_object` Smarty plug-in that glues the templates and their associated presentation objects. The `DepartmentList` presentation object is needed by the `departments_list` template.

So, let's start by creating the `department` table.

Storing Catalog Information

The vast majority of web applications, e-commerce web sites being no exception, live around the data they manage. Analyzing and understanding the data you need to store and process is an essential step in successfully completing your project.

The typical data storage solution for this kind of application is a relational database. However, this is not a requirement—you have the freedom to create your own data access layer and have whatever kind of data structures you want to support your application.

■ **Note** In some particular cases, it may be preferable to store your data in plain text files or XML files instead of databases, but these solutions are generally not suited for applications such as TShirtShop, so we won't cover them in this book. However, it's good to know your options.

Although this is not a book about databases or relational database design, you'll learn all you need to know to understand the product catalog and make it work.

Essentially, a relational database is made up of *data tables* and the *relationships* that exist among them. Because you'll work with a single data table in this chapter, we'll cover only the database theory that applies to the table as a separate, individual database item. In the next chapter, when you'll add the other tables to the picture, we'll take a closer look at the theory behind relational databases by analyzing how the tables relate to each other and how MySQL helps you deal with these relationships.

■ **Note** In a real-world situation, you would probably design the whole database (or at least all the tables relevant to the feature you build) from the start. In this book, we chose to split the development over two chapters to maintain a better balance of theory and practice.

So, let's start with a little bit of theory, after which you'll create the `department` data table and the rest of the required components:

Understanding Data Tables

This section provides a quick database lesson covering the essential information you need to know to design simple data tables. We'll briefly discuss the main parts that make up a database table:

- Primary keys
- MySQL data types
- `UNIQUE` columns
- `NOT NULL` columns and default values
- Autoincrement columns
- Indexes

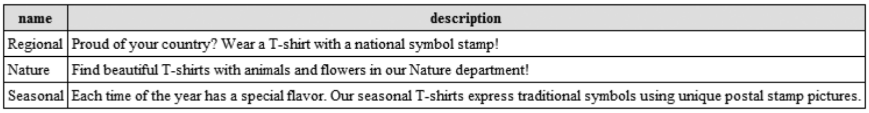
■ **Note** If you have previous experience with MySQL, you might want to skip this section and go directly to the "Creating the department Table" section.

A data table is made up of columns and rows. Columns are also referred to as *fields*, and rows are sometimes also called *records*.

Because this chapter covers the only departments list, you'll only need to create one data table: the `department` table. This table will store your departments' data and is one of the simplest tables you'll work with.

With the help of the MySQL client console interface, it's easy to create a data table in the database *if* you know for sure what kind of data it will store. When designing a table, you must consider which fields it should contain and which data types should be used for those fields. Besides a field's data type, there are a few more properties to consider, which you'll learn about in the following pages.

To determine which fields you need for the `department` table, write down a few examples of records that would be stored in that table. Remember from the previous figures that there isn't much information to store about a department—just the name and description for each department. The table containing the departments' data might look like Figure 4-6 (you'll implement the table in the database later, after we discuss the theory).



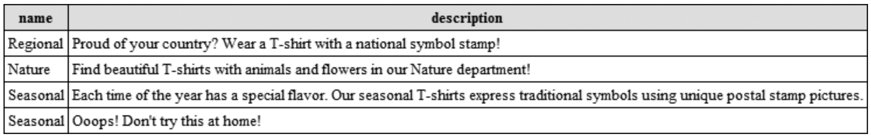
name	description
Regional	Proud of your country? Wear a T-shirt with a national symbol stamp!
Nature	Find beautiful T-shirts with animals and flowers in our Nature department!
Seasonal	Each time of the year has a special flavor. Our seasonal T-shirts express traditional symbols using unique postal stamp pictures.

Figure 4-6. *Data from the department table*

From a table like this, the names would be extracted to populate the list in the upper-left part of the web page, and the descriptions would be used as headers for the featured products list.

Primary Keys

The way you work with data tables in a relational database is a bit different from the way you usually work on paper. A fundamental requirement in relational databases is that each data row in a table must be *uniquely identifiable*. This makes sense because you usually save records into a database so that you can retrieve them later; however, you can't always do that if each table row doesn't have something that makes it unique. For example, suppose you add another record to the `department` table shown previously in Figure 4-6, making it look like the table shown in Figure 4-7.



name	description
Regional	Proud of your country? Wear a T-shirt with a national symbol stamp!
Nature	Find beautiful T-shirts with animals and flowers in our Nature department!
Seasonal	Each time of the year has a special flavor. Our seasonal T-shirts express traditional symbols using unique postal stamp pictures.
Seasonal	Ooops! Don't try this at home!

Figure 4-7. *Two departments with the same name*

Look at this table, and tell me the description of the Seasonal department! Yep, we have a problem—we have two departments with the same name Seasonal (the name isn't unique). If you queried the table using the `name` column, you would get two results. Sometimes getting multiple results for a query is what you expect—but other times you want the rows to be uniquely identifiable depending on the value of a column, which is supposed to be unique.

This problem is addressed, in the world of relational database design, using the concept of the *primary key*, which allows you to uniquely identify a specific row out of many rows. Technically, the primary key is not a column itself. Instead, the `PRIMARY KEY` is a *constraint* that when applied on a column guarantees that the column will have unique values across the table.

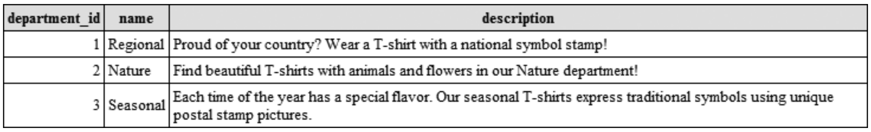
Constraints are rules that apply to data tables and make up part of the *data integrity* rules of the database. The database takes care of its own integrity and makes sure these rules aren't broken. If, for example, you try to add two identical values for a column that has a `PRIMARY KEY` constraint, the database refuses the operation and generates an error. We'll do some experiments later in this chapter to show this.

■ **Note** A primary key is not a column but a constraint that applies to that column; however, from now on and for convenience, when referring to the primary key, we'll be talking about the column that has the `PRIMARY KEY` constraint applied to it.

Back to the example, setting the `name` column as the primary key of the `department` table would solve the problem because two departments would not be allowed to have the same name. If `name` is the primary key of the `department` table, searching for a product with a specific `name` will always produce exactly one result if the name exists, or no results if no records have the specified name.

■ **Tip** This is common sense, but it has to be said: a primary key column will never allow `NULL` values.

An alternative solution, and usually the preferred one, is to have an additional column in the table, called an `ID` column, to act as its primary key. With an `ID` column, the department table would look like Figure 4-8.



department_id	name	description
1	Regional	Proud of your country? Wear a T-shirt with a national symbol stamp!
2	Nature	Find beautiful T-shirts with animals and flowers in our Nature department!
3	Seasonal	Each time of the year has a special flavor. Our seasonal T-shirts express traditional symbols using unique postal stamp pictures.

Figure 4-8. *Adding an ID column as the primary key of department*

The primary key column is named `department_id`. We'll use this naming convention for primary key columns in all data tables we'll create. In this scenario, having departments with the same name is now acceptable, because they would have different IDs. (To guard against unique column values for columns that are not the primary key you'd need to use the `UNIQUE` constraint, which is discussed next.)

There are two main reasons it's better to create a separate numerical primary key column than to use the `name` (or another existing column) as the primary key:

Performance: The database engine handles sorting and searching operations much faster with numerical values than with strings. This becomes even more relevant in the context of working with multiple related tables that need to be frequently joined (you'll learn more about this in Chapter 5).

Department name changes: If you need to rely on the `ID` value being stable in time, creating an artificial key solves the problem because it's unlikely you'll ever want to change the `ID`.

In Figure 4-8, the primary key is composed of a single column, but this is not a requirement. If the primary key is set on more than one column, the group of primary key columns (taken as a unit) is guaranteed to be unique, but the individual columns that form the primary key can have repeating values in the table. In Chapter 5, you'll see an example of a multivalued primary key. For now, it's enough to know that they exist.

■ **Note** Applying a `PRIMARY KEY` constraint on a field also generates a unique index created on it by default. Indexes are objects that improve performance of many database operations, dramatically speeding up your web application (you'll learn more about this later in the "Indexes" section of this chapter).

Unique Columns

UNIQUE is yet another kind of constraint that can be applied to table columns. This constraint is similar to the PRIMARY KEY constraint in that it doesn't allow duplicate data in a column. Still, there are differences. Although there is only one PRIMARY KEY constraint per table, you are allowed to have as many UNIQUE constraints as you like.

Columns that have the UNIQUE constraint are useful when you already have a primary key but still have columns (or groups of columns) for which you want to have unique values. You can set name to be unique in the department table if you want to forbid repeating values.

The facts that you need to remember about UNIQUE constraints follow:

- The UNIQUE constraint forbids having identical values on the field.
- You can have more than one UNIQUE field in a data table.
- A UNIQUE field is allowed to accept NULL values, in which case it will accept any number of them.
- Indexes are automatically created on UNIQUE and PRIMARY KEY columns.

Columns and Data Types

Each column in a table has a particular data type. By looking at the department table shown in Figure 4-8, you can see that department_id has a numeric data type, whereas name and description contain text.

It's important to consider the many data types that MySQL Server supports so that you'll be able to make correct decisions about how to create your tables. Table 4-1 isn't an exhaustive list of MySQL data types, but it focuses on the main types you might come across in your project. Refer to the MySQL documentation for a more detailed list at http://www.mysql.org/doc/refman/5.1/en/data-types.html.

■ **Tip** For more information about any specific detail regarding MySQL or PHP, including MySQL data types, you can always refer to W. Jason Gilmore's *Beginning PHP and MySQL 5: From Novice to Professional, Second Edition* (Apress, 2006), which is an excellent reference.

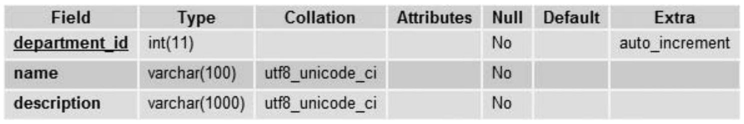
To keep the table short, under the Data Type heading, we have listed the types used in this project, while similar data types are explained under the Description and Notes headings. You don't need to memorize the list, but you should get an idea of which data types are available.

Table 4-1. *MySQL Server Data Types for Use in TShirtShop*

Data Type	Size in Bytes	Description and Notes
`int`	4	Stores integer numbers from –2,147,483,648 to 2,147,483,647. Related types are `bigint`, `mediumint`, `smallint`, and `tinyint`. A `bit` data type is able to store values of 0 and 1.
`decimal (M,N)`	`M+2` bytes if `N > 0`	One character for each digit of the value, the decimal point (if the scale is greater than 0), and the negative sign (for negative numbers).
	`M+1` bytes if `N = 0`	`decimal` is a numeric data type you'll use to store monetary information because of its exact precision. To preserve the decimal precision of these numbers, MySQL stores decimal values internally as strings. `M` represents the precision (the number of significant decimal digits that will be stored for values), and `N` is the scale (the number of digits after the decimal point). If `N` is 0, decimal will only store integer values.
`datetime`	8 bytes	Supports date and time data from `1000-01-01 00:00:00` to `9999-12-31 23:59:59`.
`varchar`	`variable`	Stores variable-length character data from 0 to 65,535. The dimension you set represents the maximum length of strings it can accept.
`text (blob)`	`L+2` bytes, where `L < 2^16`	A column with a maximum length of 65,535 (`2^16 - 1`) characters.

Keep in mind that data type names are case insensitive, so you might see them capitalized differently depending on the database console program you're using.

Now, let's get back to the `department` table and determine which data types to use. Don't worry that you don't have the table yet in your database; you'll create it a bit later. Figure 4-9 shows the fields of the `department` table. `department_id` is an `int` data type, and `name` and `description` are `varchar` data types.



Field	Type	Collation	Attributes	Null	Default	Extra
department_id	int(11)			No		auto_increment
name	varchar(100)	utf8_unicode_ci		No		
description	varchar(1000)	utf8_unicode_ci		No		

Figure 4-9. *Designing the department table*

For `varchar`, the associated dimension—such as in `varchar(100)`—represents the maximum length of the stored strings. We'll choose to have 100 characters available for the department's name and 1,000 for the description. An `integer` record, as shown in the table, always occupies 4 bytes.

NOT NULL Columns and Default Values

For each column of the table, you can specify whether it is allowed to be `NULL`. The best and shortest definition for `NULL` is "undefined." For example, in your `department` table, only `department_id` and `name` are really required, whereas `description` is optional—meaning that you are allowed to add a new department without supplying a description for it. If you add a new row of data without supplying a value for columns that allow nulls, `NULL` is automatically supplied for them.

Especially for character data, there is a subtle difference between the `NULL` value and an empty value. If you add a product with an empty string for its description, this means that you actually set a value for its description; it's an empty string, not an undefined (`NULL`) value.

The primary key field never allows `NULL` values. For the other columns, it's up to you to decide which fields are required and which are not.

In some cases, instead of allowing `NULL`s, you'll prefer to specify default values. This way, if the value is unspecified when creating a new row, it will be supplied with the default value. The default value can be a literal value (such as `0` for a `salary` column or `"unknown"` for a `description` column), a system value, or a function.

Autoincrement Columns

Autoincrement columns are automatically numbered columns. When a column is set as an autoincrement column, MySQL automatically provides values for it when inserting new records into the table. Usually if `max` is the largest value currently in the table for that column, then the next generated value will be `max + 1`.

This way, the generated values are always unique, which makes them especially useful when used in conjunction with the `PRIMARY KEY` constraint. You already know that primary keys are used on columns that uniquely identify each row of a table. If you set a primary key column to also be an autoincrement column, the MySQL server automatically fills that column with values when adding new rows (in other words, it generates new IDs), ensuring that the values are unique.

When setting an autoincrement column, the first value that the MySQL server provides for that column is `1`, but you can change this before adding data to your table with an SQL statement like the following:

```
ALTER TABLE your_table_name AUTO_INCREMENT = 1234;
```

This way, your MySQL server will start generating values with `1234`.

The table structure you saw in Figure 4-9 shows that `department_id` in your `department` table is an autoincrement column.

■ **Note** Unlike other database servers, MySQL still allows you to manually specify for an autonumbered field when adding new rows, if you want.

For more details about the autoincrement columns, see its official documentation at `http://www.mysql.org/doc/refman/5.1/en/example-auto-increment.html`.

Indexes

Indexes are related to MySQL performance tuning, so we'll mention them only briefly here. *Indexes* are database objects meant to increase the overall speed of database operations. Indexes work on the presumption that the vast majority of database operations are read operations. Indexes increase the speed of search operations but slow down insert, delete, and update operations. Usually, the gains of using indexes considerably outweigh the drawbacks.

On a table, you can create one or more indexes, with each index working on one column or on a set of columns. When a table is indexed on a specific column, its rows are either indexed or physically arranged based on the values of that column and the type of index. This makes search operations on that column very fast. If, for example, an index exists on `department_id` and then you do a search for the department with the `ID` value `934`, the search would be performed very quickly.

The drawback of indexes is that they can slow down database operations that add new rows or update existing ones because the index must be actualized (or the table rows rearranged) each time these operations occur.

You should keep the following in mind about indexes:

- Indexes greatly increase search operations on the database, but they slow down operations that change the database (delete, update, and insert operations).
- Having too many indexes can slow down the general performance of the database. The general rule is to set indexes on columns frequently used in `WHERE`, `ORDER BY`, and `GROUP BY` clauses or used in table joins.
- By default, unique indexes are automatically created on primary key table columns.

You can use dedicated tools to test the performance of a database under stress conditions with and without particular indexes; in fact, a serious database administrator will want to run some of these tests before deciding on a winning combination for indexes.

■ **Note** You learned about some data table properties in the previous pages. For more details about each of them, refer to the MySQL online manual at `http://dev.mysql.com/doc/` or Jason Gilmore's *Beginning PHP and MySQL 5: From Novice to Professional, Second Edition* (Apress, 2006).

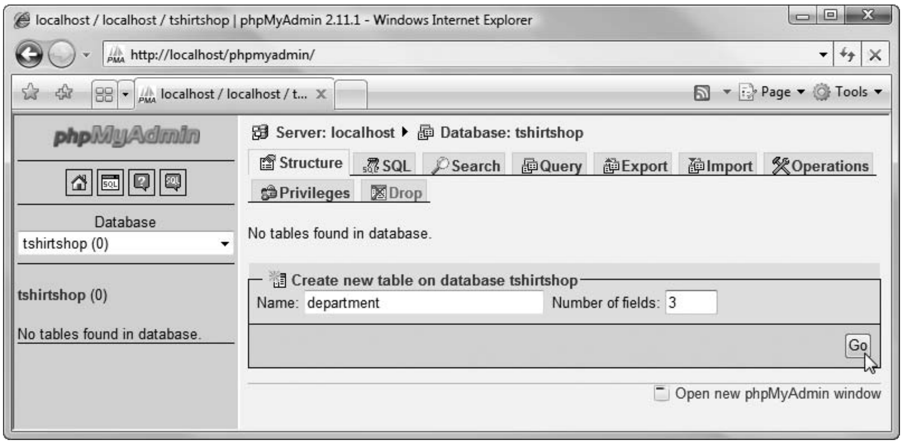
Creating the department Table

You created the `tshirtshop` database in Chapter 3. In the following exercise, you'll add the `department` table to it using the phpMyAdmin web client interface.

■ **Note** Alternatively, you can use the SQL scripts from the Source Code/Download section of the Apress web site to create and populate the `department` table. You can find the database creation scripts in the Database folder for this chapter in the code download for this book. You can also find the files on the authors' web sites.

Exercise: Creating the department Table

1. Point your web browser to your phpMyAdmin location (`http://localhost/phpmyadmin/`), like you did in Chapter 3 when creating the `tshirtshop` database.
2. Select the `tshirtshop` database from the Database combo box in the left side of the window. Type **department** in the Name text box of the "Create a new table on database tshirtshop" section, and type **3** in the "Number of fields" text box, as shown in Figure 4-10.


localhost / localhost / tshirtshop | phpMyAdmin 2.11.1 - Windows Internet Explorer
http://localhost/phpmyadmin/
localhost / localhost / t...
Page
Tools
phpMyAdmin
Server: localhost
Database: tshirtshop
Structure
SQL
Search
Query
Export
Import
Operations
Privileges
Drop
Database
tshirtshop (0)
Go
Open new phpMyAdmin window

tshirtshop (0)

No tables found in database.

No tables found in database.

Create new table on database tshirtshop

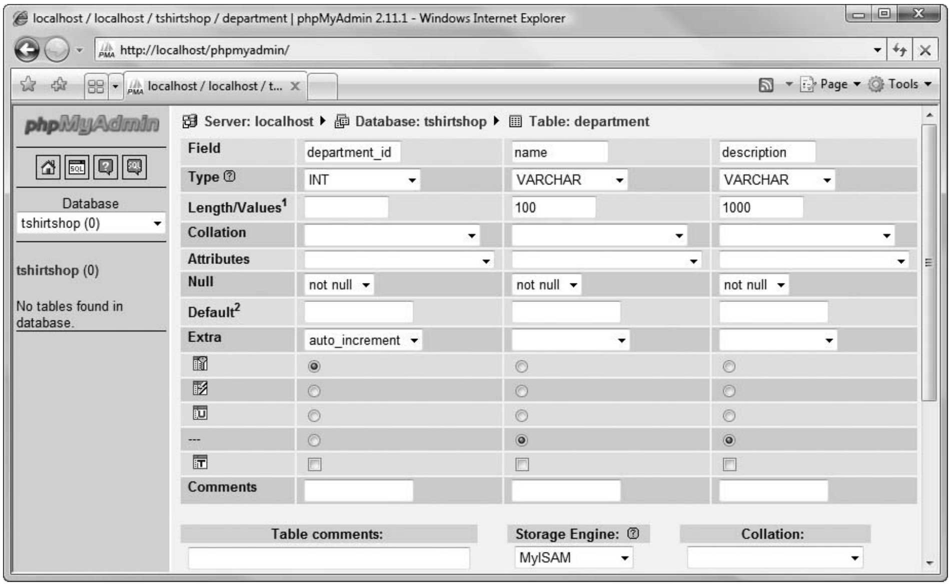
Name: department Number of fields: 3

Figure 4-10. *Adding the department table to the database*

3. Click Go. You'll be presented with a screen where you need to specify the details for each of the three table columns as shown in Figure 4-11.

   If you prefer to type the SQL code yourself instead of using the visual builder of phpMyAdmin, here's the code you need (you can find it in the source code download as well):

```
-- Create deparment table
CREATE TABLE `department` (
  `department_id` INT           NOT NULL  AUTO_INCREMENT,
  `name`          VARCHAR(100)  NOT NULL,
  `description`   VARCHAR(1000),
  PRIMARY KEY (`department_id`)
) ENGINE=MyISAM;
```


localhost / localhost / tshirtshop / department | phpMyAdmin 2.11.1 - Windows Internet Explorer

http://localhost/phpmyadmin/

Database: tshirtshop (0)

tshirtshop (0)

No tables found in database.

Server: localhost ▸ Database: tshirtshop ▸ Table: department

Field	department_id	name	description
Type	INT	VARCHAR	VARCHAR
Length/Values[1]		100	1000
Collation			
Attributes			
Null	not null	not null	not null
Default[2]			
Extra	auto_increment		
	●	○	○
	○	○	○
	○	○	○
---	○	●	●
	☐	☐	☐
Comments			

Table comments: | Storage Engine: MyISAM | Collation:

Figure 4-11. *Designing the department table*

4. Click Save. You'll be shown a page with many details about the table you just created. There, you can see the SQL code that was generated to create the table and various other details, such as confirmation that an index was indeed created automatically for the primary key field.

5. Now, you can add some sample data in the `department` table. Click the SQL tab, type the following query, and then click Go to execute it. The command should add three records to the `department` table you created earlier.

```
-- Populate department table
INSERT INTO `department` (`department_id`, `name`, `description`) VALUES
       (1, 'Regional', 'Proud of your country? Wear a T-shirt with a national
symbol stamp!'),
       (2, 'Nature', 'Find beautiful T-shirts with animals and flowers in our
Nature department!'),
       (3, 'Seasonal', 'Each time of the year has a special flavor. Our seasonal
T-shirts express traditional symbols using unique postal stamp pictures.');
```

How It Works: Creating MySQL Data Tables

You have just created your first database table! You also filled the table with some data using the `INSERT` SQL command, which we use to add records to a database table. You'll learn more about it soon.

As you can see, as soon as you have a clear idea about the structure of a table, it's relatively easy to use the phpMyAdmin web interface to create it into your database. Let's move on!

Communicating with the Database

Now that you have a table filled with data, let's do something useful with it! The ultimate goal with this table is to get the list of department names from a PHP page and populate the Smarty template with that list.

To get data from a database, you first need to know how to communicate with the database. Relational databases understand dialects and variants of *SQL*. The usual way of communicating with MySQL is to write an SQL command, send it to the MySQL server, and get the results back.

In practice, as you'll see later, we prefer to centralize the data access code using MySQL *stored procedures*, but before you can learn about them, you need to know the basics of SQL.

The Structured Query Language (SQL)

SQL is the language used to communicate with modern relational database management systems (RDBMSs). However, we haven't seen a database system yet that supports exactly the SQL 99 and SQL 2003 standards. This means that in many cases, the SQL code that works with one database will not work with the other. Currently, MySQL supports most of SQL 92 and an important part of SQL 99.

The most commonly used SQL commands are `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. These commands allow you to perform the most basic operations on the database.

The basic syntax of these commands is very simple, as you'll see in the following pages. However, keep in mind that SQL is a very flexible and powerful language and can be used to create much more complicated and powerful queries than what you see here. You'll learn more while building the web site, but for now, let's take a quick look at the basic syntax. For more details about any of these commands, you can always refer to their official documentation:

- `http://www.mysql.org/doc/refman/5.1/en/select.html`
- `http://www.mysql.org/doc/refman/5.1/en/insert.html`
- `http://www.mysql.org/doc/refman/5.1/en/update.html`
- `http://www.mysql.org/doc/refman/5.1/en/delete.html`

SELECT

The `SELECT` statement is used to query the database and retrieve selected data that match the criteria you specify. Its basic structure is

```
SELECT <column list>
[FROM <table name(s)>]
[WHERE <restrictive condition(s)>]
```

■ Note In this book, the SQL commands and queries appear in uppercase for consistency and clarity although SQL is not case sensitive. The `WHERE` and `FROM` clauses appear in brackets because they are optional.

The following command returns the name of the department that has the `department_id` of `1`. In your case, the returned value is `Regional`, but you would receive no results if there was no department with an `ID` of `1`.

```
SELECT name FROM department WHERE department_id = 1;
```

Tip You can easily test these queries to make sure they actually work by using the MySQL console interface or phpMyAdmin.

If you want more columns to be returned, you simply list them, separated by commas. Alternatively, you can use an asterisk (`*`), which means "all columns." However, for performance reasons, if you need only certain columns, you should list them separately instead of asking for them all. Using * is not advisable even if at a particular moment you do want all the columns for a query, because in the future, you may add even more columns to the table, and your query would end up asking for more data than is needed. Finally, using * doesn't guarantee the order in which the columns are returned, as the order of the columns in a table may change (although this is not likely to happen). For these reasons, we don't use * in this book.

With your current `department` table, the following two statements return the same results:

```
SELECT department_id, name, description
FROM   department
WHERE  department_id = 1;

SELECT * FROM department WHERE department_id = 1;
```

Tip You can split an SQL query on more lines, if you prefer—MySQL won't mind.

If you don't want to place any condition on the query, simply remove the `WHERE` clause, and you'll get all the rows. The following `SELECT` statement returns all rows and all columns from the `department` table:

```
SELECT * FROM department;
```

Tip If you are impatient and can't wait until later in the chapter, you can test the SQL queries right now by using the phpMyAdmin web client interface! Be careful, though, because in the rest of the book, we'll assume the data in your `department` table is the same as shown previously in the chapter.

Unless a sorting order is specified, the order in which the rows are returned by a `SELECT` clause can't be determined. Moreover, executing the same query twice could generate different results! To sort the results, you use `ORDER BY`. The following query will return the list of departments sorted alphabetically by the department name:

```
SELECT   department_id, name, description
FROM     department
ORDER BY name;
```

INSERT

The INSERT statement is used to insert a row of data into the table. Its syntax is as follows:

```
INSERT INTO <table name> [(column list)] VALUES (column values)
```

■ **Tip** Although the column list is optional (if you don't include it, column values are assigned to columns in the order in which they appear in the table's definition), you should always include it. This ensures that changing the table definition doesn't break the existing INSERT statements.

The following INSERT statement adds a department named Zodiac T-Shirts Department to the department table:

```
INSERT INTO department (name) VALUES ('Zodiac T-Shirts Department');
```

No value was specified for the description field, because it was marked to allow NULLs in the department table. This is why you can omit specifying a value, if you want to. Also, you're allowed to omit specifying a department ID, because the department_id column was created with the AUTO_INCREMENT option, which means the database takes care of automatically generating a value for it when adding new records. However, you're allowed to manually specify a value, if you prefer.

■ **Tip** Because department_id is the primary key column, trying to add more records with the same ID would cause the database to generate an error. The database doesn't permit having duplicate values in the primary key field.

When letting MySQL generate values for AUTO_INCREMENT columns, you can obtain the last generated value using the LAST_INSERT_ID() function. Here's an example of how this works:

```
INSERT INTO department (name) VALUES ('Some New Department');
SELECT LAST_INSERT_ID();
```

■ **Tip** In MySQL, the semicolon (;) is the delimiter between SQL commands.

UPDATE

The UPDATE statement is used to modify existing data and has the following syntax:

```
UPDATE  <table name>
SET <column name> = <new value> [, <column name> = <new value> ... ]
[WHERE <restrictive condition>]
```

The following query changes the name of the department with the ID of 43 to Cool Department. If there were more departments with that ID, all of them would have been modified, but because department_id is the primary key, you can't have more departments with the same ID.

```
UPDATE department SET name='Cool Department' WHERE department_id = 43;
```

Be careful with the UPDATE statement, because it makes messing up an entire table easy. If the WHERE clause is omitted, the change is applied to every record of the table, which you usually don't want to happen. MySQL will be happy to change all of your records; even if all departments in the table would have the same name and description, they would still be perceived as different entities because they have different department_id values.

DELETE

The syntax of the DELETE command is actually very simple:

```
DELETE FROM <table name>
[WHERE <restrictive condition>]
```

Most of the time, you'll want to use the WHERE clause to delete a single row:

```
DELETE FROM department WHERE department_id = 43;
```

As with UPDATE, be careful with this command, because if you forget to specify a WHERE clause, you'll end up deleting all of the rows in the table. The table itself isn't deleted by the DELETE command; for that purpose, you'd use DROP TABLE (http://dev.mysql.com/doc/refman/5.0/en/drop-table.html).

The following query deletes all the records in department:

```
DELETE FROM department;
```

MySQL Stored Procedures

A stored procedure is a named set of SQL commands stored in the MySQL server. Similar to functions in PHP, stored procedures can receive parameters and return data. Stored procedures in MySQL 5.1 follow the ANSI SQL 2003 specification. Their official documentation page is http://www.mysql.org/doc/refman/5.1/en/stored-procedures.html.

You don't need to use stored procedures if you want to perform database operations. You can directly send SQL commands from an external application (such as a PHP script of your TShirtShop application) to your MySQL database. When using stored procedures, instead of passing the SQL code you want executed, you just call the stored procedure and the values for any parameters it might have. Using stored procedures for data operations has the following advantages:

- Performance can be better, because MySQL generates an execution plan for the queries in the stored procedure when it's first executed, and then reuses the same plan on subsequent executions of the procedure.
- Using stored procedures allows for better maintainability of the data access and manipulation code, which is stored in a central place, and permits easier implementation of the three-tier architecture (the database stored procedures forming the data tier).
- Security can be better controlled, because MySQL permits setting different security permissions for each stored procedure.
- SQL queries created ad hoc in PHP code are more vulnerable to SQL injection attacks, which is a major security threat (many Internet resources cover this security subject, and you can find the most popular of them by Googling for "SQL injection attack").
- This might be a matter of taste, but separating the SQL logic from the PHP code keeps the PHP code cleaner and easier to manage; it simply looks better to execute a stored procedure than to build SQL queries by joining strings in PHP.

When developing TShirtShop, we'll save all the data access code as MySQL stored procedures inside the `tshirtshop` database. The syntax for creating stored procedures is

```
DELIMITER $$
CREATE PROCEDURE <name>(<param1 type>, <param2 type> ... )
BEGIN
  <code>
END$$

DELIMITER;
```

Note that the delimiter can be defined as something other than `$$`. The key is to define it as something different than the default delimiter, the semicolon.

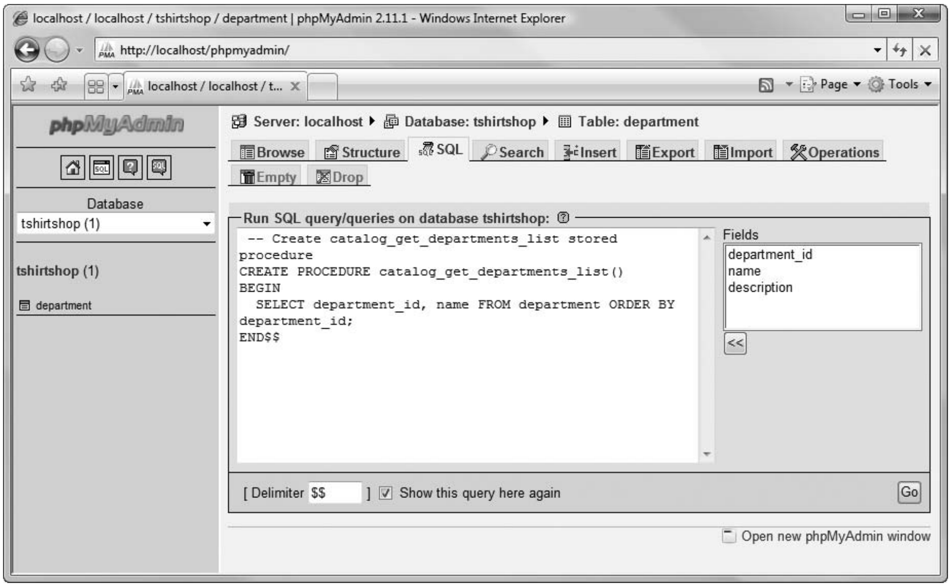
You can't create a stored procedure if your database already has a procedure with the same name. To remove an existing stored procedure, you use the `DROP PROCEDURE` command. To change the body or parameters of an existing procedure, you need to delete it using `DROP PROCEDURE` and create it again. MySQL supports a command named `ALTER PROCEDURE`, but unlike with other database applications, it can't be used to update the body or parameters of an existing procedure.

For the data tier of the departments list, you need to create stored procedure called `catalog_get_departments_list`. This procedure returns a list with the IDs and names of the departments in TShirtShop, and it will be called by business tier methods that need this data. Let's implement `catalog_get_departments_list` in the following exercise.

Exercise: Creating MySQL Stored Procedures

1. Load phpMyAdmin (`http://localhost/phpmyadmin/`) into your favorite browser. Select the `tshirtshop` database from the Database combo box in the left side of the window.
2. Select the SQL tab, and change the delimiter to `$$` as shown in Figure 4-12.
3. Execute the following code, which creates the `catalog_get_departments_list` stored procedure:

```
 -- Create catalog_get_departments_list stored procedure
CREATE PROCEDURE catalog_get_departments_list()
BEGIN
  SELECT department_id, name FROM department ORDER BY department_id;
END$$
```


localhost / localhost / tshirtshop / department | phpMyAdmin 2.11.1 - Windows Internet Explorer
http://localhost/phpmyadmin/
localhost / localhost / t...
Page
Tools
phpMyAdmin
Database
tshirtshop (1)
tshirtshop (1)
department
Server: localhost
Database: tshirtshop
Table: department
Browse
Structure
SQL
Search
Insert
Export
Import
Operations
Empty
Drop
Run SQL query/queries on database tshirtshop:
Fields
department_id
name
description
<<
[Delimiter $$]
Show this query here again
Go
Open new phpMyAdmin window

```
 -- Create catalog_get_departments_list stored
procedure
CREATE PROCEDURE catalog_get_departments_list()
BEGIN
  SELECT department_id, name FROM department ORDER BY
department_id;
END$$
```

Figure 4-12. *Creating the catalog_get_deparments_list stored procedure*

How It Works: MySQL Stored Procedures

Let's break down in parts the `catalog_get_departments_list` stored procedure. On the first line, we're defining the stored procedure name:

```
PROCEDURE catalog_get_departments_list()
```

The body of the stored procedure is between `BEGIN` and `END$$`. The following code snippet represents the typical way we'll code our stored procedures. The bold line represents the query we're interested in, and the rest is auxiliary code required to define the body of the stored procedure.

```
BEGIN
  SELECT department_id, name FROM department ORDER BY department_id;
END$$
```

So what happens here? The code that performs the actual functionality is written between `BEGIN` and `END$$`. The syntax may look weird at first, but what it does is pretty straightforward.

The stored procedure executes the `SELECT` statement and returns the results.

Adding Logic to the Site

The business tier (or middle tier) is said to be the brains of the application, because it manages the application's business logic. However, for simple tasks such as getting a list of departments from the data tier, the business tier doesn't have much logic to implement. It just requests the data from the database and passes it along. Usually, there will be a presentation tier object that will request this data, but it could be another business tier method that needs the data to implement some more complex functionality.

In this chapter, we're building the foundation of the business tier, which includes the functionality to open and close database connections, store data logic as MySQL stored procedures, and access these stored procedures from PHP.

For the business tier of the departments list, you'll implement two classes:

- `DatabaseHandler` will store the common functionality that you'll reuse whenever you need to access the database. Having this kind of generic functionality packed in a separate class saves keystrokes and avoids bugs in the long run.

- `Catalog` contains product-catalog-specific functionality, such as the `GetDepartments()` method that will retrieve the list of departments from the database.

Connecting to MySQL

The SQL queries you write must be sent somehow to the database engine for execution. As you learned in Chapter 2, you'll use PHP PDO to access the MySQL server.

Before writing the business tier code, you need to analyze and understand the possibilities for implementation. The important questions to answer before writing any code include the following:

- What strategy should you adopt for opening and closing database connections when you need to execute an SQL query?

- Which methods of PHP PDO should you use for executing database stored procedures and returning the results?

- How should you handle possible errors and integrate the error-handling solution with the error-handling code you wrote in Chapter 3?

Let's have a look at each of these questions one by one, and then we'll start writing some code.

Opening and Closing Connections to the MySQL Server

There are two main possible approaches you can take for this. The first is illustrated by the following sequence of actions, which needs to be executed each time the database needs to be accessed.

1. *Open* a connection to the database immediately before you need to execute a command on the database.

2. *Execute* the SQL query (or the database stored procedure) using the open connection, and get back the results. At this stage, you also need to handle any possible errors.

3. *Close* the database connection immediately after executing the command.

This method has the advantage that you don't keep database connections for a long time (which is good because database connections consume server resources), and it is also encouraged for servers that don't allow many simultaneous database connections. The disadvantage is the overhead implied by opening and closing the database connection every time, which can be partially reduced by using persistent connections.

■ **Note** "Persistent connections" refers to a technology that attempts to improve the efficiency of opening and closing database connections with no impact on functionality. You can learn more about this technology at `http://www.php.net/manual/en/features.persistent-connections.php`.

The alternative solution, and the one you'll use when implementing TShirtShop, is like this:

1. *Open* a connection to the database the first time you need to access the database during a request.
2. *Execute* all database stored procedures (or SQL queries) through that connection without closing it. Here, you also need to handle any possible errors.
3. *Close* the database connection when the client request finishes processing.

Using this method, all database operations that happen for a single client request (which happens each time a user visits a new page of our site) will go through a single database connection, avoiding opening and closing the connection each time you need something from the database. You'll still use persistent connections to improve the efficiency of opening a new database connection for each client request.

This solution is the one you will use for data access in the TShirtShop project.

Using PHP PDO for Database Operations

Now, we'll talk about how to put this in practice—opening and closing database connections and executing queries using those connections—using PHP PDO.

As explained in Chapter 2, you won't access MySQL through PHP's MySQL extension functions, but through a database abstraction layer (PHP PDO). The PDO classes permit accessing various data sources using the same application programming interface (API), so you won't need to change the PHP data access code or learn different data-access techniques when working with database systems other than MySQL (but you might need to change the SQL code itself if the database you migrate to uses a different dialect). Using PHP PDO is the modern way to interact with your database, and it makes your life as a programmer easier in the long run.

The important PHP PDO class you'll work with is `PDO`, which provides methods for performing various database operations. We can take advantage of the many methods already in the `PDO` class to process data, make connections to the DB, and for many other common tasks; we are spared having to write the code for these common tasks, because they are already included in the `PDO` class. It is a good idea to be familiar with the methods that are made available to you through the `PDO` class—you don't have to understand exactly how they work, but knowing that the functionality is already available can save you hours of painstakingly reinventing the wheel.

■ **Note** In this book, you'll learn about the PHP PDO functionality as used in TShirtShop. For more details about PHP PDO, see the *PHP Manual* documentation at `http://www.php.net/manual/en/ref.pdo.php`.

The `PDO` class provides the functionality to connect to the MySQL server and execute SQL queries. The method that opens a database connection is `PDO`'s constructor, which receives as parameters the connection string to the database server and an optional parameter that specifies whether the connection is a persistent connection. The connection string contains the data required to connect to the database server. You create a new `PDO` object like this:

```
$dbh = new PDO('mysql:dbname=' . $db_name . ';host=' . $db_host,
               $db_user,
               $db_pass,
               array(PDO::ATTR_PERSISTENT => $persistent));
```

■ **Note** The constructor of the `PDO` class returns an initialized database connection object (which is specific to the type of database you're connecting to, such as `mysql`) if the connection is successful; otherwise, an exception is thrown.

The previous code snippet shows the standard data you need to supply when connecting to a MySQL server and uses five variables:

- `$db_user` represents the username.
- `$db_pass` represents the user's password.
- `$db_host` is the hostname of your MySQL server.
- `$db_name` is the name of the database you're connecting to.
- `$persistent` is `true` if we want to create a persistent database connection or `false` otherwise.

To disconnect from the database, you need to make `$dbh` equal `null` (`$dbh = null`).

The following code snippet demonstrates how to create, open, and then close a MySQL database connection and also catch any exceptions that are thrown:

```
try
{
  // Open connection
  $dbh = new PDO('mysql:dbname=' . $db_name . ';host=' . $db_host,
                 $db_user, $db_pass);

  // Close connection
  $dbh = null;
}
```

```
catch (PDOException $e)
{
  echo 'Connection failed: ' . $e->getMessage();
}
```

The `try` and `catch` keywords are used to handle *exceptions*.

PHP 5 EXCEPTION HANDLING

In Chapter 3, you implemented the code that intercepts and handles (and eventually reports) errors that happen in the TShirtShop site. *PHP errors* are the standard mechanism that you can use to react with an error happening in your PHP code. When a PHP error occurs, the execution stops; you can, however, define an error-handling function that is called just before the execution is terminated. You added such a function in Chapter 3, where you obtained as many details as possible about the error and logged them for future reference. Having those details, a programmer can fix the code to avoid the same error happening in the future.

PHP 5 introduced, along with other object-oriented programming (OOP) features, a new way to handle runtime errors: enter exceptions. Exceptions represent the modern way of managing runtime errors in your code and are much more powerful and flexible than PHP errors. Exceptions are a very important part of the OO (object oriented) model, and PHP 5 introduces an exception model resembling that of other OOP languages, such as Java and C#. However, exceptions in PHP coexist with the standard PHP errors in a strange combination, and you can't solely rely on exceptions for dealing with runtime problems. Some PHP extensions, such as PDO, can be configured to generate exceptions to signal problems that happen at runtime, whereas in other cases, your only option is to deal with standard PHP errors.

The advantages of exceptions over errors lay in the flexibility you're offered in handling them. When an exception is generated, you can handle it locally and let your script continue executing normally, or you can pass the exception to another class for further processing. With exceptions, your script isn't terminated like it is when a PHP error appears. When using exceptions, you place the code that you suspect could throw an exception inside a `try` block and handle potential exceptions in an associated `catch` block:

```
try
{
  // Code that could generate an exception that you want to handle
}
catch (Exception $e)
{
  // Code that is executed when an exception is generated
  // (exception details are accessible through the $e object)
}
```

When an exception is generated by any of the code in the `try` block, the execution is passed directly to the `catch` block. Unless the code in the `catch` block rethrows the exception, it is assumed that it handled the exception, and the execution of your script continues normally. This kind of flexibility allows you to prevent many causes that could make your pages stop working, and you'll appreciate the power exceptions give you when writing PHP code!

A PHP 5 exception is represented by the `Exception` class, which contains the exception's details. You can generate (throw) an exception yourself using the `throw` keyword. The `Exception` object that you throw is propagated through the call stack until it is intercepted using the `catch` keyword. The *call stack* is the list of methods being executed. So if a function `A()` calls a function `B()`, which in turn calls a function `C()`, then the call stack will be formed of these three methods. In this scenario, an exception that is raised in function `C()` can be handled in the same function, provided the offending code is inside a `try-catch` block. If this is not the case, the exception propagates to method `B()`, which has a chance to handle the exception, and so on. If no method handles the exception, the exception is finally intercepted by the PHP interpreter, which transforms the exception into a PHP Fatal Error.

In our database-handling code, we'll catch the potential exceptions that could be generated by PDO. Although it doesn't do it by default, PDO can be instructed to generate exceptions in case something goes wrong when executing an SQL command or opening a database connection, like this:

```
// Create a new PDO class instance
$handler = new PDO( ... );

// Configure PDO to throw exceptions
self::$_mHandler->setAttribute(PDO::ATTR_ERRMODE,
                               PDO::ERRMODE_EXCEPTION);
```

We catch any exceptions the data access code may throw, and we pass the error details to the error-handling code you wrote in Chapter 3. The following code snippet shows a short method with this functionality implemented:

```
// Wrapper method for PDOStatement::fetch()
public static function GetRow($sqlQuery, $params = null,
                              $fetchStyle = PDO::FETCH_ASSOC)
{
  // Initialize the return value to null
  $result = null;

  // Try to execute an SQL query or a stored procedure
  try
  {
    // Get the database handler
    $database_handler = self::GetHandler();

    // Prepare the query for execution
    $statement_handler = $database_handler->prepare($sqlQuery);

    // Execute the query
    $statement_handler->execute($params);

    // Fetch result
    $result = $statement_handler->fetch($fetchStyle);
  }
```

```
    // Trigger an error if an exception was thrown when executing the SQL query
    catch(PDOException $e)
    {
      // Close the database handler and trigger an error
      self::Close();
      trigger_error($e->getMessage(), E_USER_ERROR);
    }

    // Return the query results
    return $result;
  }
```

Issuing Commands Using the Connection

After opening the connection, you're now at the stage we've been aiming for from the start: executing SQL commands through the connection.

You can execute the command in many ways, depending on the specifics. Does the SQL query you want to execute return any data? If so, what kind of data and in which format? The PDO methods that we'll use to execute SQL queries follow:

- `PDOStatement::execute()` is used to execute an `INSERT`, `UPDATE`, or `DELETE` queries that don't return any data.
- `PDOStatement::fetch()` is used to retrieve one row of data from the database.
- `PDOStatement::fetchAll()` is used to retrieve multiple rows of data from the database.
- `PDO::prepare()` prepares an SQL query to be executed, creating a so-called prepared statement.

A *prepared statement* is a parameterized SQL query whose parameter values are replaced by either parameter markers (`?`) or named variables (`:variable_name`), like in these examples:

```
$query1 = "SELECT name FROM department WHERE department_id = ?"
$query2 = "SELECT name FROM department WHERE department_id = :dept_id"
```

To execute a prepared statement, you supply the parameter values to the functions that execute your query, which take care of building the complete SQL query for you. To implement the list of departments, you won't need to work with parameters, but you'll learn how to handle them in Chapter 5.

In this book, we'll always use prepared statements, because they bring two important benefits:

- Parameter values are checked to prevent injection attacks.
- The query will likely execute faster with prepared statements, because the database server can reuse the access plan it builds for a prepared statement.

To be able to reuse more of the database-handling code and to have a centralized error-handling mechanism for the database code, we won't be using the PDO methods directly from the business tier of our application. Instead, we'll wrap the PDO functionality into a class named `DatabaseHandler`, and we'll use this class from the other classes of the business tier.

Writing the Business Tier Code

OK, let's write some code! You'll start by writing the DatabaseHandler class, which will be a support class that contains generic functionality needed in the other business tier methods. Next, you'll create a business tier class named Catalog, which uses the DatabaseHandler class to provide the functionality required by the presentation tier. The Catalog class will contain methods such as GetDepartments() (which will be used to generate the list of departments), GetCategories(), and so on. The only method we'll need to add to the Catalog class in this chapter is GetDepartments().

Although in this chapter we won't need all this functionality, we'll write the complete code of the DatabaseHandler class. DatabaseHandler will have the following methods:

- Execute() executes a stored procedure that doesn't return records from the database, such as INSERT, DELETE, or UPDATE statements.
- GetAll() is used to execute queries that return more rows of data, such as when requesting the list of departments.
- GetRow() is used to execute queries that return a row data.
- GetOne() returns a single value from the database. We can use this method to call database stored procedures that return a single value, such as one that returns the subtotal of a shopping cart.

Exercise: Creating and Using the DatabaseHandler Class

1. Add the database login information at the end of tshirtshop/include/config.php, modifying the constants' values to fit your server's configuration. The following code assumes you created the admin user account as instructed in Chapter 3:

```
// Database connectivity setup
define('DB_PERSISTENCY', 'true');
define('DB_SERVER', 'localhost');
define('DB_USERNAME', 'tshirtshopadmin');
define('DB_PASSWORD', 'tshirtshopadmin');
define('DB_DATABASE', 'tshirtshop');
define('PDO_DSN', 'mysql:host=' . DB_SERVER . ';dbname=' . DB_DATABASE);
```

2. Create a new file named database_handler.php in the tshirtshop/business folder, and create the DatabaseHandler class as shown in the following code listing. At this moment, we only included its constructor (which is private, so the class can't be instantiated), and the static GetHandler() method, which creates a new database connection, saves it into the $_mHandler member, and then returns this object (find more explanations about the process in the upcoming "How it Works" section).

```
<?php
// Class providing generic data access functionality
class DatabaseHandler
{
  // Hold an instance of the PDO class
  private static $_mHandler;
```

```
    // Private constructor to prevent direct creation of object
    private function __construct()
    {
    }

    // Return an initialized database handler
    private static function GetHandler()
    {
      // Create a database connection only if one doesn't already exist
      if (!isset(self::$_mHandler))
      {
        // Execute code catching potential exceptions
        try
        {
          // Create a new PDO class instance
          self::$_mHandler =
            new PDO(PDO_DSN, DB_USERNAME, DB_PASSWORD,
                    array(PDO::ATTR_PERSISTENT => DB_PERSISTENCY));

          // Configure PDO to throw exceptions
          self::$_mHandler->setAttribute(PDO::ATTR_ERRMODE,
                                         PDO::ERRMODE_EXCEPTION);
        }
        catch (PDOException $e)
        {
          // Close the database handler and trigger an error
          self::Close();
          trigger_error($e->getMessage(), E_USER_ERROR);
        }
      }

      // Return the database handler
      return self::$_mHandler;
    }
  }
  ?>
```

3. Add the `Close()` method to the `DatabaseHandler` class. This method will be called to close the database connection:

```
    // Clear the PDO class instance
    public static function Close()
    {
      self::$_mHandler = null;
    }
```

4. Add the `Execute()` method to the `DatabaseHandler` class. This method uses the `PDOStatement::execute()` to run queries that don't return records (`INSERT`, `DELETE`, or `UPDATE` queries):

```
// Wrapper method for PDOStatement::execute()
public static function Execute($sqlQuery, $params = null)
{
  // Try to execute an SQL query or a stored procedure
  try
  {
    // Get the database handler
    $database_handler = self::GetHandler();

    // Prepare the query for execution
    $statement_handler = $database_handler->prepare($sqlQuery);

    // Execute query
    $statement_handler->execute($params);
  }
  // Trigger an error if an exception was thrown when executing the SQL query
  catch(PDOException $e)
  {
    // Close the database handler and trigger an error
    self::Close();
    trigger_error($e->getMessage(), E_USER_ERROR);
  }
}
```

5. Add the `GetAll()` method, which is the wrapper method for `PDOStatement::fetchAll()`. You'll call this method for retrieving a complete result set from a `SELECT` query:

```
// Wrapper method for PDOStatement::fetchAll()
public static function GetAll($sqlQuery, $params = null,
                              $fetchStyle = PDO::FETCH_ASSOC)
{
  // Initialize the return value to null
  $result = null;

  // Try to execute an SQL query or a stored procedure
  try
  {
    // Get the database handler
    $database_handler = self::GetHandler();

    // Prepare the query for execution
    $statement_handler = $database_handler->prepare($sqlQuery);

    // Execute the query
    $statement_handler->execute($params);
```

```
      // Fetch result
      $result = $statement_handler->fetchAll($fetchStyle);
    }
    // Trigger an error if an exception was thrown when executing the SQL query
    catch(PDOException $e)
    {
      // Close the database handler and trigger an error
      self::Close();
      trigger_error($e->getMessage(), E_USER_ERROR);
    }

    // Return the query results
    return $result;
  }
```

6. Add the `GetRow()` method, which is the wrapper class for `PDOStatement::fetch()`, as shown. This will be used to get a row of data resulted from a `SELECT` query:

```
  // Wrapper method for PDOStatement::fetch()
  public static function GetRow($sqlQuery, $params = null,
                                $fetchStyle = PDO::FETCH_ASSOC)
  {
    // Initialize the return value to null
    $result = null;

    // Try to execute an SQL query or a stored procedure
    try
    {
      // Get the database handler
      $database_handler = self::GetHandler();

      // Prepare the query for execution
      $statement_handler = $database_handler->prepare($sqlQuery);

      // Execute the query
      $statement_handler->execute($params);

      // Fetch result
      $result = $statement_handler->fetch($fetchStyle);
    }
    // Trigger an error if an exception was thrown when executing the SQL query
    catch(PDOException $e)
    {
      // Close the database handler and trigger an error
      self::Close();
      trigger_error($e->getMessage(), E_USER_ERROR);
    }
    // Return the query results
    return $result;
  }
```

7. Add the `GetOne()` method, which is the wrapper class for `PDOStatement::fetch()`, as shown. This will be used to get a single value resulted from a `SELECT` query:

```
  // Return the first column value from a row
  public static function GetOne($sqlQuery, $params = null)
  {
    // Initialize the return value to null
    $result = null;

    // Try to execute an SQL query or a stored procedure
    try
    {
      // Get the database handler
      $database_handler = self::GetHandler();

      // Prepare the query for execution
      $statement_handler = $database_handler->prepare($sqlQuery);

      // Execute the query
      $statement_handler->execute($params);

      // Fetch result
      $result = $statement_handler->fetch(PDO::FETCH_NUM);

      /* Save the first value of the result set (first column of the first row)
         to $result */
      $result = $result[0];
    }
    // Trigger an error if an exception was thrown when executing the SQL query
    catch(PDOException $e)
    {
      // Close the database handler and trigger an error
      self::Close();
      trigger_error($e->getMessage(), E_USER_ERROR);
    }

    // Return the query results
    return $result;
  }
```

8. Create a file named `catalog.php` file inside the `business` folder. Add the following code into this file:

```
<?php
// Business tier class for reading product catalog information
```

```
class Catalog
{
  // Retrieves all departments
  public static function GetDepartments()
  {
    // Build SQL query
    $sql = 'CALL catalog_get_departments_list()';

    // Execute the query and return the results
    return DatabaseHandler::GetAll($sql);
  }
}
?>
```

9. You need to include the newly created `database_handler.php` in `index.php` to make the class available for the application. To do this, add the highlighted code to the `index.php` file:

```
<?php
// Include utility files
require_once 'include/config.php';
require_once BUSINESS_DIR . 'error_handler.php';

// Sets the error handler
ErrorHandler::SetHandler();

// Load the application page template
require_once PRESENTATION_DIR . 'application.php';

// Load the database handler
require_once BUSINESS_DIR . 'database_handler.php';
```

10. At the end of `index.php`, add the highlighted code that closes the database connection:

```
// Load Smarty template file
$application = new Application();

// Display the page
$application->display('store_front.tpl');

// Close database connection
DatabaseHandler::Close();
?>
```

How It Works: The Business Tier Code

After adding the database connection data to `include/config.php`, you created the `DatabaseHandler` class. This class contains a number of wrapper methods that access PDO methods and provide the functionality needed for the rest of the business tier methods.

The `DatabaseHandler` class has a *private constructor*, meaning that it can't be instantiated; you can't create `DatabaseHandler` objects, but you can execute the *static methods* for the class. Static class members and methods, as opposed to instance members and methods, are owned not by a particular instance of the class but by the class as a whole. In other words, to execute an SQL query using `GetAll()`, we wouldn't create a new class instance, like in the following example (and we couldn't do it not only because there's no instance version of the `GetAll()` method but also because the private constructor prevents us from instantiating the `DatabaseHandler` class):

```
$myHandler = new DatabaseHandler();
$results = $myHandler->GetAll($sql);
```

Instead, static methods are called directly using the class name (using the `::` notation) as follows, instead of an object of the class (which uses the `->` notation):

```
DatabaseHandler::GetAll($sql);
```

Static members of a class are internally stored by PHP using a global instance of the class. In our PDO scenario, the advantage of storing the database connection in a static member (`private static $_mHandler`) is that all database operations our site makes during one web request go through this one database connection. As explained earlier, for performance, we prefer to use this technique instead of creating a new database connection for each query that needs to be executed, and the support of static members of PHP allows us to implement it.

■ **Note** Static members are OOP-specific features that aren't supported by PHP 4 and older versions. You can find a very good introduction to the OOP features in PHP 5 at `http://php.net/manual/en/language.oop5.php`.

The methods that execute database stored procedures have a standard structure, taking advantage of the fact that PDO has been configured to throw exceptions. Let's take a closer look at the `GetRow()` method:

```
  // Wrapper method for PDOStatement::fetch()
  public static function GetRow($sqlQuery, $params = null,
                                $fetchStyle = PDO::FETCH_ASSOC)
  {
    // Initialize the return value to null
    $result = null;

    // Try to execute an SQL query or a stored procedure
    try
    {
      // Get the database handler
      $database_handler = self::GetHandler();

      // Prepare the query for execution
      $statement_handler = $database_handler->prepare($sqlQuery);
      // Execute the query
      $statement_handler->execute($params);

      // Fetch result
      $result = $statement_handler->fetch($fetchStyle);
    }
    // Trigger an error if an exception was thrown when executing the SQL query
    catch(PDOException $e)
    {
      // Close the database handler and trigger an error
      self::Close();
      trigger_error($e->getMessage(), E_USER_ERROR);
    }

    // Return the query results
    return $result;
  }
```

This method generates an error (using the `trigger_error()` function) if the database command didn't execute successfully. The error is captured by the error-handling mechanism you implemented in Chapter 3.

Because of the way you implemented the error-handling code in Chapter 3, generating an `E_USER_ERROR` error freezes the execution of the request, eventually logging and/or e-mailing the error data, and showing the visitor a nice "Please come back later" message (if there is such thing as a nice "Please come back later" message, anyway).

Note that before the error is generated, we also close the database connection to ensure that we're not leaving any database resources occupied by the script.

By default, if you don't specify to `trigger_error()` the kind of error to generate, an `E_USER_NOTICE` message is generated, which doesn't interfere with the normal execution of the request (the error is eventually logged, but execution continues normally afterward).

The functionality in the `DatabaseHandler` class is meant to be used in the other business tier classes, such as `Catalog`. At this moment, `Catalog` contains a single method: `GetDepartments()`.

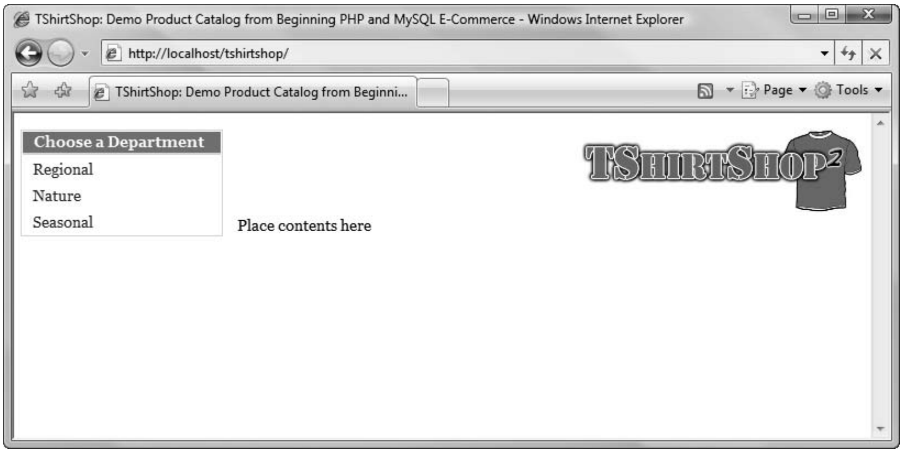
```
// Business tier class for reading product catalog information
class Catalog
{
  // Retrieves all departments
  public static function GetDepartments()
  {
    // Build SQL query
    $sql = 'CALL catalog_get_departments_list()';

    // Execute the query and return the results
    return DatabaseHandler::GetAll($sql);
  }
}
```

Because it relies on the functionality you've already included in the DatabaseHandler class and in the database functions in place, the code in Catalog is very simple and straightforward. The GetDepartments() method will be called from the presentation tier, which will display the returned data to the visitor. It starts by building the SQL query, and then calling the appropriate DatabaseHandler method to execute the query. In this case, we're calling GetAll() to retrieve the list of departments.

Right now, the database connection is opened when index.php starts processing and is closed at the end. All database operations that happen in one iteration of this file will be done through this connection.

Displaying the List of Departments

Now that everything is in place in the other tiers, all you have to do is create the presentation tier part—this is the final goal that we've been aiming toward from the beginning of this chapter. As shown previously, the departments list needs to look something like the one shown in Figure 4-13 when the site is loaded in the browser.


TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
TSHIRTSHOP2

Choose a Department
Regional
Nature
Seasonal

Place contents here

Figure 4-13. *TShirtShop with a dynamically generated list of departments*

You'll implement this functionality as a separate componentized template named departments_list. You'll then just include departments_list.tpl in the main Smarty template (templates/store_front.tpl).

The departments_list componentized template is made up of three files: the Smarty design template (templates/departments_list.tpl), the presentation object (presentation/departments_list.php), and the Smarty plug-in file (presentation/smarty_plugins/function.load_presentation_object.php). The Smarty plug-in file is a generic plug-in that will be used by all Smarty templates to load presentation objects.

Using Smarty Plug-ins

The *Smarty plug-in* is the Smarty technique we'll use to implement the logic behind Smarty design template files (with the `.tpl` extension). This is not the only way to store the logic behind a Smarty design template, but it's the way the Smarty documentation recommends at `http://smarty.php.net/manual/en/tips.componentized.templates.php`.

The layout and presentation in our project is generated using Smarty design templates. When a certain component is more complex and needs PHP code to supply it with additional data or functionality, we use a Smarty componentized template, which is composed of

- *Smarty design template*: This is a `.tpl` file containing HTML and Smarty-specific tags. For the departments list, the design template is named `departments_list.tpl`.

- *Smarty plug-in function*: The Smarty plug-in function is referenced from the Smarty design template, and its role is to supply the template with data it needs to display. In our project, the same Smarty plug-in function (`function.load_presentation_object.php`) will be loaded by all Smarty design templates. However, the Smarty plug-in function will return different results depending on the parameters it is invoked with.

- *Presentation object*: This is a class that returns the data required by a Smarty design template. In the case of the departments list, this class will be named `DepartmentsList`. This class reads the list of departments and stores it into a public member that can then be accessed by the Smarty design template.

Smarty plug-in files and functions must follow strict naming conventions to be located by Smarty. Smarty plug-in files must be named as `type.name.php` (in our case, `function.load_presentation_object.php`), and the functions inside these files must be named as `smarty_type_name` (in our case, `smarty_function_load_presentation_object`). The official page for Smarty plug-ins naming conventions is `http://smarty.php.net/manual/en/plugins.naming.conventions.php`. You can learn more about Smarty plug-ins at `http://smarty.php.net/manual/en/plugins.php`.

After the Smarty plug-in file is in place, you can reference it from the Smarty design template file (`departments_list.tpl`) with a line like this:

```
{load_presentation_object filename="departments_list" assign="obj"}
```

Given the correct naming conventions where used, this line is enough to get Smarty to load the plug-in file, which at its turn will load the presentation object mentioned through the `filename` parameter, and assign the loaded object to a template variable (in our example the name of the variable will be `obj`). The Smarty design template file can then access the variables populated by the plug-in function like this:

```
{$obj->mDepartments[i].name}
```

To understand how the whole mechanism works, let's create the `departments_list` componentized template, and all the other pieces required to have it working. We'll continue to use CSS for defining the visual styles of our presentation. While CSS is very powerful, learning the basics of its use is straightforward and easy—and even fun!

Exercise: Creating the departments_list Componentized Template

1. Open the `tshirtshop.css` file in the `tshirtshop/styles` folder, and add the following code listing. These styles refer to the way department names should look inside the departments list when they are unselected, unselected but with the mouse hovering over them, or selected.

```
div.yui-b div.box {
  color: #333333;
  border: 1px solid #c6e1ec;
  margin-top: 15px;
}

div.yui-b div p.box-title {
  background: #0590C7;
  border-bottom: 2px solid #c6e1ec;
  color: #FFFFFF;
  display: block;
  font-size: 93%;
  font-weight: bold;
  margin: 1px;
  padding: 2px 10px;
}

a {
  color: #0590C7;
}

a:hover {
  color: #ff0000;
}

a.selected {
  font-weight: bold;
}

div.yui-b div ul {
  margin: 0;
}

div.yui-b div ul li {
  border-bottom: 1px solid #fff;
  list-style-type: none;
}

div.yui-b div ul li a {
  color: #333333;
  display: block;
  text-decoration: none;
  padding: 3px 10px;
}

div.yui-b div ul li a:hover {
  background: #c6e1ec;
  color: #333333;
}
```

2. Edit the `presentation/application.php` file, and add the following two lines to the constructor of the `Application` class. These lines configure the plug-in folders used by Smarty. The first one is for the internal Smarty plug-ins, and the second specifies the `smarty_plugins` folder you'll create to hold the plug-ins you'll write for TShirtShop.

```
/* Class that extends Smarty, used to process and display Smarty
   files */
classApplication extends Smarty
{
  // Class constructor
  public function __construct()
  {
    // Call Smarty's constructor
    parent::Smarty();

    // Change the default template directories
    $this->template_dir = TEMPLATE_DIR;
    $this->compile_dir = COMPILE_DIR;
    $this->config_dir = CONFIG_DIR;
    $this->plugins_dir[0] = SMARTY_DIR . 'plugins';
    $this->plugins_dir[1] = PRESENTATION_DIR . 'smarty_plugins';
  }
}
```

3. Now, create the Smarty template file for the `departments_list` componentized template. Write the following lines in `presentation/templates/departments_list.tpl`. This will create the presentation or visual layout of the departments list.

```
{* departments_list.tpl *}
{load_presentation_object filename="departments_list" assign="obj"}
{* Start departments list *}
<div class="box">
  <p class="box-title">Choose a Department</p>
  <ul>
  {* Loop through the list of departments *}
  {section name=i loop=$obj->mDepartments}
    {assign var=selected value=""}
    {* Verify if the department is selected to decide what CSS style
       to use *}
    {if ($obj->mSelectedDepartment ==
         $obj->mDepartments[i].department_id)}
      {assign var=selected value="class=\"selected\""}
    {/if}
    <li>
      {* Generate a link for a new department in the list *}
      <a {$selected} href="{$obj->mDepartments[i].link_to_department}">
        {$obj->mDepartments[i].name}
      </a>
    </li>
  {/section}
  </ul>
</div>
{* End departments list *}
```

4. Create a folder named `smarty_plugins` in the `presentation` folder. This will contain the Smarty plug-in files.

5. Inside the `smarty_plugins` folder, create a file named `function.load_presentation_object.php`, and add the following code to it:

```
<?php
// Plug-in functions inside plug-in files must be named: smarty_type_name
function smarty_function_load_presentation_object($params, $smarty)
{
  require_once PRESENTATION_DIR . $params['filename'] . '.php';

  $className = str_replace(' ', '',
                        ucfirst(str_replace('_', ' ',
                                          $params['filename'])));

  // Create presentation object
  $obj = new $className();

  if (method_exists($obj, 'init'))
  {
    $obj->init();
  }

  // Assign template variable
  $smarty->assign($params['assign'], $obj);
}
?>
```

6. Inside the `presentation` folder, create a file named `departments_list.php`, and add the following code to it:

```
<?php
// Manages the departments list
class DepartmentsList
{
  /* Public variables available in departments_list.tpl Smarty template */
  public $mSelectedDepartment = 0;
  public $mDepartments;

  // Constructor reads query string parameter
  public function __construct()
  {
    /* If DepartmentId exists in the query string, we're visiting a
       department */
    if (isset ($_GET['DepartmentId']))
      $this->mSelectedDepartment = (int)$_GET['DepartmentId'];
  }

  /* Calls business tier method to read departments list and create
     their links */
  public function init()
  {
    // Get the list of departments from the business tier
    $this->mDepartments = Catalog::GetDepartments();

    // Create the department links
    for ($i = 0; $i < count($this->mDepartments); $i++)
      $this->mDepartments[$i]['link_to_department'] =
        'index.php?DepartmentId=' . $this->mDepartments[$i]['department_id'];
  }
}
?>
```

7. Modify the `index.php` file to include a reference to the `Catalog` business tier class:

```
// Load the application page template
require_once PRESENTATION_DIR . 'application.php';

// Load the database handler
require_once BUSINESS_DIR . 'database_handler.php';

// Load Business Tier
require_once BUSINESS_DIR . 'catalog.php';

// Load Smarty template file
$application = new Application();
```

8. Make the following modification in `presentation/templates/store_front.tpl` to load the newly created `departments_list` componentized template. Search for the following code:

```
        <div class="yui-b">
          Place list of departments here
        </div>
```

and replace it with this:

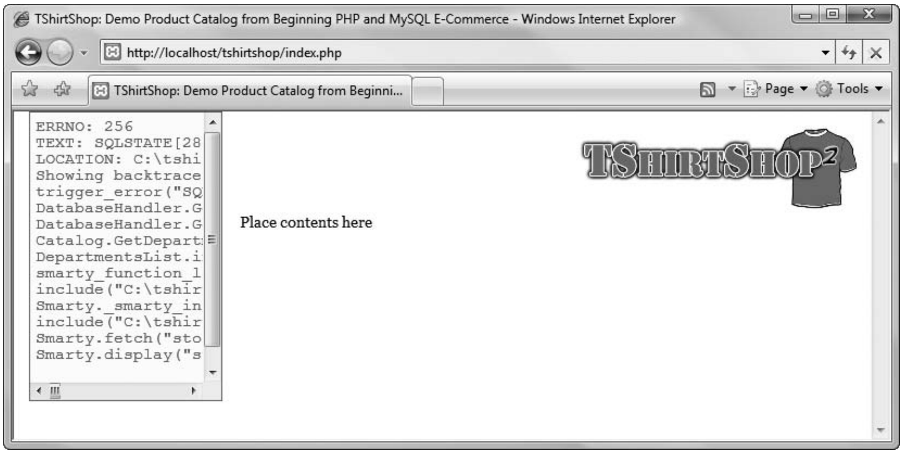
```
<div class="yui-b">
  {include file="departments_list.tpl"}
</div>
```

9. Examine the result of your work with your favorite browser by loading `http://localhost/tshirtshop/index.php` (refer to Figure 4-13). Play a little with the page to see what happens when you click a department or place the mouse over a link.

■ **Note** If you don't get the expected output, make sure your machine is configured correctly and all PHP required modules, such as PDO, were loaded successfully. Many errors will be reported in the Apache error log file (by default, `C:/xampp/apache/logs/error.log` on Windows or `/opt/lampp/logs/error_log` on Linux). Also, make sure to check the book's errata page, which we'll keep updated with solutions to potential problems you may run into.

How It Works: The departments_list Smarty Template

If the page worked as expected from the start, you're certainly one lucky programmer! Most of the time, errors happen because of typos, so watch out for them! Database access problems are also common, so make sure you correctly configured the `tshirtshop` database and the `tshirtshopadmin` user, as shown in Chapter 3. In any case, we're lucky to have a good error-reporting mechanism, which shows a detailed error report if something goes wrong. Figure 4-14 shows the error message I received when mistyping the database password in `config.php`. The error message shows up in the box that generated it (to be able to read the message, you need to select it in the box it was generated, and paste it in another document).


TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/index.php
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
TSHIRTSHOP2

```
ERRNO: 256
TEXT: SQLSTATE[28
LOCATION: C:\tshi
Showing backtrace
trigger_error("SQ
DatabaseHandler.G
DatabaseHandler.G
Catalog.GetDepart
DepartmentsList.i
smarty_function_l
include("C:\tshir
Smarty._smarty_in
include("C:\tshir
Smarty.fetch("sto
Smarty.display("s
```

Place contents here

Figure 4-14. *The error-handling code you've written in Chapter 2 is helpful for debugging.*

If everything goes right, however, you'll get the neat page containing a list of departments generated using a Smarty template. Each department name in the list is a link to the department's page, which, in fact, is a link to the `index.php` page with a `DepartmentId` parameter in the query string that specifies which department was selected. Here's an example of such a link:

```
http://localhost/tshirtshop/index.php?DepartmentId=3
```

When clicking a department's link, the selected department will be displayed using a different CSS style in the list (see Figure 4-15).

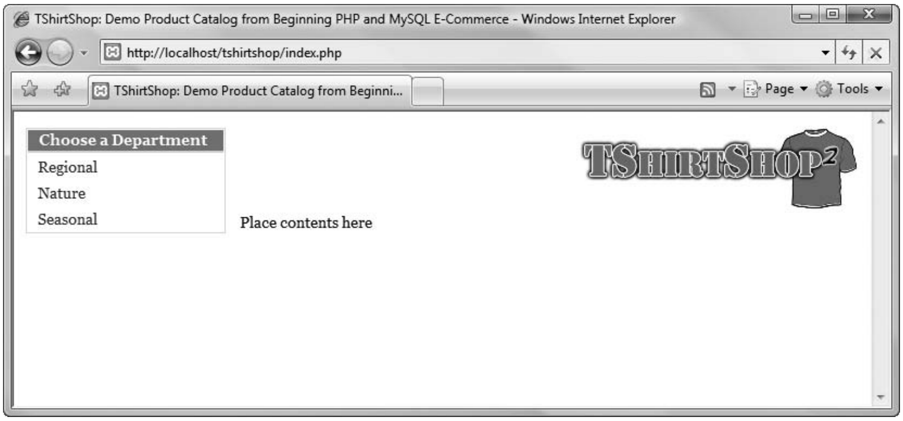

TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/index.php
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
Choose a Department
Regional
Nature
Seasonal
Place contents here
TSHIRTSHOP2

Figure 4-15. *Selecting a department*

It is important to understand how the Smarty template file (`presentation/templates/departments_list.tpl`) and the plug-in file (`presentation/smarty_plugins/function.load_presentation_object.php`) work together to generate the list of departments and to use the correct style for the currently selected one.

The processing starts at `function.load_presentation_object.php`, which is included in the `store_front.tpl` file. The first line in `departments_list.tpl` loads the `DepartmentList` presentation object through the Smarty plug-in:

```
{load_presentation_object filename="departments_list" assign="obj"}
```

The `smarty_function_load_presentation_object()` plug-in function creates and initializes a `DepartmentsList` object (this class is included in `presentation/departments_list.php`), which is then assigned to the `obj` variable accessible from the Smarty design template file:

```
// Plug-in functions inside plug-in files must be named: smarty_type_name
function smarty_function_load_presentation_object($params, $smarty)
{
  require_once PRESENTATION_DIR . $params['filename'] . '.php';

  $className = str_replace(' ', '',
                           ucfirst(str_replace('_', ' ',
                                               $params['filename'])));

  // Create presentation object
  $obj = new $className();
  if (method_exists($obj, 'init'))
  {
    $obj->init();
  }

  // Assign template variable
  $smarty->assign($params['assign'], $obj);
}
```

The `init()` method in `DepartmentsList` populates a public member of the class (`$mDepartments`) with an array containing the list of departments and another public member containing the index of the currently selected department (`$mSelectedDepartment`).

Back to the Smarty code now—inside the HTML code that forms the layout of the Smarty template (`presentation/templates/departments_list.tpl`), you can see the Smarty tags that do the magic:

```
  {* Loop through the list of departments *}
  {section name=i loop=$obj->mDepartments}
    {assign var=selected value=""}
    {* Verify if the department is selected to decide what CSS style
       to use *}
    {if ($obj->mSelectedDepartment == $obj->mDepartments[i].department_id)}
      {assign var=selected value="class=\"selected\""}
    {/if}
    <li>
      {* Generate a link for a new department in the list *}
      <a {$selected} href="{$obj->mDepartments[i].link_to_department}">
        {$obj->mDepartments[i].name}
      </a>
    </li>
  {/section}
```

Smarty template sections are used for looping over arrays of data. In this case, you want to loop over the departments array kept in `$obj->mDepartments`:

```
{section name=i loop=$obj->mDepartments}
  ...
{/section}
```

Inside the loop, you verify whether the current department in the loop (`$obj->mDepartments[i].department_id`) has the `ID` that was mentioned in the query string (`$obj->mSelectedDepartment`). Depending on this, you decide what style to apply to the name by saving the style name (`selected` or default style) to a variable named `selected`.

This variable is then used to generate the link:

```
      <a {$selected} href="{$obj->mDepartments[i].link_to_department}">
        {$obj->mDepartments[i].name}
      </a>
```

Creating the Link Factory

In a well-constructed web site, all the links must have a consistent format. For example, in PHP, you read query string parameters by name rather than ordinal, so the following two links would normally have the same output:

- `http://localhost/index.php?DepartmentId=3&CategoryId=5`
- `http://localhost/index.php?CategoryId=5&DepartmentId=3`

In many cases, the case of parts of an URL can be changed without affecting the output. To have all the URLs in our web site follow a consistent style, we'll create a class that creates links.

This class will prove to be very useful in the long term. In Chapter 7, we'll update the URLs in our web site, so that they will be more friendly to search engines and human visitors browsing your site. Having a central place that generates links will make this feature easy to implement.

Also, at some point in the development process, you'll want certain pages of your site to be accessible only through secured HTTPS connections to ensure the confidentiality of the data passed from the client to the server and back. Such sensitive pages include user login forms, pages where the user enters credit card data, and so on. We don't get into much detail here. However, what you do need to know is that pages accessed through HTTPS occupy much of a server's resources, and we only want to use a secure connection when visiting secure pages. Once again, the link factory can come in handy, as it can be configured to generate HTTPS links only for the sections of the web site that need increased security.

Our link factory will always generate absolute links. Most of the time, it's more comfortable to use relative links inside the web site. For example, it's typical for the header image of a site to contain a link to `index.php` rather than the full URL, such as `http://www.example.com/index.php`. In this case, clicking the header image from a secured page would redirect the user to `https://www.example.com/index.php`, so the visitor would end up accessing through a secure connection a page that isn't supposed to be accessed like that (and, in effect, consumes much more server resources than necessary).

To avoid this problem and other similar ones, we'll write a bit of code that makes sure all the links in the web site are absolute links.

Exercise: Creating the Link Factory

1. Create a new file named `link.php` in the `presentation` folder, and add the following code to it:

```
<?php
class Link
{
  public static function Build($link)
  {
    $base = 'http://' . getenv('SERVER_NAME');
    // If HTTP_SERVER_PORT is defined and different than default
    if (defined('HTTP_SERVER_PORT') && HTTP_SERVER_PORT != '80')
    {
      // Append server port
      $base .= ':' . HTTP_SERVER_PORT;
    }

    $link = $base . VIRTUAL_LOCATION . $link;

    // Escape html
    return htmlspecialchars($link, ENT_QUOTES);
  }

  public static function ToDepartment($departmentId)
  {
    $link = 'index.php?DepartmentId=' . $departmentId;

    return self::Build($link);
  }
}
?>
```

2. Add two new constants to include/config.php:

```
// Server HTTP port (can omit if the default 80 is used)
define('HTTP_SERVER_PORT', '80');
/* Name of the virtual directory the site runs in, for example:
   '/tshirtshop/' if the site runs at http://www.example.com/tshirtshop/
   '/' if the site runs at http://www.example.com/ */
define('VIRTUAL_LOCATION', '/tshirtshop/');
```

3. Modify the init() method from the DepartmentsList class in presentation/departments_list.php as shown in the highlighted code:

```
  /* Calls business tier method to read departments list and create
     their links */
  public function init()
  {
    // Get the list of departments from the business tier
    $this->mDepartments = Catalog::GetDepartments();

    // Create the department links
    for ($i = 0; $i < count($this->mDepartments); $i++)
      $this->mDepartments[$i]['link_to_department'] =
        Link::ToDepartment($this->mDepartments[$i]['department_id']);
  }
```

4. Create a new file named `store_front.php` in the `presentation` folder, and add the following code to it. This is the presentation object that we'll use for the `store_front` template, and it builds a link to the main page of the site.

```
<?php
class StoreFront
{
  public $mSiteUrl;

  // Class constructor
  public function __construct()
  {
    $this->mSiteUrl = Link::Build('');
  }
}
?>
```

5. Modify `presentation/templates/store_front.tpl` like this:

```
{* smarty *}
{config_load file="site.conf"}
{load_presentation_object filename="store_front" assign="obj"}
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>{#site_title#}</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link type="text/css" rel="stylesheet"
     href="{$obj->mSiteUrl}styles/tshirtshop.css" />
  </head>
  <body>
    <div id="doc" class="yui-t2">
      <div id="bd">
        <div id="yui-main">
          <div class="yui-b">
            <div id="header" class="yui-g">
              <a href="{$obj->mSiteUrl}">
                <img src="{$obj->mSiteUrl}images/tshirtshop.png"
                 alt="tshirtshop logo" />
              </a>
            </div>
            <div id="contents" class="yui-g">
              Place contents here
            </div>
          </div>
        </div>
        <div class="yui-b">
          {include file="departments_list.tpl"}
        </div>
      </div>
    </div>
  </body>
</html>
```

6. Open `index.php`, and add a reference to the `Link` class as shown in the highlighted code:

```
<?php
// Include utility files
require_once 'include/config.php';
require_once BUSINESS_DIR . 'error_handler.php';

// Sets the error handler
ErrorHandler::SetHandler();

// Load the application page template
require_once PRESENTATION_DIR . 'application.php';
require_once PRESENTATION_DIR . 'link.php';

// Load the database handler
require_once BUSINESS_DIR . 'database_handler.php';
```

7. Load TShirtShop, and make sure it still works as expected. This exercise isn't supposed to alter our existing functionality but to implement an improvement that will prove to be of great help when extending the site in the following chapters.

How It Works: Using the Link Factory

First of all, make sure the new entry you added to `config.php` is configured correctly. If you're running your web site on a different port than the default of 80 (say, if you're using port 8080), make sure you specify the correct port in the `HTTP_SERVER_PORT` constant. Now, let's see how the link factory works. The `Link` presentation object is used as shown by the modifications you've implemented in `store_front.tpl` and `departments_list.tpl`, and it transforms the relative links received as parameters to absolute links.

■ **Note** In case you aren't using the `tshirtshop` alias as explained in Chapter 3, you'll need to modify the `VIRTUAL_LOCATION` constant in `config.php` to reflect the real location of your web application.

Note that the `Build()` method doesn't add the port if the `HTTP_SERVER_PORT` constant isn't defined or if it contains the default port 80:

```
// If HTTP_SERVER_PORT is defined and different than default
if (defined('HTTP_SERVER_PORT') && HTTP_SERVER_PORT != '80')
{
  // Append server port
  $base .= ':' . HTTP_SERVER_PORT;
}
```

However, you should add the HTTP_SERVER_PORT to config.php anyway to make it easier to modify in case you move the application to a server that runs on another port. If HTTP_SERVER_PORT would be, for example, 8080, the links to index.php specified earlier would be transformed to

```
<a href="http//www.example.com:8080/index.php">
```

Summary

This long chapter was well worth the effort when you consider how much theory you've learned and applied to the TShirtShop project! In this chapter, you accomplished the following:

- You created the department table and populated it with data.
- You learned how to access this data from the data tier using PDO and then how to access the data tier method from the business tier.
- You learned how to use PHP 5 exceptions.
- You implemented the user interface using a Smarty template.

In the next chapter, you will finish creating the product catalog by displaying the site's categories and products! After that's accomplished, we'll have the opportunity to review the structure our web site is built on and the theory you've learned so far.