CHAPTER 5

Creating the Product Catalog: Part 2

In the previous chapter, you implemented a selectable list of departments for the TShirtShop web site. However, a product catalog is much more than that list of departments. In this chapter, you'll add many new product catalog features, including displaying product lists and a product details.

Review Figures 4-1, 4-2, and 4-3 to get a visual feeling of the new functionality you'll implement in this chapter. More specifically, in this chapter, you will

- Learn about relational data and the types of relationships that occur among data tables and then create the new data structures in your database.
- Learn how to join related data tables and even more theory about MySQL stored procedures and techniques.
- Complete the business tier to work with the new MySQL stored procedures, send parameters, and pass requested data to the presentation tier.
- Complete the presentation tier to show your visitor details about the catalog's categories, products, and more.

Storing the New Data

Given the new functionality you are adding in this chapter, it's not surprising that you need to add more data tables to the database. However, this isn't just about adding new data tables. You also need to learn about relational data and the relationships that you can implement among the data tables so that you can obtain more significant information from your database.

What Makes a Relational Database

It's no mystery that a database is something that stores data. However, today's modern Relational Database Management Systems (RDBMS), such as MySQL, PostgreSQL, SQL Server, Oracle, DB2, and others, have extended this basic role by adding the capability to store and manage relational data. This is a concept that deserves some attention.

So what does *relational data* mean? It's easy to see that every piece of data ever written in a real-world database is somehow related to some already existing information. Products are related to categories and departments; orders are related to products and customers, and so on. A relational database keeps its information stored in data tables but is also aware of the relationships between the tables.

These related tables form the *relational database*, which becomes an object with a significance of its own, rather than simply being a group of unrelated data tables. It is said that *data* becomes *information* only when we give significance to it, and establishing relations with other pieces of data is an ideal means of doing so.

Look at the product catalog to see what pieces of data it needs and how you can transform this data into information. For the product catalog, you'll need at least three data tables: one for departments, one for categories, and one for products. It's important to note that physically each data table is an independent database object, even if logically it's part of a larger entity—in other words, even though we say that a category *contains* products, the table that contains the products is not inside the table that contains categories. This is not in contradiction with the relational character of the database.

Figure 5-1 depicts a simple representation of three data tables, including some selected sample data.

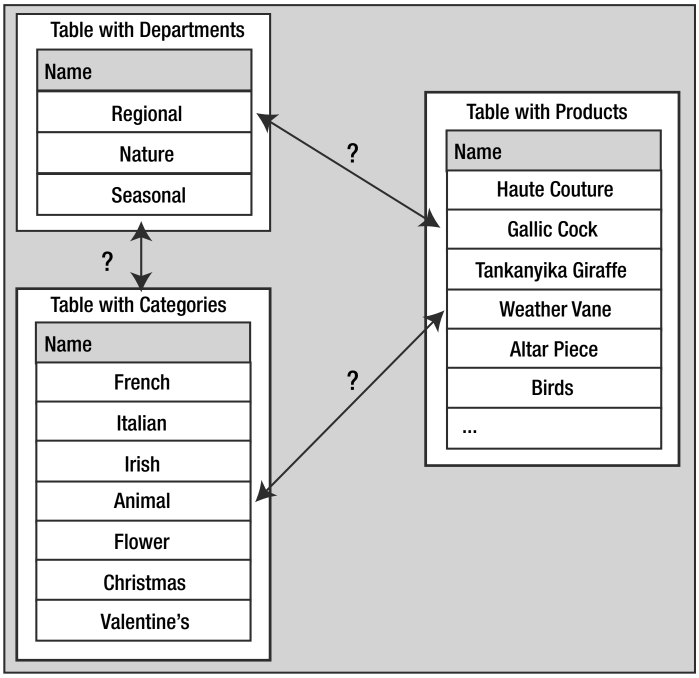


Figure 5-1. *Unrelated departments, categories, and products*

When two tables are said to be related, this more specifically means that the *records* of those tables are related. So, if the products table is related to the categories table, this translates into each product record being somehow related to one of the records in the categories table.

Figure 5-1 doesn't show the physical representation of the database, so we didn't list the table names there. Diagrams like this are used to decide *what* needs to be stored in the database. After you know *what* to store, the next step is to decide *how* the listed data is related, which leads to the physical structure for the database. Although Figure 5-1 shows three kinds of data that you want to store, you'll learn later that to implement this structure in the database, you'll actually use four tables.

So, now that you know the data you want to store, let's think about how the three parts relate to each other. Apart from knowing that the records of two tables are related *somehow*, you also need to know *the kind of relationship* between them. Let's now take a closer look at the different ways in which two tables can be related.

Relational Data and Table Relationships

To continue exploring the world of relational databases, let's further analyze the three logical tables we've been looking at so far. To make life easier, let's give them names now: the table containing products is `product`; the table containing categories is `category`; and the last one is our old friend, `department`. No surprises here! Luckily, these tables implement the most common kinds of relationships that exist between tables, the *one-to-many* and *many-to-many* relationships, so you have the chance to learn about them.

■ **Note** Some variations of these two relationship types exist, as well as the less popular *one-to-one* relationship. In the one-to-one relationship, each row in one table matches exactly one row in the other. For example, in a database that allowed patients to be assigned to beds, you would hope that there would be a one-to-one relationship between patients and beds! Database systems don't support enforcing this kind of relationship, because you would have to add matching records in both tables at the same time. Moreover, two tables with a one-to-one relationship can be joined to form a single table.

One-to-Many Relationships

The one-to-many relationship happens when one record in a table can be associated with multiple records in the related table but not vice versa. In our case, this happens for the `department-category` relation. A specific department can contain any number of categories, but each category belongs to *exactly one* department. Figure 5-2 better represents the one-to-many relationship among departments and categories.

Another common scenario in which you see the one-to-many relationship is with the `order-order_detail` tables, where `order` contains general details about the order (such as date, total amount, and so on) and `order_detail` contains the products related to the order.

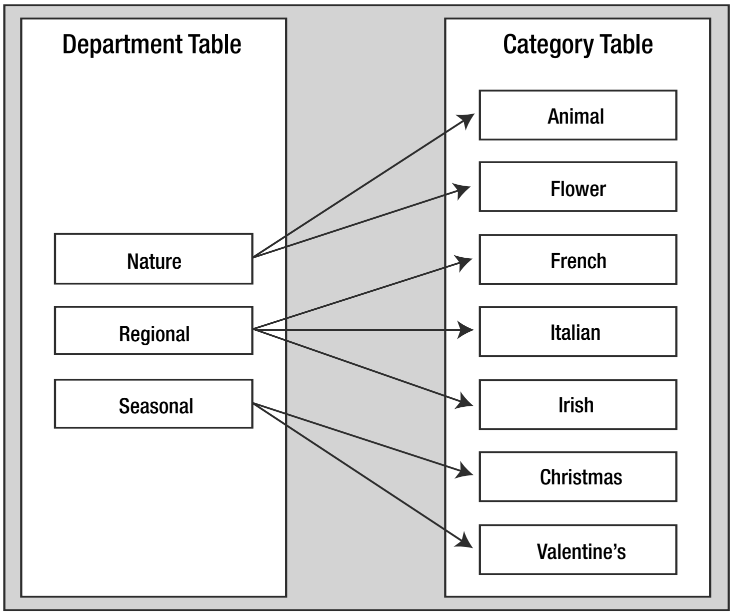


Figure 5-2. *A one-to-many relationship among departments and categories*

The one-to-many relationship is implemented in the database by adding an extra column in the table at the *many* side of the relationship, which references the ID column of the table in the *one* side of the relationship. Simply said, in the `category` table, you'll have an extra column (called `department_id`) that will hold the ID of the department the category belongs to. You'll implement this in your database a bit later, after you learn about the many-to-many relationships and foreign key constraints.

Many-to-Many Relationships

The other common type of relationship is the many-to-many relationship. This kind of relationship is implemented when records in both tables of the relationship can have multiple matching records in the other. In our scenario, this happens for the `product` and `category` tables, because we know that a product can exist in more than one category (*one* product with *many* categories), and a category can have more than one product (*one* category with *many* products).

This happens because we decided earlier that a product could be in more than one category. If a product could only belong to a single category, you would have another one-to-many relationship, just like the one that exists among departments and categories (where a category can't belong to more than one department).

If you represent this relationship with a picture, as shown previously in Figure 5-2, but with generic names this time, you get something like what is shown in Figure 5-3.

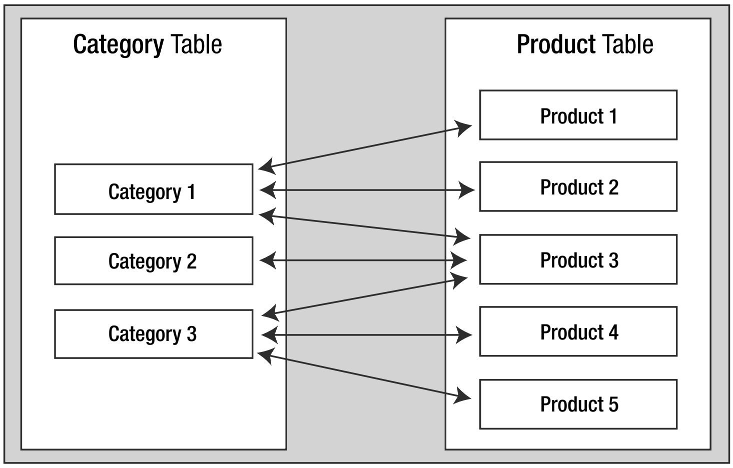


Figure 5-3. *The many-to-many relationship between categories and products*

Although logically the many-to-many relationship happens between two tables, databases (including MySQL databases) don't have the means to physically implement this kind of relationship by using just two tables, so we cheat by adding a third table to the mix. This third table, called a *junction table* (also known as a *linking table* or *associate table*) and two one-to-many relationships will help achieve the many-to-many relationship. The junction table is used to associate products and categories, with no restriction on how many products can exist for a category or to how many categories a product can be added. Figure 5-4 shows the role of the junction table.

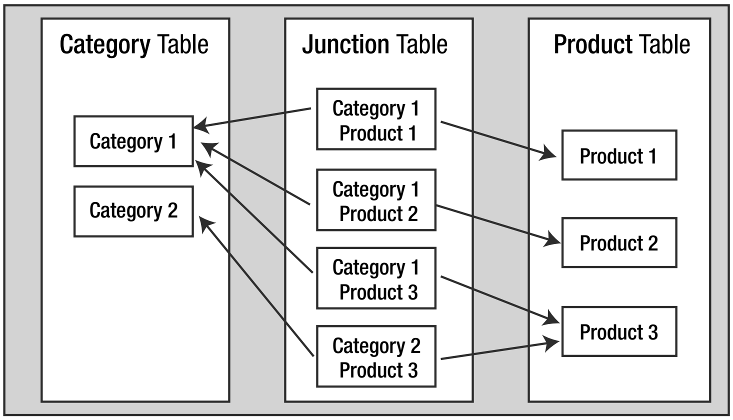


Figure 5-4. *The many-to-many relationship among categories and products*

Note that each record in the junction table links one category with one product. You can have as many records as you like in the junction table, linking any category to any product. The linking table contains two fields, each one referencing the primary key of one of the two linked tables. In our case, the junction table will contain two fields: a `category_id` field and a `product_id` field.

Each record in the junction table will consist of a product and category ID pair (`product_id`, `category_id`), which will be used to associate a particular product with a particular category. By adding more records to the `product_category` table, you can associate a product with more categories or a category with more products, effectively implementing the many-to-many relationship.

Because the many-to-many relationship is implemented using a third table that makes the connection between the linked tables, there is no need to add additional fields to the related tables in the way that we added the `department_id` to the `category` table for implementing the one-to-many relationship.

There's no definitive naming convention to use for the junction table. Most of the time it's OK to just join the names of the two linked tables—in this case, the junction table is named `product_category`.

Enforcing Table Relationships Using Foreign Keys

Relationships among tables can be physically enforced in the database using `FOREIGN KEY` constraints, or simply *foreign keys*.

You learned in the previous chapter about the `PRIMARY KEY` and `UNIQUE` constraints. We covered them there, because they apply to the table as an individual entity. Foreign keys, on the other hand, occur between two tables: the table in which the foreign key is defined (the *referencing table*) and the table the foreign key references (the *referenced table*).

■ **Tip** Actually, the referencing table and the referenced table can be one and the same. This isn't seen too often in practice, but it's not unusual either. For example, you can have a table with employees, where each employee references the employee that is his or her boss (in this scenario, the big boss's row would probably reference itself).

A *foreign key* is a column or combination of columns used to enforce a link between data in two tables (usually representing a one-to-many relationship). Foreign keys are used both as a method of ensuring data integrity and to establish a relationship between tables.

To enforce database integrity, the foreign keys, like the other types of constraints, apply certain restrictions. Unlike `PRIMARY KEY` and `UNIQUE` constraints that apply restrictions to a single table, the `FOREIGN KEY` constraint *applies restrictions on both the referencing and referenced tables*. For example, if you enforce the one-to-many relationship between the `department` and `category` tables by using a `FOREIGN KEY` constraint, the database will include this relationship as part of its integrity. It will not allow you to add a category to a nonexistent department, nor will it allow you to delete a department if there are categories that belong to it.

There's good news and bad news about the `FOREIGN KEY` constraint and MySQL. The bad news is that the default storage engine in most MySQL instances—MyISAM—doesn't support enforcing `FOREIGN KEY` constraints. The alternative to MyISAM is the InnoDB storage engine, but InnoDB tables don't support full-text searching, which will be needed when implementing the search feature (in Chapter 8).

The good news is that you can have different types of tables inside a single database, so you can use MyISAM for tables that don't need free-text searching and/or foreign keys and InnoDB for the others. You must be extra careful when manipulating data from MyISAM tables, however, because you can't rely on the database to enforce its integrity on its own.

■**Note** Foreign keys can be programmatically implemented for the storing engines types that don't support them, with the aid of *triggers*. You can find a good tutorial about this technique at `http://dev.mysql.com/tech-resources/articles/mysql-enforcing-foreign-keys.html`.

Before implementing the rest of the product catalog tables, we need to explain more about the various types of MySQL table types.

MySQL Table Types

MySQL supports several storage engines that can be used to store your data. When creating a new data table, if not specified otherwise, the default table type (MyISAM) is used. Following are three important table types supported by MySQL:

MyISAM is the default storage engine when creating new tables since MySQL 3.23 (when it replaced its older version, ISAM). It is the fastest table type in MySQL, at the cost of not supporting foreign keys, `CHECK` constraints, transactions, and some other advanced features. However, unlike the other table types, it supports full-text searching, which is very helpful when implementing the searching capability in the web site.

InnoDB is a very popular and powerful database engine for MySQL that, among other features, supports transactions, has great capability to handle many simultaneous update operations, and can enforce `FOREIGN KEY` constraints. The engine is developed independently of MySQL, and its home page is `http://www.innodb.com`.

HEAP is a special kind of table type in that it is constructed in system memory. It cannot be used to reliably store data (in case of a system failure, all data is lost and cannot be recovered), but it can be a good choice when working with tables that need to be very fast with data that can be easily reconstructed if accidentally lost.

To learn more about these storage engines, and about the other storage engines supported by MySQL, see the manual page at `http://dev.mysql.com/doc/refman/5.0/en/storage-engines.html`.

■**Note** For the TShirtShop product catalog, you'll be using MyISAM tables mainly because you need their full-text search feature. If you change your mind about the type of a table, you can easily change it with the `ALTER TABLE` command. The following line of code would make the department table an InnoDB table:

```
ALTER TABLE department ENGINE=InnoDB;
```

Creating and Populating the New Data Tables

Now, it's time to create the three new tables to complete our product catalog:

- category
- product
- product_category

Adding Categories

The process of creating the category table is pretty much the same as for the department table you created in Chapter 3. The category table will have four fields, described in Table 5-1.

Table 5-1. *Designing the category Table*

Field Name	Data Type	Description
category_id	int	An autoincrement integer that represents the unique ID for the category. It is the primary key of the table, and it doesn't allow NULLs.
department_id	int	An integer that represents the department the category belongs to. It doesn't allow NULLs.
name	varchar(100)	Stores the category name. It does not allow NULLs.
description	varchar(1000)	Stores the category description. It allows NULLs.

There are two ways to create the category table and populate it: either execute the SQL scripts from the Source Code/Download section of the Apress web site (http://www.apress.com/) or follow the steps in the following exercise.

Exercise: Creating the category Table

1. Using phpMyAdmin, navigate to your tshirtshop database.
2. Click the SQL button in the top menu, and use the form to execute the following SQL query, which creates the category table (alternatively, you can use phpMyAdmin to create the table by specifying the fields using a visual interface as you did in Chapter 4 for the department table).

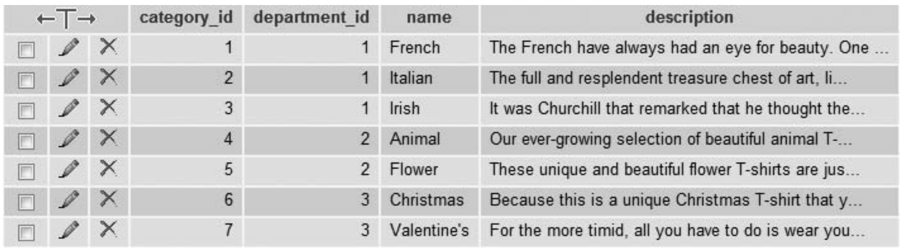
```
-- Create category table
CREATE TABLE 'category' (
  'category_id'   INT            NOT NULL  AUTO_INCREMENT,
  'department_id' INT            NOT NULL,
  'name'          VARCHAR(100)   NOT NULL,
  'description'   VARCHAR(1000),
  PRIMARY KEY ('category_id'),
  KEY 'idx_category_department_id' ('department_id')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

3. Now, let's populate the table with some data. Execute the following SQL script:

```
-- Populate category table
INSERT INTO `category` (`category_id`, `department_id`, `name`,
`description`) VALUES
       (1, 1, 'French', 'The French have always had an eye for beauty. One look
 at the T-shirts below and you''ll see that same appreciation has been applied
 abundantly to their postage stamps. Below are some of our most beautiful and
 colorful T-shirts, so browse away! And don''t forget to go all the way to the
 bottom - you don''t want to miss any of them!'),
       (2, 1, 'Italian', 'The full and resplendent treasure chest of art,
 literature, music, and science that Italy has given the world is reflected
 splendidly in its postal stamps. If we could, we would dedicate hundreds of
 T-shirts to this amazing treasure of beautiful images, but for now we will
 have to live with what you see here. You don''t have to be Italian to love
 these gorgeous T-shirts, just someone who appreciates the finer things in
 life!'),
       (3, 1, 'Irish', 'It was Churchill who remarked that he thought the Irish
 most curious because they didn''t want to be English. How right he was! But
 then, he was half-American, wasn''t he? If you have an Irish genealogy you
will want these T-shirts! If you suddenly turn Irish on St. Patrick''s Day,
you too will want these T-shirts! Take a look at some of the coolest T-shirts
we have!'),
       (4, 2, 'Animal', ' Our ever-growing selection of beautiful animal T-
shirts represents critters from everywhere, both wild and domestic. If you
 don''t see the T-shirt with the animal you''re looking for, tell us and
 we''ll find it!'),
       (5, 2, 'Flower', 'These unique and beautiful flower T-shirts are just
 the item for the gardener, flower arranger, florist, or general lover of
things beautiful. Surprise the flower in your life with one of the beautiful
 botanical T-shirts or just get a few for yourself!'),
       (6, 3, 'Christmas', ' Because this is a unique Christmas T-shirt that
 you''ll only wear a few times a year, it will probably last for decades (unless
 some grinch nabs it from you, of course). Far into the future, after you''re
 gone, your grandkids will pull it out and argue over who gets to wear it. What
 great snapshots they''ll make dressed in Grandpa or Grandma''s incredibly
 tasteful and unique Christmas T-shirt! Yes, everyone will remember you forever
 and what a silly goof you were when you would wear only your Santa beard and
 cap so you wouldn''t cover up your nifty T-shirt.'),
       (7, 3, 'Valentine''s', 'For the more timid, all you have to do is wear
 your heartfelt message to get it across. Buy one for you and your sweetie(s)
 today!');
```

How It Works: Populating the categories Table

Adding data to your table should be a trivial task, given that you know the data that needs to be inserted. As pointed out earlier, you can find the SQL scripts in the book's code in the Source Code/Download section of the Apress web site (`http://www.apress.com`). Figure 5-5 shows data from the category table as shown by phpMyAdmin.



←T→			category_id	department_id	name	description
☐	✎	✕	1	1	French	The French have always had an eye for beauty. One ...
☐	✎	✕	2	1	Italian	The full and resplendent treasure chest of art, li...
☐	✎	✕	3	1	Irish	It was Churchill that remarked that he thought the...
☐	✎	✕	4	2	Animal	Our ever-growing selection of beautiful animal T-...
☐	✎	✕	5	2	Flower	These unique and beautiful flower T-shirts are jus...
☐	✎	✕	6	3	Christmas	Because this is a unique Christmas T-shirt that y...
☐	✎	✕	7	3	Valentine's	For the more timid, all you have to do is wear you...

Figure 5-5. *Data from the category table*

In the SQL code, note how we escaped the special characters in the category descriptions, such as the single quotes, that need to be doubled, so MySQL will know to interpret those as quotes to be added to the description, instead of as string termination characters.

Adding Products and Relating Them to Categories

You'll now go through the same steps as earlier, but this time, you'll create a slightly more complicated table: `product`. The `product` table has the fields shown in Table 5-2.

Table 5-2. *Designing the product Table*

Field Name	Data Type	Description
`product_id`	`int`	An integer that represents the unique ID for the category. It is the primary key of the table and an autoincrement field.
`name`	`varchar(100)`	Stores the product name. It doesn't allow `NULL`s.
`description`	`varchar(1000)`	Stores the category description. It allows `NULL`s.
`price`	`numeric(10, 2)`	Stores the product price.
`discounted_price`	`numeric(10, 2)`	Stores the discounted product price. Will store `0.00` if the product doesn't have a current discount price.
`image`	`varchar(150)`	Stores the name of the product's picture file (or eventually the complete path), which gets displayed on the product details page. You could keep the picture directly in the table, but in most cases, it's much more efficient to store the picture files in the file system and have only their names stored into the database. If you have a high-traffic web site, you might even want to place the image files in a separate physical location (for example, another hard disk) to increase site performance. This field allows `NULL`s.
`image_2`	`varchar(150)`	Stores the name of a second picture of the product, which gets displayed on the product details page. It allows `NULL`s.

Field Name	Data Type	Description
thumbnail	varchar(150)	Stores the name of the product's thumbnail picture. This image gets displayed in product lists when browsing the catalog.
display	smallint	Stores a value specifying in what areas of the catalog this product should be displayed. The possible values are 0 (default), meaning the product is listed only in the page of the category it's a part of); 1, which indicates that the product is also featured on the front catalog page; 2, indicating the product is also featured in the departments it's a part of; and 3, which means the product is also featured on both the front and the department pages. With the help of this field, the site administrators can highlight a set of products that will be of particular interest to visitors at a specific season, holiday, and so on. Also, if you want to promote products that have a discounted price, this feature is just what you need.

The product_category table is the linking table that allows implementing the many-to-many relationship between the product and category tables. It has two fields that form the primary key of the table: product_id and category_id.

Follow the steps of the exercise to create the product table in your database.

Exercise: Creating the product Table

1. Using phpMyAdmin, execute the following command to create the product table:

```
-- Create product table
CREATE TABLE `product` (
  `product_id`       INT            NOT NULL AUTO_INCREMENT,
  `name`             VARCHAR(100)   NOT NULL,
  `description`      VARCHAR(1000)  NOT NULL,
  `price`            NUMERIC(10, 2) NOT NULL,
  `discounted_price` NUMERIC(10, 2) NOT NULL DEFAULT 0.00,
  `image`            VARCHAR(150),
  `image_2`          VARCHAR(150),
  `thumbnail`        VARCHAR(150),
  `display`          SMALLINT       NOT NULL DEFAULT 0,
  PRIMARY KEY (`product_id`)
) ENGINE=MyISAM;
```

2. Now, create the product_category table by executing this query:

```
-- Create product_category table
CREATE TABLE `product_category` (
  `product_id`  INT NOT NULL,
  `category_id` INT NOT NULL,
  PRIMARY KEY (`product_id`, `category_id`)
) ENGINE=MyISAM;
```

3. Use the `populate_product.sql` script from the code download to populate the `product` table with sample data.
4. Use the `populate_product_category.sql` script from the code download to populate the `product_category` table with sample data.

How It Works: Many-to-Many Relationships

Many-to-many relationships are created by adding a third table, called a junction table, which is named `product_category` in this case. This table contains `product_id` and `category_id` pairs, and each record in the table associates a particular product with a particular category. So, if you see a record such as (`1`, `4`) in `product_category`, you know that the product with the ID of `1` belongs to the category with the ID of `4`.

This is also the first time that you set a primary key consisting of more than one column. The primary key of `product_category` is formed by both its fields: `product_id` and `category_id`. This means that you won't be allowed to have two identical (`product_id`, `category_id`) pairs in the table. However, it's perfectly legal to have a `product_id` or `category_id` appear more than once, as long as it is part of a unique (`product_id`, `category_id`) pair. This makes sense because you don't want to have two identical records in the `product_category` table. A product can be associated with a particular category or not; it cannot be associated with a category multiple times.

Using Database Diagrams

All the theory about table relationships can be a bit confusing at first, but you'll get used to it. To understand the relationship more clearly, you can get a picture by using database diagrams.

A number of tools allow you to build database structures visually, implement them physically in the database for you, and generate the necessary SQL script. Although we won't present any particular tool in this book, it's good to know that they exist. You can find a list of the most popular tools at `http://www.databaseanswers.com/modelling_tools.htm`.

Database diagrams also have the capability to implement the relationships between tables. For example, if you had implemented the relationships among your four tables so far, the database diagram would look something like Figure 5-6.

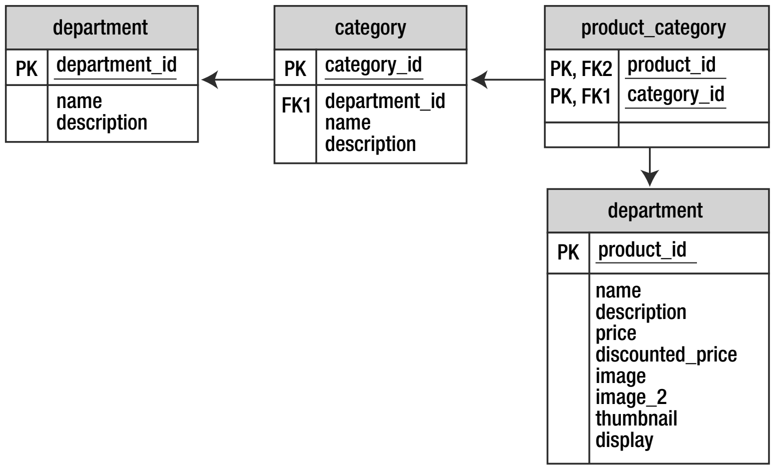


Figure 5-6. *Viewing tables and relationships using a database diagram*

In the diagram, the primary keys of each table are marked with "PK." Foreign keys are marked with "FK" (because there can be more of them in a table, they're numbered). The arrows between two tables point toward the table in the *one* part of the relationship.

Querying the New Data

Now, you have a database with a wealth of information just waiting to be read by somebody. However, the new elements bring with them a set of new things you need to learn.

For this chapter, the data tier logic is a little bit more complicated than in the previous chapter, because it must answer to queries like "give me the second page of products from the Cartoons category" or "give me the products on promotion for department *X*." Before moving on to writing the stored procedures that implement this logic, let's first cover the theory about

- Retrieving short product descriptions
- Joining data tables
- Implementing paging

Let's deal with these tasks one by one.

Getting Short Descriptions

In the product lists that your visitor sees while browsing the catalog, we won't display full product descriptions, only a portion of them. In TShirtShop, we'll display the first 150 characters of every product description, after which, if the description has a greater length, we concatenate (append) ellipses (. . .) to the end of the description. Of course, you can decide if you would like more or less of the description to display by simply changing this length (150 in our design) to whatever number you choose; be sure to verify that this displays well by previewing your changes in your browser of choice.

We'll use the `LEFT(str, len)` MySQL function to extract the first *N* (`len`) characters from the product description, where *N* is the number of characters to be extracted. The following `SELECT` command returns products' descriptions trimmed at 30 characters, with ". . ." appended to the end if the description has a length greater than 30 characters:

```
SELECT   name,
         IF(LENGTH(description) <= 30, description,
                   CONCAT(LEFT(description, 30), '...')) AS description
FROM     product
ORDER BY name;
```

The new column generated by the `CONCAT(LEFT(description, 30), '...')` expression doesn't have a name by default, so we create an alias for it using the `AS` keyword. With your current data, this query would return something like this:

```
name                                    description
A Partridge in a Pear Tree              The original of this beautiful...
Adoration of the Kings                  This design is from a miniatur...
Afghan Flower                           This beautiful image was issue...
Albania Flower                          Well, these crab apples starte...
Alsace                                  It was in this region of Franc...
...                                     ...
```

Joining Data Tables

Because your data is stored in several tables, all of the information you'll need might not be one table. Take a look at the following list, which contains data from both the `department` and `category` tables:

```
Department Name                            Category Name
-----------------------------------------------------------------------------------
Regional                                   French
Regional                                   Italian
Regional                                   Irish
Nature                                     Animal
Nature                                     Flower
Seasonal                                   Christmas
Seasonal                                   Valentine's
```

In other cases, all the information you need is in just one table, but you need to place conditions on it based on the information in another table. You cannot get this kind of result set with simple queries such as the ones you've used so far. Needing a result set based on data from multiple tables is a good indication that you might need to use *table joins*.

When extracting the products that belong to a category, the SQL query isn't the same as when extracting the categories that belong to a department. This is because products and categories are linked through the `product_category` linking table.

To get the list of products in a category, you first need to look in the `product_category` table and get all the (`product_id`, `category_id`) pairs where `category_id` is the ID of the category you're looking for. That list contains the IDs of the products in that category. Using these IDs, you'll be able to generate the required product list from the product table. Although this sounds pretty complicated, it can be done using a single SQL query. The real power of SQL lies in its capability to perform complex operations on large amounts of data using simple queries.

Joining one table with another results in the columns (not the rows) of those tables being joined. When joining two tables, there always must be a common column on which the join will be made. Tables are joined in SQL using the `JOIN` clause. You'll learn how to make table joins by analyzing the product and `product_category` tables and by analyzing how you can get a list of products that belong to a certain category.

Suppose you want to get all the products where `category_id = 5`. The query that joins the `product` and `product_category` tables is as follows:

```
SELECT     product_category.product_id,
           product_category.category_id,
           product.name
```

```
FROM       product_category
INNER JOIN product
        ON product.product_id = product_category.product_id
ORDER BY   product.product_id;
```

The result will look something like this (to save space, the listing doesn't include all returned rows and columns):

```
product_id                   category_id                  name
---------------------------------------------------------------------------------
1                            1                            Arc d'Triomphe
2                            1                            Chartres Cathedral
3                            1                            Coat of Arms
4                            1                            Gallic Cock
5                            1                            Marianne
6                            1                            Alsace
7                            1                            Apocalypse Tapestry
8                            1                            Centaur
9                            1                            Corsica
10                           1                            Haute Couture
11                           1                            Iris
12                           1                            Lorraine
13                           1                            Mercury
14                           1                            County of Nice
15                           1                            Notre Dame
16                           1                            Paris Peace Conference
17                           1                            Sarah Bernhardt
18                           1                            Hunt
19                           2                            Italia
20                           2                            Torch
...
```

The resultant table is composed of the requested fields from the joined tables synchronized on the `product_id` column, which was specified as the column to make the join on. You can see that the products that exist in multiple categories are listed more than once, once for each category they belong in, but this problem will go away after we filter the results to get only the products for a certain category.

Note that in the `SELECT` clause, the column names are prefixed by the table name. *This is a requirement if the columns exist in more tables that participate in the table join*, such as `product_id` in our case. For the other column, prefixing its name with the table name is optional, although it's a good practice to avoid confusion.

The query that returns only the products that belong to category 5 is

```
SELECT     product.product_id, product.name
FROM       product_category
```

```
INNER JOIN product
        ON product.product_id = product_category.product_id
WHERE      product_category.category_id = 5;
```

The results follow:

```
product_id              name
-----------------------------------------------------------------------------------
65                      Afghan Flower
66                      Albania Flower
67                      Austria Flower
68                      Bulgarian Flower
```

A final thing worth discussing here is the use of *aliases*. Aliases aren't necessarily related to table joins, but they become especially useful (and sometimes necessary) when joining tables, and they assign different (usually) shorter names for the tables involved. Aliases are necessary when joining a table with itself, in which case, you need to assign different aliases for its different instances to differentiate them. The following query returns the same products as the query before, but it uses aliases:

```
SELECT     p.product_id, p.name
FROM       product_category pc
INNER JOIN product p
        ON p.product_id = pc.product_id
WHERE      pc.category_id = 5;
```

Showing Products Page by Page

If certain web sections need to list large numbers of products, it's useful to let the visitor browse them page by page, with a predefined (or configurable by the visitor) number of products per page.

Depending on the tier on your architecture where paging is performed, there are three main ways to implement paging:

Paging at the data tier level: In this case, the database returns only a single page of products, the page needed by the presentation tier.

Paging at the business tier level: The business tier requests the complete page of products from the database, performs filtering, and returns to the presentation tier only the page of products that needs to be displayed.

Paging at the presentation tier level: In this scenario, the presentation tier receives the complete list of products and extracts only the page that needs to be displayed for the visitor.

Paging at the business tier and presentation tier levels has potential performance problems, especially when dealing with large result sets, because they imply transferring unnecessarily large quantities of data from the database to the presentation tier. Additional data also needs to be stored on the server's memory, unnecessarily consuming server resources.

In our web site, we'll implement paging at the data tier level, not only because of its better performance but also because it allows you to learn some tricks about database programming that you'll find useful when developing your web sites.

To implement paging at the data tier level, we need to know how to build a SELECT query that returns just a portion of records (products) from a larger set, and each database language seems to have different ways of doing this. To achieve this functionality in MySQL, you need to use the LIMIT keyword with the SELECT statement. LIMIT takes one or two arguments. The first argument specifies the index of the first returned record, and the second specifies how many rows to return.

The following SQL query tells MySQL to return the rows 15, 16, 17, 18, and 19 from the list of products ordered by their IDs (remember that an index starts at zero for the first record, so we ask for *row 15* by asking for the *index 14*—the row number minus one):

```
SELECT   name
FROM     product
ORDER BY product_id
LIMIT    14, 5;
```

With the current database you should get these results:

```
name
----------------------------------------------------------------------------------
Notre Dame
Paris Peace Conference
Sarah Bernhardt
Hunt
Italia
```

You'll use the LIMIT keyword to specify the range of records you're interested in when retrieving lists of products. For more details, you can always refer to the official documentation at http://dev.mysql.com/doc/refman/5.1/en/select.html.

Writing the New Database Stored Procedures

Now that you are familiar with how to query the database, you can implement the data tier stored procedures, which will return data from the database. First, you'll implement the MySQL stored procedures that retrieve department and category information:

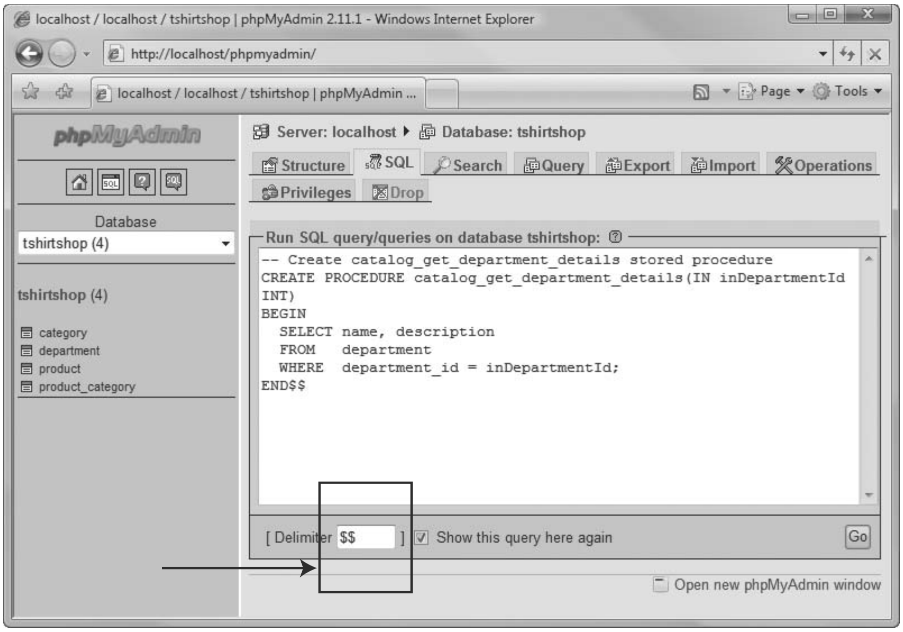
- catalog_get_department_details
- catalog_get_categories_list
- catalog_get_category_details

Afterward, you'll write the stored procedures that deal with products. Only four stored procedures ask for products, but you'll also implement three helper functions (catalog_count_products_in_category, catalog_count_products_on_department, and catalog_count_products_on_catalog) to assist in implementing the paging functionality. The complete list of stored procedures you need to implement follows:

- `catalog_count_products_in_category`
- `catalog_get_products_in_category`
- `catalog_count_products_on_department`
- `catalog_get_products_on_department`
- `catalog_count_products_on_catalog`
- `catalog_get_products_on_catalog`
- `catalog_get_product_details`
- `catalog_get_product_locations`

Notice that we have named our stored procedures in such a way that the name tells you what the stored procedure does. This is called *self-documenting* or *self-describing* code and is a good habit to form when coding; it will help you—or anyone else who needs to work on your site—to quickly understand what each of your stored procedures does without having to look inside them. Many a programmer has learned the hard way that two weeks from now you will not remember what a particular function does, and naming it `function_one` tells you absolutely nothing about its purpose; following a self-documenting style will, in the long run, save hours during debugging and redesign.

In the following sections, you'll be shown the code of each stored procedure. We won't go though individual exercises to create these stored procedures. Use phpMyAdmin to add them to your database, using the SQL tab *and changing the* `DELIMITER` *to* `$$`, as shown in Figure 5-7.


localhost / localhost / tshirtshop | phpMyAdmin 2.11.1 - Windows Internet Explorer
http://localhost/phpmyadmin/
localhost / localhost / tshirtshop | phpMyAdmin ...
Page
Tools
phpMyAdmin
Server: localhost
Database: tshirtshop
Structure
SQL
Search
Query
Export
Import
Operations
Privileges
Drop
Database
tshirtshop (4)
tshirtshop (4)
category
department
product
product_category
Run SQL query/queries on database tshirtshop:
Show this query here again
Go
Open new phpMyAdmin window

```
-- Create catalog_get_department_details stored procedure
CREATE PROCEDURE catalog_get_department_details(IN inDepartmentId
INT)
BEGIN
  SELECT name, description
  FROM   department
  WHERE  department_id = inDepartmentId;
END$$
```

[Delimiter $$]

Figure 5-7. *Changing the SQL delimiter in phpMyAdmin*

Caution Don't forget to set the delimiter when you create *each* stored procedure!

catalog_get_department_details

The `catalog_get_department_details` stored procedure returns the name and description for a given department whose ID is received as parameter. This is needed when the user selects a department in the product catalog, and the database must be queried to find out the name and the description of the particular department.

Next is the SQL code that creates the `catalog_get_department_details` stored procedure. Execute it using phpMyAdmin. *Don't forget to set the* `$$` *delimiter!*

```
-- Create catalog_get_department_details stored procedure
CREATE PROCEDURE catalog_get_department_details(IN inDepartmentId INT)
BEGIN
  SELECT name, description
  FROM   department
  WHERE  department_id = inDepartmentId;
END$$
```

As you can see, a stored procedure is very much like a PHP function, in that it receives parameters, executes some code, and returns the results. In this case, we have a single input (`IN`) parameter named `inDepartmentId`, whose data type is `int`. Take note of the naming convention we've chosen for parameter names; the name of the parameter includes the parameter direction (`in`), and uses camel casing.

MySQL also supports output (`OUT`) parameters and input-output (`INOUT`) parameters. You'll see examples of using these a little later. The official documentation page for the `CREATE PROCEDURE` command, which contains the complete details about using parameters with stored procedures, is located at `http://dev.mysql.com/doc/refman/5.1/en/create-procedure.html`.

The `catalog_get_department_details` stored procedure returns the names and descriptions of all departments whose `department_ids` match the one specified by the `inDepartmentId` input parameter. The `WHERE` clause (`WHERE department_id = inDepartmentId`) is used to request the details of that specific department.

catalog_get_categories_list

When a visitor selects a department, the categories that belong to that department must be displayed. The categories will be retrieved by the `catalog_get_categories_list` stored procedure, which returns the list of categories in a specific department. The stored procedure needs to know the ID of the department for which to retrieve the categories.

```
-- Create catalog_get_categories_list stored procedure
CREATE PROCEDURE catalog_get_categories_list(IN inDepartmentId INT)
BEGIN
  SELECT   category_id, name
  FROM     category
  WHERE    department_id = inDepartmentId
  ORDER BY category_id;
END$$
```

catalog_get_category_details

When the visitor selects a particular category, we need to display its name and description. Execute this code, *using the* `$$` *delimiter*, to create the procedure:

```
-- Create catalog_get_category_details stored procedure
CREATE PROCEDURE catalog_get_category_details(IN inCategoryId INT)
BEGIN
  SELECT name, description
  FROM   category
  WHERE  category_id = inCategoryId;
END$$
```

catalog_count_products_in_category

This function returns the number of products in a category. This data will be necessary when paginating the lists of products, and we'll need to be able to calculate how many pages of products we have in a category.

Note that, unlike the previous procedures you've written, this time, we return a single value rather than a set of data. That value is calculated using `COUNT`, which returns the number of records that match a particular query.

```
-- Create catalog_count_products_in_category stored procedure
CREATE PROCEDURE catalog_count_products_in_category(IN inCategoryId INT)
BEGIN
  SELECT     COUNT(*) AS categories_count
  FROM       product p
  INNER JOIN product_category pc
               ON p.product_id = pc.product_id
  WHERE      pc.category_id = inCategoryId;
END$$
```

catalog_get_products_in_category

This stored procedure returns the products that belong to a certain category. To obtain this list of products, you need to join the `product` and `product_category` tables, as explained earlier in this chapter. We also trim the product's description.

The stored procedure receives four parameters:

- `inCategoryID` is the ID for which we're returning products.
- `inShortProductDescriptionLength` is the maximum length allowed for the product's description. If the description is longer than this value, it will be truncated and "..." will be added at the end. Note that this is only used when displaying product lists; in the product details page, the description won't be truncated. You will declare the global value of this variable later on in `config.php`, but for now, it is enough to know that it exists and what its value will be set to: in our case 150 characters.

- `inProductsPerPage` is the maximum number of products our site can display on a single catalog page. If the total number of products in the category is larger than this number, we return only a page containing the number of `inProductsPerPage` products. Again, this variable's value will be declared globally in `config.php` later on; for now, you only need to know that for our project we will set this to `4`.

- `inStartItem` is the index of the first product to return. So, for example, when the visitor visits the *second page* of products—using pagination and displaying four products per page—`inStartItem` will be `4` and `inProductsPerPage` will be `4`. With these values, the `catalog_get_products_in_category` function will return the products from the fifth to ninth rows of results.

```
-- Create catalog_get_products_in_category stored procedure
CREATE PROCEDURE catalog_get_products_in_category(
  IN inCategoryId INT, IN inShortProductDescriptionLength INT,
  IN inProductsPerPage INT, IN inStartItem INT)
BEGIN
  -- Prepare statement
  PREPARE statement FROM
   "SELECT     p.product_id, p.name,
               IF(LENGTH(p.description) <= ?,
                  p.description,
                  CONCAT(LEFT(p.description, ?),
                         '...')) AS description,
               p.price, p.discounted_price, p.thumbnail
    FROM       product p
    INNER JOIN product_category pc
                 ON p.product_id = pc.product_id
    WHERE      pc.category_id = ?
    ORDER BY   p.display DESC
    LIMIT      ?, ?";

  -- Define query parameters
  SET @p1 = inShortProductDescriptionLength;
  SET @p2 = inShortProductDescriptionLength;
  SET @p3 = inCategoryId;
  SET @p4 = inStartItem;
  SET @p5 = inProductsPerPage;

  -- Execute the statement
  EXECUTE statement USING @p1, @p2, @p3, @p4, @p5;
END$$
```

In this procedure, you can find a demonstration of using *prepared statements*, which represent query templates that contain input parameters whose values you supply right before executing the statement.

The reason we need to use prepared statements is that they allow adding parameters to the LIMIT clause of a SELECT query. MySQL 5, at the time of this writing, doesn't allow using an input parameter to set the value of LIMIT except when using prepared statements.

We created a prepared statement that retrieves the necessary products using the PREPARE command. The statement variables are marked with a question mark (?) in the query. When executing the query with EXECUTE, we provide the values of those parameters as parameters of the EXECUTE command. The prepared statement contains five parameters, so we supply five parameters to EXECUTE (@p1, @p2, @p3, @p4, @p5).

Prepared statements are also useful for performance (because the same statement can be executed multiple times) and security reasons (because the data types for parameters can be checked for data type compliancy). However, in our case, we're using PDO to implement these features, so we're really using prepared statements only so that we can supply parameters to LIMIT.

We'll use the same technique on the other procedures that use LIMIT as well.

catalog_count_products_on_department

This stored procedure counts the number of products that are to be displayed in the page of a given department. Note that all the department's products aren't listed on the department's page, but only those products whose display value is 2 (product on department promotion) or 3 (product on department and catalog promotion).

```
-- Create catalog_count_products_on_department stored procedure
CREATE PROCEDURE catalog_count_products_on_department(IN inDepartmentId INT)
BEGIN
  SELECT DISTINCT COUNT(*) AS products_on_department_count
  FROM            product p
  INNER JOIN      product_category pc
                    ON p.product_id = pc.product_id
  INNER JOIN      category c
                    ON pc.category_id = c.category_id
  WHERE           (p.display = 2 OR p.display = 3)
                  AND c.department_id = inDepartmentId;
END$$
```

The SQL code is almost the same as the one in catalog_get_products_on_department, which we're discussing next.

catalog_get_products_on_department

When the visitor selects a particular department, apart from needing to list its name, description, and list of categories (you wrote the necessary code for these tasks earlier), you also want to display the list of featured products for that department.

catalog_get_products_on_department returns all the products that belong to a specific department and has the display set to 2 (product on department promotion) or 3 (product on department and catalog promotion).

In catalog_get_products_in_category, you needed to make a table join to find out the products that belong to a specific category. Now that you need to do this for departments, the task is a bit more complicated, because you can't directly know what products belong to each department.

You know how to find categories that belong to a specific department (you did this in catalog_get_categories_list), and you know how to get the products that belong to a specific category (you did that in catalog_get_products_in_category). By combining these pieces of information, you can generate the list of products in a department. For this, you need two table joins.

You will also use the DISTINCT clause to filter the results to avoid getting the same record multiple times. This can happen when a product belongs to more than one category, and these categories are in the same department. In this situation, you would get the same product returned for each of the matching categories, unless the results are filtered using DISTINCT.

```
-- Create catalog_get_products_on_department stored procedure
CREATE PROCEDURE catalog_get_products_on_department(
  IN inDepartmentId INT, IN inShortProductDescriptionLength INT,
  IN inProductsPerPage INT, IN inStartItem INT)
BEGIN
  PREPARE statement FROM
    "SELECT DISTINCT p.product_id, p.name,
                     IF(LENGTH(p.description) <= ?,
                        p.description,
                        CONCAT(LEFT(p.description, ?),
                               '...')) AS description,
                     p.price, p.discounted_price, p.thumbnail
     FROM            product p
     INNER JOIN      product_category pc
                       ON p.product_id = pc.product_id
     INNER JOIN      category c
                       ON pc.category_id = c.category_id
     WHERE           (p.display = 2 OR p.display = 3)
                     AND c.department_id = ?
     ORDER BY        p.display DESC
     LIMIT           ?, ?";

  SET @p1 = inShortProductDescriptionLength;
  SET @p2 = inShortProductDescriptionLength;
  SET @p3 = inDepartmentId;
  SET @p4 = inStartItem;
  SET @p5 = inProductsPerPage;

  EXECUTE statement USING @p1, @p2, @p3, @p4, @p5;
END$$
```

Tip If the way table joins work looks too complicated, try following them on the diagram shown earlier in Figure 5-6. The more you work with DBs and queries, the easier it is to do; even experts sometimes struggle with creating the perfect query or get bogged down keeping track of the joins and so forth, so take heart and keep at it. It really is a case of practice makes perfect!

catalog_count_products_on_catalog

The `catalog_count_products_on_catalog` stored procedure returns the count of the number of products to be displayed on the catalog's front page. These are products whose `display` fields have the value of `1` (product is promoted on the first page) or `3` (product is promoted on the first page and on the department pages).

```
-- Create catalog_count_products_on_catalog stored procedure
CREATE PROCEDURE catalog_count_products_on_catalog()
BEGIN
  SELECT COUNT(*) AS products_on_catalog_count
  FROM   product
  WHERE  display = 1 OR display = 3;
END$$
```

catalog_get_products_on_catalog

The `catalog_get_products_on_catalog` stored procedure returns the actual products to be displayed on the catalog's front page. These are products whose `display` fields have the value of `1` (product is promoted on the first page) or `3` (product is promoted on the first page and on the department pages). The product description is trimmed at a specified number of characters. The pagination is implemented the same way as in the previous two stored procedures that return lists of products.

```
-- Create catalog_get_products_on_catalog stored procedure
CREATE PROCEDURE catalog_get_products_on_catalog(
  IN inShortProductDescriptionLength INT,
  IN inProductsPerPage INT, IN inStartItem INT)
BEGIN
  PREPARE statement FROM
    "SELECT   product_id, name,
              IF(LENGTH(description) <= ?,
                 description,
                 CONCAT(LEFT(description, ?),
                        '...')) AS description,
              price, discounted_price, thumbnail
     FROM     product
     WHERE    display = 1 OR display = 3
     ORDER BY display DESC
     LIMIT    ?, ?";

  SET @p1 = inShortProductDescriptionLength;
  SET @p2 = inShortProductDescriptionLength;
  SET @p3 = inStartItem;
  SET @p4 = inProductsPerPage;

  EXECUTE statement USING @p1, @p2, @p3, @p4;
END$$
```

catalog_get_product_details

The `catalog_get_product_details` stored procedure returns detailed information about a product and is called to get the data that will be displayed on the product's details page.

```
-- Create catalog_get_product_details stored procedure
CREATE PROCEDURE catalog_get_product_details(IN inProductId INT)
BEGIN
  SELECT product_id, name, description,
         price, discounted_price, image, image_2
  FROM   product
  WHERE  product_id = inProductId;
END$$
```

catalog_get_product_locations

The `catalog_get_product_locations` stored procedure returns the categories and departments a product is part of. This will help us display the "Find similar products in our catalog" feature in the product details pages.

This is also the first time we're using subqueries in this book. In the following code, we have highlighted the subqueries to make them easier to read. As you can see, a subquery is just like a simple query, except it's written inside another query.

An interesting detail about subqueries is that most of the time they can be used instead of table joins to achieve the same results. Choosing one solution over the other is, in many cases, a matter of preference. In mission-critical solutions, depending on the circumstances, you may choose one over the other for performance reasons. However, this is an advanced subject that we'll let you learn from specialized database books or tutorials.

```
-- Create catalog_get_product_locations stored procedure
CREATE PROCEDURE catalog_get_product_locations(IN inProductId INT)
BEGIN
  SELECT c.category_id, c.name AS category_name, c.department_id,
         (SELECT name
          FROM   department
          WHERE  department_id = c.department_id) AS department_name
          -- Subquery returns the name of the department of the category
  FROM   category c
  WHERE  c.category_id IN
           (SELECT category_id
            FROM   product_category
            WHERE  product_id = inProductId);
            -- Subquery returns the category IDs a product belongs to
END$$
```

Well, that's about it. Right now, your data store is ready to hold and process the product catalog information. To make sure you haven't missed creating any of the stored procedures, you can execute the following command, which shows the stored procedures you currently have in your database:

```
SHOW PROCEDURE STATUS
```

By writing the stored procedures, you've already implemented a significant part of your product catalog! It's time to move to the next step: implementing the business tier of the product catalog.

Completing the Business Tier Code

In the business tier, you'll add some new methods that will call the previously created stored procedures in the data tier. Recall that you started working on the `Catalog` class (located in the `business/catalog.php` file) in Chapter 4. The new methods that you'll add here follow:

- `GetDepartmentDetails()`
- `GetCategoriesInDepartment()`
- `GetCategoryDetails()`
- `HowManyPages()`
- `GetProductsInCategory()`
- `GetProductsOnDepartment()`
- `GetProductsOnCatalog()`
- `GetProductDetails()`
- `GetProductLocations()`

Defining Product List Constants and Activating Session

Before writing the business tier methods, let's first update the `include/config.php` file by adding the `SHORT_PRODUCT_DESCRIPTION_LENGTH` and `PRODUCTS_PER_PAGE` constants. These allow you to easily define the product display of your site by specifying the length of product descriptions and how many products to be displayed per page. These are the values the business tier methods will supply when calling stored procedures that return pages of products, such as `catalog_get_products_on_catalog`.

```
...
// 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/');

// Configure product lists display options
define('SHORT_PRODUCT_DESCRIPTION_LENGTH', 150);
define('PRODUCTS_PER_PAGE', 4);
?>
```

Then, modify index.php by adding these lines to it:

```
<?php
// Activate session
session_start();

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

// Set the error handler
ErrorHandler::SetHandler();
...
```

The SHORT_PRODUCT_DESCRIPTION_LENGTH constant specifies how many characters from the product's description should appear when displaying product lists. The complete description gets displayed in the product's details page, which you'll implement at the end of this chapter.

PRODUCTS_PER_PAGE specifies the maximum number of products that can be displayed in any catalog page. If the visitor's selection contains more than PRODUCTS_PER_PAGE products, the list of products is split into subpages, accessible through the navigation controls.

We also enabled the PHP session, which will help us improve performance when navigating through pages of products.

Note Session handling is a great PHP feature that allows you to keep track of variables specific to a certain visitor accessing the web site. While the visitor browses the catalog, the session variables are persisted by the web server and associated to a unique visitor identifier (which is stored by default in the visitor's browser as a *cookie*). The visitor's session object stores (name, value) pairs that are saved at server-side and are accessible for the visitor's entire session. In this chapter, we'll use the session feature for improving performance. When implementing the paging functionality, before requesting the list of products, you first ask the database for the total number of products that are going to be returned, so you can show the visitor how many pages of products are available. This number will be saved in the visitor's session, so if the visitor browses the pages of a list of products, the database wouldn't be queried multiple times for the total number of products on subsequent calls; this number will be directly read from the session (this functionality is implemented in the HowManyPages() method that you'll implement later). In this chapter, you'll also use the session to implement the Continue Shopping buttons in product details pages.

Let's work through each business tier method. All these methods need to be added to the Catalog class, located in the business/catalog.php file that you started writing in Chapter 4.

GetDepartmentDetails

GetDepartmentDetails() is called from the presentation tier when a department is clicked to display its name and description. The presentation tier passes the ID of the selected department, and you need to send back the name and the description of the selected department.

The needed data is obtained by calling the `catalog_get_department_details` stored procedure that you've written earlier. Just as planned, the business tier acts, in this case, as a buffer between the presentation tier and the data tier.

```
// Retrieves complete details for the specified department
public static function GetDepartmentDetails($departmentId)
{
  // Build SQL query
  $sql = 'CALL catalog_get_department_details(:department_id)';

  // Build the parameters array
  $params = array (':department_id' => $departmentId);

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

GetCategoriesInDepartment

The `GetCategoriesInDepartment()` method is called to retrieve the list of categories that belong to a department. Add this method to the `Catalog` class:

```
// Retrieves list of categories that belong to a department
public static function GetCategoriesInDepartment($departmentId)
{
  // Build SQL query
  $sql = 'CALL catalog_get_categories_list(:department_id)';

  // Build the parameters array
  $params = array (':department_id' => $departmentId);

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

GetCategoryDetails

`GetCategoryDetails()` is called from the presentation tier when a category is clicked to display its name and description. The presentation tier passes the ID of the selected category, and you need to send back the name and the description of the selected category.

```
// Retrieves complete details for the specified category
public static function GetCategoryDetails($categoryId)
{
  // Build SQL query
  $sql = 'CALL catalog_get_category_details(:category_id)';

  // Build the parameters array
  $params = array (':category_id' => $categoryId);
```

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

HowManyPages

As you know, our product catalog will display a fixed number of products in every page. When a catalog page contains more than an established number of products, we display navigation controls that allow the visitor to browse back and forth through the subpages of products. You can see the navigation controls in Figure 4-2 in Chapter 4.

When displaying the navigation controls, you need to calculate the number of subpages of products you have for a given catalog page; for this, we're creating the `HowManyPages()` helper method.

This method receives as an argument a `SELECT` query that counts the total number of products of the catalog page (`$countSql`) and returns the number of subpages. This will be done by simply dividing the total number of products by the number of products to be displayed in a subpage of products; this latter number is configurable through the `PRODUCTS_PER_PAGE` constant in `include/config.php`.

To improve the performance when a visitor browses back and forth through the subpages, after we calculate the number of subpages for the first time, we're saving it to the visitor's session. This way, the SQL query received as parameter won't need to be executed more than once on a single visit to a catalog page.

This method is called from the other data tier methods (`GetProductsInCategory()`, `GetProductsOnDepartment()`, `GetProductsOnCatalog()`), which we'll cover next.

Add `HowManyPages()` to the `Catalog` class:

```
  /* Calculates how many pages of products could be filled by the
     number of products returned by the $countSql query */
  private static function HowManyPages($countSql, $countSqlParams)
  {
    // Create a hash for the sql query
    $queryHashCode = md5($countSql . var_export($countSqlParams, true));

    // Verify if we have the query results in cache
    if (isset ($_SESSION['last_count_hash']) &&
        isset ($_SESSION['how_many_pages']) &&
        $_SESSION['last_count_hash'] === $queryHashCode)
    {
      // Retrieve the the cached value
      $how_many_pages = $_SESSION['how_many_pages'];
    }
    else
    {
      // Execute the query
      $items_count = DatabaseHandler::GetOne($countSql, $countSqlParams);
      // Calculate the number of pages
      $how_many_pages = ceil($items_count / PRODUCTS_PER_PAGE);

      // Save the query and its count result in the session
      $_SESSION['last_count_hash'] = $queryHashCode;
      $_SESSION['how_many_pages'] = $how_many_pages;
    }

    // Return the number of pages
    return $how_many_pages;
  }
```

Let's analyze the function to see how it does its job.

The method is private, because you won't access it from within other classes—it's a helper class for other methods of `Catalog`.

The method verifies whether the previous call to it was for the same `SELECT` query. If it was, the result cached from the previous call is returned. This small trick improves performance when the visitor is browsing subpages of the same list of products because the actual counting in the database is performed only once.

```
    // Create a hash for the sql query
    $queryHashCode = md5($countSql . var_export($countSqlParams, true));

    // Verify if we have the query results in cache
    if (isset ($_SESSION['last_count_hash']) &&
        isset ($_SESSION['how_many_pages']) &&
        $_SESSION['last_count_hash'] === $queryHashCode)
    {
      // Retrieve the cached value
      $how_many_pages = $_SESSION['how_many_pages'];
    }
```

The number of pages associated with the received query and parameters is saved in the current visitor's session in a variable named `how_many_pages`. If the conditions aren't met, which means the results of the query aren't cached, we calculate them and save them to the session:

```
    else
    {
      // Execute the query
      $items_count = DatabaseHandler::GetOne($countSql, $countSqlParams);

      // Calculate the number of pages
      $how_many_pages = ceil($items_count / PRODUCTS_PER_PAGE);

      // Save the query and its count result in the session
      $_SESSION['last_count_hash'] = $queryHashCode;
      $_SESSION['how_many_pages'] = $how_many_pages;
    }
```

In the end, no matter if the number of pages was fetched from the session or calculated by the database, it is returned to the calling function:

```
    // Return the number of pages
    return $how_many_pages;
```

GetProductsInCategory

`GetProductsInCategory()` returns the list of products that belong to a particular category. Add the following method to the `Catalog` class in `business/catalog.php`:

```
  // Retrieves list of products that belong to a category
  public static function GetProductsInCategory(
                         $categoryId, $pageNo, &$rHowManyPages)
  {
    // Query that returns the number of products in the category
    $sql = 'CALL catalog_count_products_in_category(:category_id)';
    // Build the parameters array
    $params = array (':category_id' => $categoryId);

    // Calculate the number of pages required to display the products
    $rHowManyPages = Catalog::HowManyPages($sql, $params);
    // Calculate the start item
    $start_item = ($pageNo - 1) * PRODUCTS_PER_PAGE;

    // Retrieve the list of products
    $sql = 'CALL catalog_get_products_in_category(
               :category_id, :short_product_description_length,
               :products_per_page, :start_item)';

    // Build the parameters array
    $params = array (
      ':category_id' => $categoryId,
      ':short_product_description_length' =>
        SHORT_PRODUCT_DESCRIPTION_LENGTH,
      ':products_per_page' => PRODUCTS_PER_PAGE,
      ':start_item' => $start_item);

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

This function has two purposes:

- Calculate the number of subpages of products and return this number through the `&$rHowManyPages` parameter. To calculate this number, the `HowManyPages()` method you added earlier is used. The SQL query that is used to retrieve the total number of products calls the `catalog_count_products_in_category` database stored procedure that you added earlier to your databases.
- Return the list of products in the mentioned category.

Note The ampersand (&) before a function parameter means it is passed by reference. When a variable is passed by reference, an alias of the variable is passed instead of creating a new copy of the value. This way, when a variable is passed by reference and the called function changes its value, its new value will be reflected in the caller function, too. Passing by reference is an alternative method to receiving a return value from a called function and is particularly useful when you need to get multiple return values from the called function. `CreateSubpageQuery()` returns the text of a `SELECT` query through its return value and the total number of subpages through the `$rHowManyPages` parameter that is passed by reference.

GetProductsOnDepartment

The `GetProductsOnDepartment()` method returns the list of products featured for a particular department. The department's featured products must be displayed when the customer visits the home page of a department. Put it inside the `Catalog` class:

```
// Retrieves the list of products for the department page
public static function GetProductsOnDepartment(
                         $departmentId, $pageNo, &$rHowManyPages)
{
  // Query that returns the number of products in the department page
  $sql = 'CALL catalog_count_products_on_department(:department_id)';
  // Build the parameters array
  $params = array (':department_id' => $departmentId);

  // Calculate the number of pages required to display the products
  $rHowManyPages = Catalog::HowManyPages($sql, $params);
  // Calculate the start item
  $start_item = ($pageNo - 1) * PRODUCTS_PER_PAGE;

  // Retrieve the list of products
  $sql = 'CALL catalog_get_products_on_department(
               :department_id, :short_product_description_length,
               :products_per_page, :start_item)';

  // Build the parameters array
  $params = array (
    ':department_id' => $departmentId,
    ':short_product_description_length' =>
      SHORT_PRODUCT_DESCRIPTION_LENGTH,
    ':products_per_page' => PRODUCTS_PER_PAGE,
    ':start_item' => $start_item);

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

GetProductsOnCatalog

The `GetProductsOnCatalog()` method returns the list of products featured on the catalog's front page. It goes inside the `Catalog` class:

```
// Retrieves the list of products on catalog page
public static function GetProductsOnCatalog($pageNo, &$rHowManyPages)
{
  // Query that returns the number of products for the front catalog page
  $sql = 'CALL catalog_count_products_on_catalog()';

  // Calculate the number of pages required to display the products
  $rHowManyPages = Catalog::HowManyPages($sql, null);
  // Calculate the start item
  $start_item = ($pageNo - 1) * PRODUCTS_PER_PAGE;

  // Retrieve the list of products
  $sql = 'CALL catalog_get_products_on_catalog(
                 :short_product_description_length,
                 :products_per_page, :start_item)';

  // Build the parameters array
  $params = array (
    ':short_product_description_length' =>
      SHORT_PRODUCT_DESCRIPTION_LENGTH,
    ':products_per_page' => PRODUCTS_PER_PAGE,
    ':start_item' => $start_item);

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

GetProductDetails

Add the `GetProductDetails()` method to the `Catalog` class:

```
// Retrieves complete product details
public static function GetProductDetails($productId)
{
  // Build SQL query
  $sql = 'CALL catalog_get_product_details(:product_id)';

  // Build the parameters array
  $params = array (':product_id' => $productId);

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

GetProductLocations

Finally, add the `GetProductLocations()` method, which calls the `catalog_get_product_locations` stored procedure, to extract the categories and departments that a product is part of:

```
// Retrieves product locations
public static function GetProductLocations($productId)
{
  // Build SQL query
  $sql = 'CALL catalog_get_product_locations(:product_id)';

  // Build the parameters array
  $params = array (':product_id' => $productId);

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

Implementing the Presentation Tier

Believe it or not, right now the data and business tiers of the product catalog are complete for this chapter. All that is left to do is use their functionality in the presentation tier. In this final section, you'll create a few Smarty templates and integrate them into the existing project.

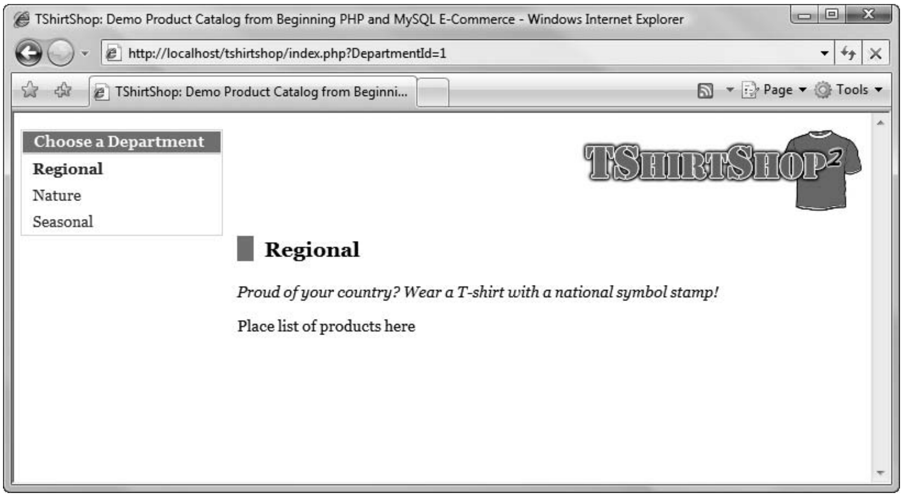
Execute the TShirtShop project (or load `http://localhost/tshirtshop/` in your favorite web browser) to see once again what happens when the visitor clicks a department. After the page loads, click one of the departments. The main page (`index.php`) is reloaded, but this time with a query string at the end of the URL:

```
http://localhost/tshirtshop/index.php?DepartmentId=1
```

Using this parameter, `DepartmentId`, you can obtain any information about the selected department, such as its name, description, list of products, and so on. In the following sections, you'll create the controls that display the list of categories associated with the selected department and the products for the selected department, category, or main web page.

Displaying Department and Category Details

The componentized template responsible for showing the contents of a particular department is named `department`, and you'll build it in the exercise that follows. You'll first create the componentized template and then modify the `store_front` componentized template to load it when `DepartmentId` is present in the query string. After this exercise, when clicking a department in the list, you should see a page like the one in Figure 5-8.


TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/index.php?DepartmentId=1
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
Choose a Department
Regional
Nature
Seasonal
TSHIRTSHOP²

Regional

Proud of your country? Wear a T-shirt with a national symbol stamp!

Place list of products here

Figure 5-8. *Selecting the Regional department*

Exercise: Displaying Department Details

1. Add the following two styles to the `tshirtshop.css` file from the `styles` folder. You'll need them for displaying the department's title and description:

```
.title {
  border-left: 15px solid #0590C7;
  padding-left: 10px;
}

.description {
  font-style: italic;
}
```

2. Create a new template file named `blank.tpl` in the `presentation/templates` folder with the following contents:

```
{* Smarty blank page *}
```

Yes, this is a blank Smarty template file, which contains just a comment. You'll use it a bit later. *Make sure you add that comment to the file*; if you leave it empty, you'll get an error when trying to use the template—and that's never fun.

3. Create a new template file named `department.tpl` in the `presentation/templates` folder, and add the following code to it:

```
{* department.tpl *}
{load_presentation_object filename="department" assign="obj"}
<h1 class="title">{$obj->mName}</h1>
<p class="description">{$obj->mDescription}</p>
Place list of products here
```

The two presentation object members, `$obj->mName` and `$obj->mDescription`, contain the name and description of the selected department. The `Department` presentation object is created by the `presentation/smarty_plugins/function.load_presentation_object.php` plug-in.

4. Let's now create the `Department` presentation object for `department.tpl`. Create the `presentation/department.php` file, and add the following code to it:

```
<?php
// Deals with retrieving department details
class Department
{
  // Public variables for the smarty template
  public $mName;
  public $mDescription;

  // Private members
  private $_mDepartmentId;
  private $_mCategoryId;

  // Class constructor
  public function __construct()
  {
    // We need to have DepartmentId in the query string
    if (isset ($_GET['DepartmentId']))
      $this->_mDepartmentId = (int)$_GET['DepartmentId'];
    else
      trigger_error('DepartmentId not set');

    /* If CategoryId is in the query string we save it
       (casting it to integer to protect against invalid values) */
    if (isset ($_GET['CategoryId']))
      $this->_mCategoryId = (int)$_GET['CategoryId'];
  }

  public function init()
  {
    // If visiting a department ...
    $department_details =
      Catalog::GetDepartmentDetails($this->_mDepartmentId);

    $this->mName = $department_details['name'];
    $this->mDescription = $department_details['description'];

    // If visiting a category ...
    if (isset ($this->_mCategoryId))
    {
      $category_details =
        Catalog::GetCategoryDetails($this->_mCategoryId);
      $this->mName = $this->mName . ' &raquo; ' .
                     $category_details['name'];
      $this->mDescription = $category_details['description'];
    }
  }
}
?>
```

5. Now, let's modify `presentation/templates/store_front.tpl` and `presentation/store_front.php` to load the newly created componentized template when `DepartmentId` appears in the query string. If the visitor is browsing a department, you set the `$mContentsCell` member in the `StoreFront` presentation object to the componentized template you have just created so that the products you've chosen to display on the store front appear.

   Modify `presentation/store_front.php` as shown:

```
<?php
class StoreFront
{
  public $mSiteUrl;
  // Define the template file for the page contents
  public $mContentsCell = 'blank.tpl';

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

  // Initialize presentation object
  public function init()
  {
    // Load department details if visiting a department
    if (isset ($_GET['DepartmentId']))
    {
      $this->mContentsCell = 'department.tpl';
    }
  }
}
?>
```

6. Open `presentation/templates/store_front.tpl`, and replace the text `Place contents here` with

```
{include file=$obj->mContentsCell}
```

7. Load your web site in a browser, and select one of the departments to ensure everything works as expected, as shown earlier in Figure 5-8.

Caution Certain versions of PHP 5 have a bug that generates an error when loading the site at this stage, which reads "General error: 2014 Cannot execute queries while other unbuffered queries are active." The bug is documented at `http://bugs.php.net/bug.php?id=39858`. Consult the book's errata page for solutions to this problem.

How It Works: The Department Componentized Template

Congratulations! If your little list of departments functions as described, you've just made it past the toughest part of this book. Make sure you understand very well what happens in the code before moving on.

After implementing the data and business tiers, adding the visual part was a fairly easy task. After adding the CSS styles and creating the blank template file, you created the Smarty template file `department.tpl`, which contains the HTML layout for displaying a department's data. This template file is loaded in the contents cell, just below the header, in `store_front.tpl`:

```
<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">
      {include file=$obj->mContentsCell}
    </div>
  </div>
</div>
```

The `$mContentsCell` field of the presentation object is populated in `store_front.php`, depending on the query string parameters. At the moment, if the `DepartmentId` parameter is found in the query string, the page contents cell is populated with the `department.tpl` template file you just wrote. Otherwise (such as when you're on the first page), the `blank.tpl` template file is used, since we haven't yet implemented the contents cell template (you'll change this when creating a template to populate the contents cell for the first page).

This is the code in `store_front.php` that assigns a value to `$mContentsCell` when a department is selected:

```
// Initialize presentation object
public function init()
{
  // Load department details if visiting a department
  if (isset ($_GET['DepartmentId']))
  {
    $this->mContentsCell = 'department.tpl';
  }
}
```

The first interesting aspect to know about `department.tpl` is the way it loads the `Department` presentation object with the help of `load_presentation_object` Smarty plug-in function.

```
{* department.tpl *}
{load_presentation_object filename="department" assign="obj"}
```

This allows you to access the instance of the Department class (which we'll discuss next) and its public members ($mName and $mDescription) from the template file (department.tpl), like this:

```
<h1 class="title">{$obj->mName}</h1>
<p class="description">{$obj->mDescription}</p>
Place list of products here
```

The next step now is to understand how the presentation object, department.php, does its work to obtain the department's name and description. The file contains the Department class. The two public members of Department are the ones you access from the Smarty template (the department's name and description). The final role of this class is to populate these members, which are required to build the output for the visitor:

```
// Deals with retrieving department details
class Department
{
  // Public variables for the smarty template
  public $mName;
  public $mDescription;
```

There are also two private members that are used for internal purposes. $_mDepartmentId and $_mCategoryId will store the values of the DepartmentId and CategoryId query string parameters:

```
  // Private members
  private $_mDepartmentId;
  private $_mCategoryId;
```

Next comes the constructor. In any object-oriented language, the constructor of the class is executed when the class is instantiated, and the constructor is used to perform various initialization procedures. In our case, the constructor of Department reads the DepartmentId and CategoryId query string parameters into the $_mDepartmentId and $_mCategoryId private class members. You need these because if CategoryId actually exists in the query string, then you also need to display the name of the category and the category's description instead of the department's description.

```
  // Class constructor
  public function __construct()
  {
    // We need to have DepartmentId in the query string
    if (isset ($_GET['DepartmentId']))
      $this->_mDepartmentId = (int)$_GET['DepartmentId'];
    else
      trigger_error('DepartmentId not set');

    /* If CategoryId is in the query string we save it
       (casting it to integer to protect against invalid values) */
    if (isset ($_GET['CategoryId']))
      $this->_mCategoryId = (int)$_GET['CategoryId'];
  }
```

The real functionality of the class is hidden inside the init() method. In our solution, the init() method is always executed immediately after the constructor, because it's called immediately after the object is created, in the load_presentation_object Smarty plug-in function (as you know from Chapter 4, this plug-in function is used by all Smarty templates to load their presentation objects).

The init() method populates the $mName and $mDescription public members with information from the business tier. The GetDepartmentDetails() method of the business tier Catalog class is used to retrieve the details of the department; if necessary, the GetCategoryDetails() method is also called to retrieve the details of the category (the details of the department need to be retrieved even when visiting a category, because in that case, we display both the department name and the category name).

```
public function init()
{
  // If visiting a department ...
  $department_details =
    Catalog::GetDepartmentDetails($this->_mDepartmentId);

  $this->mName = $department_details['name'];
  $this->mDescription = $department_details['description'];

  // If visiting a category ...
  if (isset ($this->_mCategoryId))
  {
    $category_details =
      Catalog::GetCategoryDetails($this->_mCategoryId);

    $this->mName = $this->mName . ' &raquo; ' .
                   $category_details['name'];
    $this->mDescription = $category_details['description'];
  }
}
```

Displaying the List of Categories

When a visitor selects a department, the categories that belong to that department must appear. For this, you'll implement a new Smarty template named categories_list. categories_list is very similar to the department_list componentized template. It consists of a template section used for looping over the array of categories data (category name and category ID). This template section will contain links to index.php, but this time, their query string will also contain a CategoryId showing that a category has been clicked, like this:

```
http://localhost/tshirtshop/index.php?DepartmentId=1&CategoryId=2
```

The steps in the following exercise are very much like the ones for the departments_list componentized template (created at the end of Chapter 4), so we'll move a bit more quickly this time.

Exercise: Creating the categories_list Componentized Template

1. Create the Smarty template for the `categories_list` componentized template. Write the following lines in `presentation/templates/categories_list.tpl`:

```
{* categories_list.tpl *}
{load_presentation_object filename="categories_list" assign="obj"}
{* Start categories list *}
<div class="box">
  <p class="box-title">Choose a Category</p>
  <ul>
  {section name=i loop=$obj->mCategories}
    {assign var=selected value=""}
    {if ($obj->mSelectedCategory == $obj->mCategories[i].category_id)}
      {assign var=selected value="class=\"selected\""}
    {/if}
    <li>
      <a {$selected} href="{$obj->mCategories[i].link_to_category}">
        {$obj->mCategories[i].name}
      </a>
    </li>
  {/section}
  </ul>
</div>
{* End categories list *}
```

2. Create the `presentation/categories_list.php` file, and add the following code to it:

```
<?php
// Manages the categories list
class CategoriesList
{
  // Public variables for the smarty template
  public $mSelectedCategory   = 0;
  public $mSelectedDepartment = 0;
  public $mCategories;

  // Constructor reads query string parameter
  public function __construct()
  {
    if (isset ($_GET['DepartmentId']))
      $this->mSelectedDepartment = (int)$_GET['DepartmentId'];
    else
      trigger_error('DepartmentId not set');

    if (isset ($_GET['CategoryId']))
      $this->mSelectedCategory = (int)$_GET['CategoryId'];
  }
  public function init()
  {
    $this->mCategories =
      Catalog::GetCategoriesInDepartment($this->mSelectedDepartment);

    // Building links for the category pages
    for ($i = 0; $i < count($this->mCategories); $i++)
      $this->mCategories[$i]['link_to_category'] =
        Link::ToCategory($this->mSelectedDepartment,
                         $this->mCategories[$i]['category_id']);
  }
}
?>
```

3. Open the `presentation/link.php` file, and add the `ToCategory()` method shown below to `Link` class. This method creates categories links.

```
  public static function ToCategory($departmentId, $categoryId)
  {
    $link = 'index.php?DepartmentId=' . $departmentId .
            '&CategoryId=' . $categoryId;

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

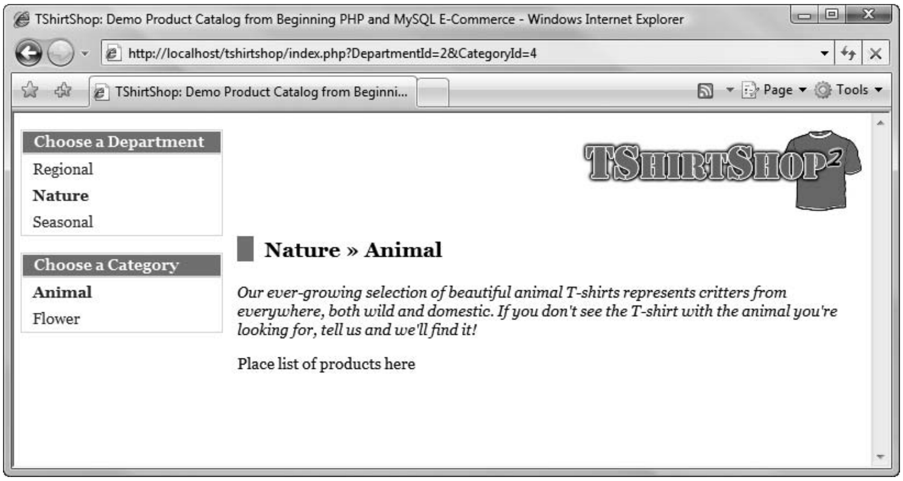
4. Modify `presentation/store_front.php` like this:

```
<?php
class StoreFront
{
  public $mSiteUrl;
  // Define the template file for the page contents
  public $mContentsCell = 'blank.tpl';
  // Define the template file for the categories cell
  public $mCategoriesCell = 'blank.tpl';

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

  // Initialize presentation object
  public function init()
  {
    // Load department details if visiting a department
    if (isset ($_GET['DepartmentId']))
    {
      $this->mContentsCell = 'department.tpl';
      $this->mCategoriesCell = 'categories_list.tpl';
    }
  }
}
?>
```

5. Now include the `categories_list` componentized template in `presentation/templates/store_front.tpl` just below the list of departments. Note that we include `$obj->mCategoriesCell` and not `categories_list.tpl`, because the list of categories needs to show up only when a department is selected. When no department is selected, `$obj->mCategoriesCell` is set to `blank.tpl` in `store_front.php`.

```
{include file="departments_list.tpl"}
{include file=$obj->mCategoriesCell}
```

6. Load TShirtShop in a web browser. When the page loads, click one of the departments. You'll see the categories list appear in the chosen place. Selecting a category displays the category description, as shown in Figure 5-9.


TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/index.php?DepartmentId=2&CategoryId=4
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
TSHIRTSHOP2

Choose a Department
Regional
Nature
Seasonal

Choose a Category
Animal
Flower

Nature » Animal

Our ever-growing selection of beautiful animal T-shirts represents critters from everywhere, both wild and domestic. If you don't see the T-shirt with the animal you're looking for, tell us and we'll find it!

Place list of products here

Figure 5-9. *Selecting the Animal category*

How It Works: The categories_list Componentized Template

The `categories_list` componentized template works similarly to the `departments_list`. The `CategoriesList` class (located in the `presentation/categories_list.php` presentation object file) has three public members that can be accessed from the template file (`categories_list.tpl`):

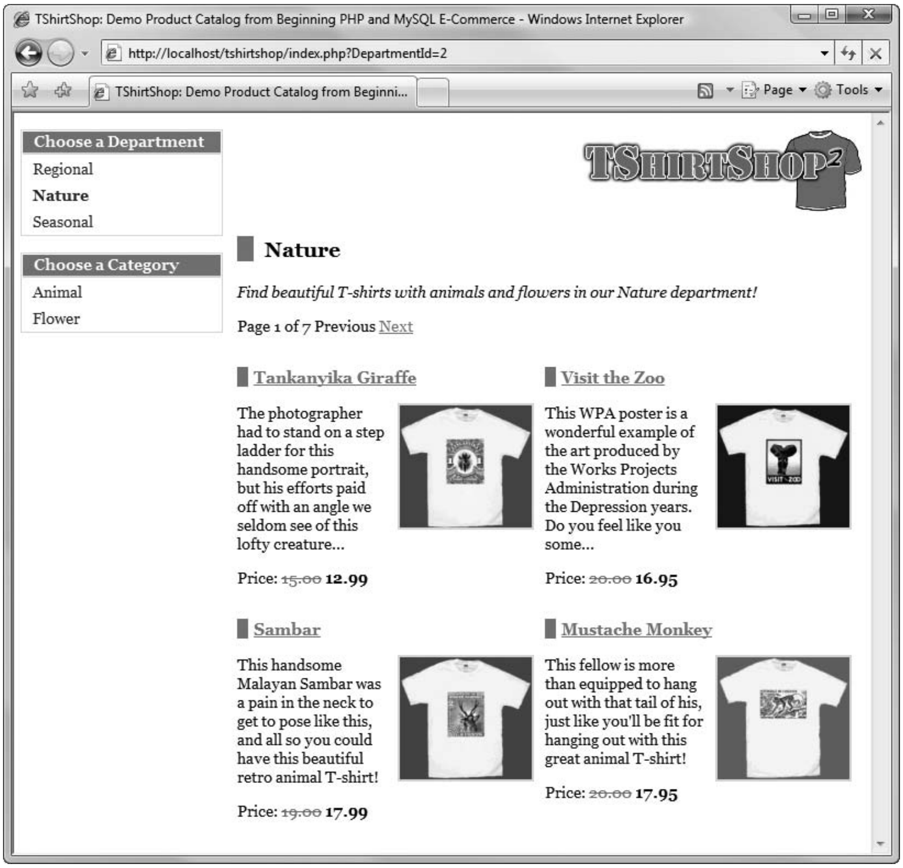
```
// Public variables for the smarty template
public $mSelectedCategory   = 0;
public $mSelectedDepartment = 0;
public $mCategories;
```

`$mSelectedCategory` retains the category that is selected, which must be displayed with a different style than the other categories in the list. The same is true with `$mSelectedDepartment`. `$mCategories` is the list of categories you populate the categories list with. This list is obtained with a call to the business tier.

The links in the categories list are created using the `Link::ToCategory()` method to ensure the consistency of the links across the site and to ensure they're also properly escaped (& is transformed to `&amp;`, and so on).

Displaying Product Lists

Whether on the main web page or browsing a category, some products should appear instead of the "Place list of products here" text. Here, you create the `products_list` componentized template, which is capable of displaying a list containing detailed information about the products. When a large number of products need to be presented, navigation links will appear (see Figure 5-10).


TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/index.php?DepartmentId=2
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
TSHIRTSHOP2

Choose a Department
Regional
Nature
Seasonal

Choose a Category
Animal
Flower

Nature

Find beautiful T-shirts with animals and flowers in our Nature department!

Page 1 of 7 Previous Next

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**

Visit the Zoo

This WPA poster is a wonderful example of the art produced by the Works Projects Administration during the Depression years. Do you feel like you some...

Price: ~~20.00~~ **16.95**

Sambar

This handsome Malayan Sambar was a pain in the neck to get to pose like this, and all so you could have this beautiful retro animal T-shirt!

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

Mustache Monkey

This fellow is more than equipped to hang out with that tail of his, just like you'll be fit for hanging out with this great animal T-shirt!

Price: ~~20.00~~ **17.95**

Figure 5-10. *The products_list componentized template with paging*

This componentized template will be used in multiple places within the web site. On the main page, it displays the products that have the display field set to 1 or 3. When a visitor selects a particular department, the products_list componentized template displays the products featured for the selected department. Finally, when the visitor clicks a category, the componentized template displays all the products that belong to that category. Due to the way the database is implemented, you can feature a product in the departments it belongs to but not on the main page or vice versa. If a product belongs to more than one department, it will appear on the main page of each of these departments.

The componentized template chooses which products to display after analyzing the query string. If both DepartmentId and CategoryId parameters are present in the query string, this means the products of that category should be listed. If only DepartmentId is present, the visitor is visiting a department, so its featured products should appear. If DepartmentId is not present, the visitor is on the main page, so the catalog featured products should appear.

To integrate the products_list componentized template with the first page, you'll need to create an additional template file (first_page_contents.tpl), which you'll implement later. After creating products_list in the following exercise, you'll be able to browse the products by department and by category. Afterward, you'll see how to add products to the main web page.

Exercise: Creating the products_list Componentized Template

1. Copy the product_images directory from the code archive of the book to your project's tshirtshop folder.
2. Add the following styles to the tshirtshop.css file, from the styles folder:

```
.product-list tbody tr td {
  border: none;
  padding: 0;
  width: 50%;
}

.product-list tbody tr td p img {
  border: 2px solid #c6e1ec;
  float: right;
  margin: 0 10px;
  vertical-align: top;
}

.product-title {
  border-left: 10px solid #0590C7;
  padding-left: 5px;
}

.section {
  display: block;
}

.price {
  font-weight: bold;
}
```

```
.old-price {
  color: #ff0000;
  font-weight: normal;
  text-decoration: line-through;
}
```

3. Create a new Smarty design template named `products_list.tpl` inside the `presentation/templates` folder, and add the following code to it:

```
{* products_list.tpl *}
{load_presentation_object filename="products_list" assign="obj"}
{if $obj->mrTotalPages > 1}
<p>
  Page {$obj->mPage} of {$obj->mrTotalPages}
  {if $obj->mLinkToPreviousPage}
  <a href="{$obj->mLinkToPreviousPage}">Previous</a>
  {else}
  Previous
  {/if}
  {if $obj->mLinkToNextPage}
  <a href="{$obj->mLinkToNextPage}">Next</a>
  {else}
  Next
  {/if}
</p>
{/if}
{if $obj->mProducts}
<table class="product-list" border="0">
  <tbody>
  {section name=k loop=$obj->mProducts}
    {if $smarty.section.k.index % 2 == 0}
    <tr>
    {/if}
      <td valign="top">
        <h3 class="product-title">
          <a href="{$obj->mProducts[k].link_to_product}">
            {$obj->mProducts[k].name}
          </a>
        </h3>
        <p>
          {if $obj->mProducts[k].thumbnail neq ""}
          <a href="{$obj->mProducts[k].link_to_product}">
            <img src="{$obj->mProducts[k].thumbnail}"
             alt="{$obj->mProducts[k].name}" />
          </a>
          {/if}
          {$obj->mProducts[k].description}
        </p>
        <p class="section">
          Price:
          {if $obj->mProducts[k].discounted_price != 0}
            <span class="old-price">{$obj->mProducts[k].price}</span>
            <span class="price">{$obj->mProducts[k].discounted_price}</span>
          {else}
            <span class="price">{$obj->mProducts[k].price}</span>
          {/if}
        </p>
      </td>
    {if $smarty.section.k.index % 2 != 0 && !$smarty.section.k.first ||
        $smarty.section.k.last}
    </tr>
    {/if}
  {/section}
  </tbody>
</table>
{/if}
```

4. Now, you must create the presentation object file for the `products_list.tpl` template. Create a new file named `products_list.php` in the `presentation` folder, and add the following code to it:

```
<?php
class ProductsList
{
  // Public variables to be read from Smarty template
  public $mPage = 1;
  public $mrTotalPages;
  public $mLinkToNextPage;
  public $mLinkToPreviousPage;
  public $mProducts;

  // Private members
  private $_mDepartmentId;
  private $_mCategoryId;

  // Class constructor
  public function __construct()
  {
    // Get DepartmentId from query string casting it to int
    if (isset ($_GET['DepartmentId']))
      $this->_mDepartmentId = (int)$_GET['DepartmentId'];

    // Get CategoryId from query string casting it to int
    if (isset ($_GET['CategoryId']))
      $this->_mCategoryId = (int)$_GET['CategoryId'];

    // Get Page number from query string casting it to int
    if (isset ($_GET['Page']))
      $this->mPage = (int)$_GET['Page'];

    if ($this->mPage < 1)
      trigger_error('Incorrect Page value');
  }

  public function init()
  {
    /* If browsing a category, get the list of products by calling
       the GetProductsInCategory() business tier method */
    if (isset ($this->_mCategoryId))
      $this->mProducts = Catalog::GetProductsInCategory(
        $this->_mCategoryId, $this->mPage, $this->mrTotalPages);
    /* If browsing a department, get the list of products by calling
       the GetProductsOnDepartment() business tier method */
    elseif (isset ($this->_mDepartmentId))
      $this->mProducts = Catalog::GetProductsOnDepartment(
        $this->_mDepartmentId, $this->mPage, $this->mrTotalPages);

    /* If there are subpages of products, display navigation
       controls */
    if ($this->mrTotalPages > 1)
    {
      // Build the Next link
      if ($this->mPage < $this->mrTotalPages)
      {
        if (isset($this->_mCategoryId))
          $this->mLinkToNextPage =
            Link::ToCategory($this->_mDepartmentId, $this->_mCategoryId,
                             $this->mPage + 1);
        elseif (isset($this->_mDepartmentId))
          $this->mLinkToNextPage =
            Link::ToDepartment($this->_mDepartmentId, $this->mPage + 1);
      }

      // Build the Previous link
      if ($this->mPage > 1)
      {
        if (isset($this->_mCategoryId))
          $this->mLinkToPreviousPage =
            Link::ToCategory($this->_mDepartmentId, $this->_mCategoryId,
                             $this->mPage - 1);
        elseif (isset($this->_mDepartmentId))
          $this->mLinkToPreviousPage =
            Link::ToDepartment($this->_mDepartmentId, $this->mPage - 1);
      }
    }
```

```
      // Build links for product details pages
      for ($i = 0; $i < count($this->mProducts); $i++)
      {
        $this->mProducts[$i]['link_to_product'] =
          Link::ToProduct($this->mProducts[$i]['product_id']);

        if ($this->mProducts[$i]['thumbnail'])
          $this->mProducts[$i]['thumbnail'] =
            Link::Build('product_images/' . $this->mProducts[$i]['thumbnail']);
      }
    }
  }
  ?>
```

5. Modify presentation/link.php as highlighted here to add a new method to the Link class called ToProduct(), which creates links to product pages. Also, we'll add the parameter $page to the two existing methods, ToDepartment() and ToCategory(), and the code needed for pagination.

```
  public static function ToDepartment($departmentId, $page = 1)
  {
    $link = 'index.php?DepartmentId=' . $departmentId;

    if ($page > 1)
      $link .= '&Page=' . $page;

    return self::Build($link);
  }

  public static function ToCategory($departmentId, $categoryId, $page = 1)
  {
    $link = 'index.php?DepartmentId=' . $departmentId .
            '&CategoryId=' . $categoryId;

    if ($page > 1)
      $link .= '&Page=' . $page;

    return self::Build($link);
  }

  public static function ToProduct($productId)
  {
    return self::Build('index.php?ProductId=' . $productId);
  }
}
?>
```

6. Open `presentation/templates/department.tpl` and replace

   ```
   Place list of products here
   ```

   with

   ```
   {include file="products_list.tpl"}
   ```

7. Load your project in your favorite browser; navigate to one of the departments; and then select a category from a department. Also, find a category with more than four products to test that the paging functionality works, as shown earlier in Figure 5-10.

How It Works: The products_list Componentized Template

Because most functionality regarding the products list has already been implemented in the data and business tiers, this task was fairly simple. The Smarty design template file (`products_list.tpl`) contains the layout to be used when displaying products, and its presentation object file (`presentation/products_list.php`) gets the correct list of products to display.

The constructor in `products_list.php` (the `ProductsList` class) creates a new instance of the business tier object (`Catalog`) and retrieves `DepartmentId`, `CategoryId`, and `Page` from the query string, casting them to `int` as a security measure. These values are used to decide which products to display:

```
// Class constructor
public function __construct()
{
  // Get DepartmentId from query string casting it to int
  if (isset ($_GET['DepartmentId']))
    $this->_mDepartmentId = (int)$_GET['DepartmentId'];

  // Get CategoryId from query string casting it to int
  if (isset ($_GET['CategoryId']))
    $this->_mCategoryId = (int)$_GET['CategoryId'];

  // Get Page number from query string casting it to int
  if (isset ($_GET['Page']))
    $this->mPage = (int)$_GET['Page'];

  if ($this->mPage < 1)
    trigger_error('Incorrect Page value');
}
```

The `init()` method, which continues the constructor's job, starts by retrieving the requested list of products. It decides what method of the business tier to call by analyzing the `$_mCategoryId` and `$_mDepartmentId` members (which, thanks to the constructor, represent the values of the `CategoryId` and `DepartmentId` query string parameters).

If `CategoryId` is present in the query string, it means the visitor is browsing a category, so `GetProductsInCategory()` is called to retrieve the products in that category. If only `DepartmentId` is present, `GetProductsOnDepartment()` is called to retrieve the department's featured products.

```
  public function init()
  {
    /* If browsing a category, get the list of products by calling
       the GetProductsInCategory() business tier method */
    if (isset ($this->_mCategoryId))
      $this->mProducts = Catalog::GetProductsInCategory(
        $this->_mCategoryId, $this->mPage, $this->mrTotalPages);
    /* If browsing a department, get the list of products by calling
       the GetProductsOnDepartment() business tier method */
    elseif (isset ($this->_mDepartmentId))
      $this->mProducts = Catalog::GetProductsOnDepartment(
        $this->_mDepartmentId, $this->mPage, $this->mrTotalPages);
```

The next part of the function takes care of paging. If the business tier call tells you there is more than one page of products (so there are more products than what you specified in the `PRODUCTS_PER_PAGE` constant), you need to show the visitor the current subpage of products being visited, the total number of subpages, and the Previous and Next page links. The comments in code should make the functionality fairly clear, so we won't reiterate the code here.

In the final part of the function, you added the `link_to_product` and `thumbnail` data to each `$mProducts` record, which contain, respectively, the link to the product's page and its thumbnail file name. These values are used in the template file to display the product images and create links to the product pages on the products' names and pictures. The links are created using the `Link::ToProduct()` and `Link::Build()` methods:

```
    // Build links for product details pages
    for ($i = 0; $i < count($this->mProducts); $i++)
    {
      $this->mProducts[$i]['link_to_product'] =
        Link::ToProduct($this->mProducts[$i]['product_id']);

      if ($this->mProducts[$i]['thumbnail'])
        $this->mProducts[$i]['thumbnail'] =
          Link::Build('product_images/' . $this->mProducts[$i]['thumbnail']);
    }
```

We've also modified the methods of the `Link` class to add the `Page` parameter to the query string, if the page number is greater than `1`. This is now necessary since adding product lists to our catalog implies supporting the paging functionality.

```
    if ($page > 1)
      $link .= '?Page=' . $page;
```

Displaying Front Page Contents

Apart from general information about the web site, you also want to show some promotional products on the first page of TShirtShop.

If the visitor browses a department or a category, the `department` Smarty template is used to build the output. For the main web page, we'll create the `first_page_contents` componentized template that will build the output.

Remember the `StoreFront` class in `presentation/store_front.php`, where you have a member named `$mContentsCell` that you fill with different details depending on what part of the site is being visited? When a department or a category is being visited, the department componentized template is loaded, and it takes care of filling that space. We still haven't done anything with that cell for the first page, when no department or category has been selected.

In the following exercise, you'll write a template file that contains some information about the web site and shows the products that have been set up as promotions on the first page. Remember that the product table contains a field named `display`. Site administrators will set this field to `on_catalog` for products that need to be displayed in the first page.

Exercise: Creating the first_page_contents Componentized Template

1. Start by creating the Smarty design template file. The `presentation/templates/first_page_contents.tpl` file should have these contents:

```
{* first_page_contents.tpl *}
<p class="description">
  We hope you have fun developing TShirtShop, the e-commerce store from
  Beginning PHP and MySQL E-Commerce: From Novice to Professional!
</p>
<p class="description">
  We have the largest collection of t-shirts with postal stamps on Earth!
  Browse our departments and cateogories to find your favorite!
</p>
{include file="products_list.tpl"}
```

2. Modify the `presentation/store_front.php` file, as highlighted in the following code:

```
<?php
class StoreFront
{
  public $mSiteUrl;
  // Define the template file for the page contents
  public $mContentsCell = 'first_page_contents.tpl';
  // Define the template file for the categories cell
  public $mCategoriesCell = 'blank.tpl';
```

This way, when no `DepartmentId` and `CategoryId` are in the query string, `store_front.php` will load the `first_page_contents` componentized template.

3. Modify the `init()` method from the `ProductsList` class located in the `presentation/products_list.php` file, as highlighted:

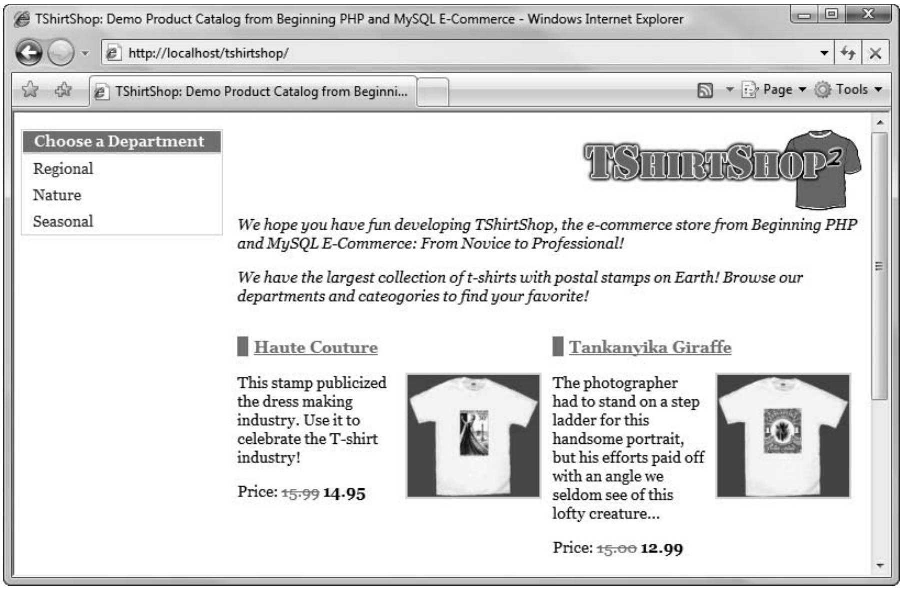
```
  public function init()
  {
    /* If browsing a category, get the list of products by calling
       the GetProductsInCategory() business tier method */
    if (isset ($this->_mCategoryId))
      $this->mProducts = Catalog::GetProductsInCategory(
        $this->_mCategoryId, $this->mPage, $this->mrTotalPages);
```

```
/* If browsing a department, get the list of products by calling
   the GetProductsOnDepartment() business tier method */
elseif (isset ($this->_mDepartmentId))
  $this->mProducts = Catalog::GetProductsOnDepartment(
    $this->_mDepartmentId, $this->mPage, $this->mrTotalPages);
/* If browsing the first page, get the list of products by
   calling the GetProductsOnCatalog() business
   tier method */
else
  $this->mProducts = Catalog::GetProductsOnCatalog(
                        $this->mPage, $this->mrTotalPages);

/* If there are subpages of products, display navigation
   controls */
if ($this->mrTotalPages > 1)
{
  // Build the Next link
  if ($this->mPage < $this->mrTotalPages)
  {
    if (isset($this->_mCategoryId))
      $this->mLinkToNextPage =
        Link::ToCategory($this->_mDepartmentId, $this->_mCategoryId,
                         $this->mPage + 1);
    elseif (isset($this->_mDepartmentId))
      $this->mLinkToNextPage =
        Link::ToDepartment($this->_mDepartmentId, $this->mPage + 1);
    else
      $this->mLinkToNextPage = Link::ToIndex($this->mPage + 1);
  }

  // Build the Previous link
  if ($this->mPage > 1)
  {
    if (isset($this->_mCategoryId))
      $this->mLinkToPreviousPage =
        Link::ToCategory($this->_mDepartmentId, $this->_mCategoryId,
                         $this->mPage - 1);
    elseif (isset($this->_mDepartmentId))
      $this->mLinkToPreviousPage =
        Link::ToDepartment($this->_mDepartmentId, $this->mPage - 1);
    else
      $this->mLinkToPreviousPage = Link::ToIndex($this->mPage - 1);
  }
}
```

4. Open the `link.php` file located in the `presentation` folder, and add the following method to the `Link` class:

```
public static function ToIndex($page = 1)
{
  $link = '';

  if ($page > 1)
    $link .= 'index.php?Page=' . $page;

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

5. Load your project in your favorite browser. The result should look like Figure 5-11.


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
Choose a Department
Regional
Nature
Seasonal
TSHIRTSHOP2

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 5-11. *The front page of TShirtShop*

How It Works: The first_page_contents Componentized Template

The actual list of products is still displayed using the `products_list` Smarty componentized template, which you built earlier in this chapter. However, this time, it isn't loaded from `department.tpl` (like it loads when browsing a department or a category) but from a new template file named `first_page_contents.tpl`.

We added new functionality to the `init()` method in the `ProductsList` class for displaying the products to be featured on the first page of the site. Also, we created the `ToIndex()` method in the `Link` class to support pagination for the first page of the site.

Showing Product Details

The last bit of code you'll implement in this chapter is about displaying product details. When a visitor clicks any product, he or she will be forwarded to the product's details page, which shows the product's complete description and the secondary product image. In later chapters, you'll add more features to this page, such as product recommendations or product reviews.

On this page, we'll also implement the Continue Shopping functionality. This consists of a Continue Shopping link at the bottom of the product details page, which links to the page the visitor was prior to looking at a product's detail page. If the visitor arrives to that page by browsing directly, or from a search engine, the Continue Shopping button will link to the home page of the shop. Let's do this in the following exercise.

Exercise: Creating the product Componentized Template

1. Add the following styles to `styles/tshirtshop.css`:

```
.product-image {
  border: 2px solid #c6e1ec;
}

ol {
  margin: 0px;
  padding: 0px 0px 0px 5px;
}

ol li {
  color: #0590C7;
  list-style-type: none;
  margin: 0px;
  padding: 5px 0px;
}
```

2. Now, get in touch with your artistic side, and use these CSS definitions in the product details page. Create the `product.tpl` file in the `presentation/templates` folder. Feel free to go wild and customize this page as you want.

```
{load_presentation_object filename="product" assign="obj"}
<h1 class="title">{$obj->mProduct.name}</h1>
{if $obj->mProduct.image}
<img class="product-image" src="{$obj->mProduct.image}"
 alt="{$obj->mProduct.name} image" />
{/if}
{if $obj->mProduct.image_2}
<img class="product-image" src="{$obj->mProduct.image_2}"
 alt="{$obj->mProduct.name} image 2" />
{/if}
<p class="description">{$obj->mProduct.description}</p>
```

```
<p class="section">
  Price:
  {if $obj->mProduct.discounted_price != 0}
    <span class="old-price">{$obj->mProduct.price}</span>
    <span class="price">{$obj->mProduct.discounted_price}</span>
  {else}
    <span class="price">{$obj->mProduct.price}</span>
  {/if}
</p>
{if $obj->mLinkToContinueShopping}
<a href="{$obj->mLinkToContinueShopping}">Continue Shopping</a>
{/if}
<h2>Find similar products in our catalog:</h2>
<ol>
{section name=i loop=$obj->mLocations}
  <li class="navigation">
    {strip}
    <a href="{$obj->mLocations[i].link_to_department}">
      {$obj->mLocations[i].department_name}
    </a>
    {/strip}
    &raquo;
    {strip}
    <a href="{$obj->mLocations[i].link_to_category}">
      {$obj->mLocations[i].category_name}
    </a>
    {/strip}
  </li>
{/section}
</ol>
```

3. OK, now create the componentized template for the product details page in which the product with full description and a second image will display. Create a file named `product.php` in the `presentation` folder with the following contents:

```
<?php
// Handles product details
class Product
{
  // Public variables to be used in Smarty template
  public $mProduct;
  public $mProductLocations;
  public $mLinkToContinueShopping;
  public $mLocations;

  // Private stuff
  private $_mProductId;
```

```
// Class constructor
public function __construct()
{
  // Variable initialization
  if (isset ($_GET['ProductId']))
    $this->_mProductId = (int)$_GET['ProductId'];
  else
    trigger_error('ProductId not set');
}

public function init()
{
  // Get product details from business tier
  $this->mProduct = Catalog::GetProductDetails($this->_mProductId);

  if (isset ($_SESSION['link_to_continue_shopping']))
  {
    $continue_shopping =
      Link::QueryStringToArray($_SESSION['link_to_continue_shopping']);

    $page = 1;

    if (isset ($continue_shopping['Page']))
      $page = (int)$continue_shopping['Page'];

    if (isset ($continue_shopping['CategoryId']))
      $this->mLinkToContinueShopping =
        Link::ToCategory((int)$continue_shopping['DepartmentId'],
                         (int)$continue_shopping['CategoryId'], $page);
    elseif (isset ($continue_shopping['DepartmentId']))
      $this->mLinkToContinueShopping =
        Link::ToDepartment((int)$continue_shopping['DepartmentId'], $page);
    else
      $this->mLinkToContinueShopping = Link::ToIndex($page);
  }

  if ($this->mProduct['image'])
    $this->mProduct['image'] =
      Link::Build('product_images/' . $this->mProduct['image']);

  if ($this->mProduct['image_2'])
    $this->mProduct['image_2'] =
      Link::Build('product_images/' . $this->mProduct['image_2']);
      $this->mLocations = Catalog::GetProductLocations($this->_mProductId);

      // Build links for product departments and categories pages
      for ($i = 0; $i < count($this->mLocations); $i++)
      {
        $this->mLocations[$i]['link_to_department'] =
          Link::ToDepartment($this->mLocations[$i]['department_id']);

        $this->mLocations[$i]['link_to_category'] =
          Link::ToCategory($this->mLocations[$i]['department_id'],
                           $this->mLocations[$i]['category_id']);
      }
    }
  }
  ?>
```

4. Add the following method to presentation/link.php. This function creates an associative array with the elements of the query string. We'll use this functionality to implement the Continue Shopping feature.

```
public static function QueryStringToArray($queryString)
{
  $result = array();

  if ($queryString != '')
  {
    $elements = explode('&', $queryString);

    foreach($elements as $key => $value)
    {
      $element = explode('=', $value);
      $result[urldecode($element[0])] =
         isset($element[1]) ? urldecode($element[1]) : '';
    }
  }

  return $result;
}
```

5. Modify presentation/products_list.php as shown in the following code snippet. This new code saves the last catalog page accessed. When displaying a product details page, this address will be used to create the Continue Shopping link.

```
// Class constructor
public function __construct()
{
...
...
```

```
    if ($this->mPage < 1)
      trigger_error('Incorrect Page value');

    // Save page request for continue shopping functionality
    $_SESSION['link_to_continue_shopping'] = $_SERVER['QUERY_STRING'];
  }
```

6. Modify `presentation/departments_list.php` by adding the highlighted code, which is also required for the Continue Shopping functionality:

```
  // 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'];
    elseif (isset($_GET['ProductId']) &&
            isset($_SESSION['link_to_continue_shopping']))
    {
      $continue_shopping =
        Link::QueryStringToArray($_SESSION['link_to_continue_shopping']);

      if (array_key_exists('DepartmentId', $continue_shopping))
        $this->mSelectedDepartment =
          (int)$continue_shopping['DepartmentId'];
    }
  }
```

7. Modify `presentation/categories_list.php` to load the department ID and category ID parameters from the Continue Shopping data saved in the session:

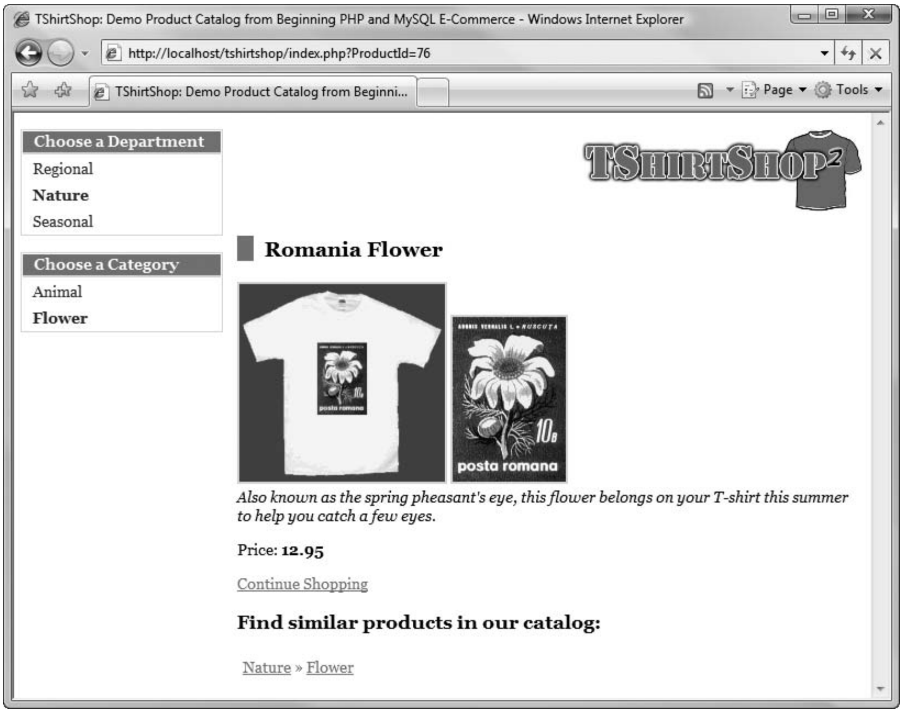
```
  // Constructor reads query string parameter
  public function __construct()
  {
    if (!isset($_GET['ProductId']))
    {
      if (isset ($_GET['DepartmentId']))
        $this->mSelectedDepartment = (int)$_GET['DepartmentId'];
      else
        trigger_error('DepartmentId not set');

      if (isset ($_GET['CategoryId']))
        $this->mSelectedCategory = (int)$_GET['CategoryId'];
    }
```

```
        else
        {
          $continue_shopping =
            Link::QueryStringToArray($_SESSION['link_to_continue_shopping']);

          if (array_key_exists('DepartmentId', $continue_shopping))
            $this->mSelectedDepartment =
              (int)$continue_shopping['DepartmentId'];
          else
            trigger_error('DepartmentId not set');

          if (array_key_exists('CategoryId', $continue_shopping))
            $this->mSelectedCategory =
              (int)$continue_shopping['CategoryId'];
        }
      }
```

8. Edit `presentation/store_front.php` to load the `product.tpl` template using the `$mContentsCell` member if the `ProductId` parameter exists in the query string. Also, if we've arrived to the product page from a department or category catalog page, we display the list of categories as well. This is a small feature but an important one for improving the catalog navigation experience of your visitor. Add the boldfaced lines to the `store_front.php` file as shown in the following code:

```
      public function init()
      {
        // Load department details if visiting a department
        if (isset ($_GET['DepartmentId']))
        {
          $this->mContentsCell = 'department.tpl';
          $this->mCategoriesCell = 'categories_list.tpl';
        }
        elseif (isset($_GET['ProductId']) &&
                isset($_SESSION['link_to_continue_shopping']) &&
                strpos($_SESSION['link_to_continue_shopping'], 'DepartmentId', 0)
                !== false)
        {
          $this->mCategoriesCell = 'categories_list.tpl';
        }

        // Load product details page if visiting a product
        if (isset ($_GET['ProductId']))
          $this->mContentsCell = 'product.tpl';
      }
    }
    ?>
```

9. Load the web site, and click the picture or name of any product. You should be forwarded to its details page. Figure 5-12 shows an example details page.


TShirtShop: Demo Product Catalog from Beginning PHP and MySQL E-Commerce - Windows Internet Explorer
http://localhost/tshirtshop/index.php?ProductId=76
TShirtShop: Demo Product Catalog from Beginni...
Page
Tools
TSHIRTSHOP2
posta romana

Choose a Department
Regional
Nature
Seasonal

Choose a Category
Animal
Flower

Romania Flower

Also known as the spring pheasant's eye, this flower belongs on your T-shirt this summer to help you catch a few eyes.

Price: **12.95**

Continue Shopping

Find similar products in our catalog:

Nature » Flower

Figure 5-12. *A product details page in TShirtShop*

How It Works: The product Componentized Template

As expected, the steps to implement the product details page were quite straightforward, since we only wrote or updated code that is already structured in a logical and consistent fashion. Implementing such features will become easier as you get used to the project structure in the following chapters.

What's interesting to note about this exercise are the way we used the session to implement the Continue Shopping link and the way we display the list of categories in the product details page. Whether a product page is visited from the front page of the catalog, is loaded directly in your browser, or is navigated to from an external web site, the list of categories doesn't show up. On the other hand, if you browsed a department or a category before loading a product details page, the department and category navigation elements show up in the product details page as well.

The code was explained throughout the exercise, so we'll not revisit it again. Read it again, and make sure you understand it clearly before moving on to the next chapter.

Summary

Congratulations! You've done a lot of work in this chapter! You finished building the product catalog by implementing the necessary logic in the data, business, and presentation tiers. On the way, you learned a great deal of theory, including

- Relational data and the types of relationships that can occur between tables
- How to obtain data from multiple tables in a single result set using `JOIN` and how to filter the results using `WHERE`
- How to display the list of categories and products depending on what page the visitor is browsing
- How to display a product details page and implement the Continue Shopping functionality
- How to implement paging in the products list when browsing pages containing many products

It's OK if you found much of the code quite complex, because it really is. Unless you're a PHP guru already, you should feel really proud of creating successfully a functional product catalog that features paging, categories and departments, product details pages, and many more smaller but equally important features!

Chapter 6 will be at least as exciting as this one, because you'll learn how to add attributes to products in your web site!