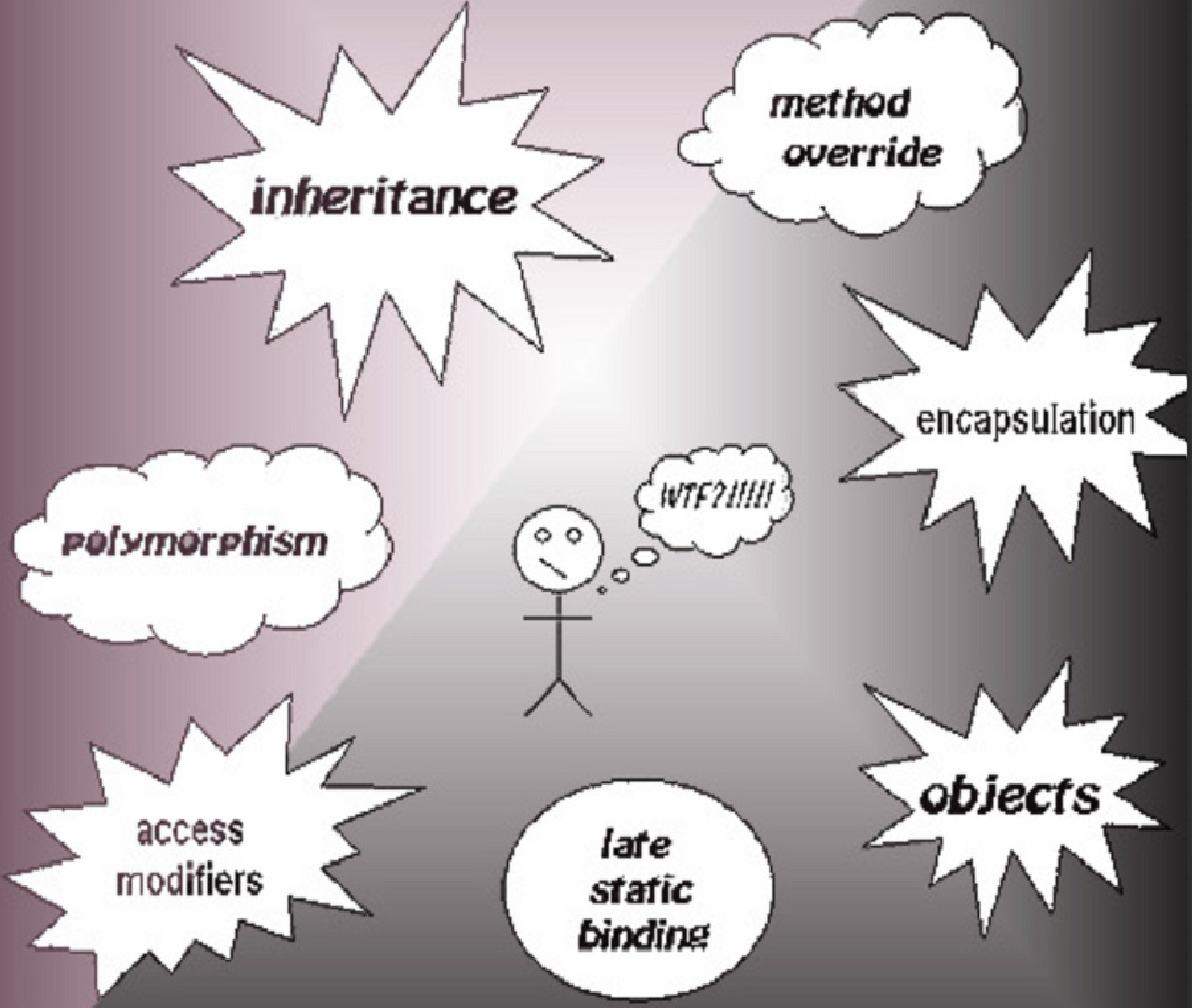
PHP This!

A Beginners Guide to Learning Object Oriented PHP



by Chad Collins

PHP This! A Beginners Guide to Learning Object Oriented PHP, First Edition

ISBN-13 (electronic): 9781456615291

Table of Contents

Introduction **6**

Why Read This Book 6
Sample Job Description: Jr. PHP Developer 7
The Eight Primary Categories of JQuery Features ....
Why Learn Object Oriented PHP 7
Six Primary Advantages to Learning Object Oriented Programming 7

Chapter 1: PHP Objects & Classes **9**
Overview – The Initial Confusion of First Learning Object Oriented Theory 10
Explanation of a Class 10
Explanation of an Object 10
Instantiation 12
`$this` Variable 12
Access Modifiers 12
Inheritance 14
Method Overriding 15
Invoking Parent Methods 16
Horizontal Inheritance - Using Traits 16
Encapsulation 17
Polymorphism 17
Polymorphism vs. Method Overloading 18
Polymorphism vs. Method Overriding 18
Late Binding / Dynamic Binding 19

Chapter 2: PHP Magic Methods **21**
`__construct()` 22
Invoking Parent Constructor and Destructor 22
`__destruct()` 23
`__get(), __set(), __isset()` 23
`__call()` 25
`__sleep()` 26
`__wakeup()` 27
`__clone()` 28

Chapter 3: Abstract Classes & Methods **30**
`abstract` Keyword 31
Extending sub-classes from an Abstract Base Class 32
Abstract Methods 33
`final` Keyword 34

Chapter 4: Interfaces **35**
PHP Interfaces 36
Explanation of What Interfaces Are and Why They are Useful 36
`interface` & `implements` Keywords 36
Implementing Multiple Interfaces 36
Programming to the Interface 37
Design-by-Contract 39

Chapter 5: Static Methods & Properties **43**
The `static` Modifier 44
The Scope Resolution Operator 44
Static Properties 45
Static Methods 48
Singleton Pattern 52
Late Static Binding 53
The `static` Keyword vs. the `self` Keyword 53

Chapter 6: PHP Error Control & Exception Handling 55
The Built-in Exception Class..56
Throwing an Exception 56
The try-catch-finally Block 58
Setting the Desired Error Sensitivity Level..59
Setting Error Reporting...59
Error Reporting Sensitivity Levels...59
Displaying Startup Errors.. 60
Logging Errors... 60
Identifying the Log File... 60
Setting the Maximum Log Line Length 60
Ignoring Repeated Errors... 60
Ignoring Errors Originating from the Same Location 60
Storing Most Recent Error in a Variable 61
Opening the Logging Connection...61
Logging Options 61
Closing the Logging Connection... 62

Chapter 7: The Model, View, Controller Design Pattern 69
Understanding the Model-View-Controller Design Pattern 69
Model 69
View 69
Controller 69
The MCV URL Structure & URL Mapping... 69
Using the .htaccess File 69
The index.php File 70
The MVC Folder Structure 70
Custom MVC Application - Restaurant Menu Management Application 70
Showing the Menu 75
Adding a Menu Item 82
Assigning a Menu Item to a Menu 86
Editing/Deleting Menu Items 90
Download the Source Code for the Custom MVC Application (Restaurant Menu Management Application) 94

Chapter 8: The Reflection API...95
Using the export Method 97
The Reflection Class 98

Chapter 9: The PEAR Library 102
Overview of PEAR 103
Install PEAR on a Zend Framework Environment on Windows...103
Install PEAR on a Windows XAMPP Environment 105
Install PEAR on a Windows WAMP Environment..106
Install PEAR on a Mac OS...106
Install PEAR on Linux...107
PEAR CAPTCHA Example 108

Chapter 10: Unit Testing with PHPUnit 112
Overview of Unit Testing with PHPUnit 113
Installing PHPUnit 113
Install PHPUnit on Windows in Zend Framework Environment 113
Install PHPUnit on Windows in XAMPP Environment.. 114
Install PHPUnit on a Mac OS 114
Install PHPUnit on Linux 114
Test fixtures 114
Test Phases.. 115
Writing Test Cases... 115
Assertion Methods... 115
Testing Data from HTML Form Submissions...120
Test suites 121
Mocks and Stubs.. 121
TestCase Matcher Methods 123

Running PHPUnit from Eclipse Without Plug-ins 124

Chapter 11: SVN - Source Code Management with Subversion 126
Windows 127
Setting Up a Subversion Server and Client on Windows 127
Adding Your Source Code the First Time 130
Creating a Copy to Work On 130
Updating and Committing Your Working Copy 130
Setting up TortoiseSVN 131
Trunk, Tags & Branches 132
Exporting a Release 136
Tagging & Branching a File 136
Mac OS X 137
Setting Up a Subversion Server and Client on Mac OS X 137
Download Subversion 137
Unpack and Install Subversion 137
Add Subversion to Your Path 137
Setting up the Subversion Repository 137
Creating a Subversion Project 137
Checking Out Files from Subversion 138
Updating and Committing Changes 138
Adding and Removing Files and Directories 139
Tagging & Exporting a Release 140
Branching & Merging a Project 140
Graphical User Interfaces 142
Linux 142
Setting Up a Subversion Server on a Linux Server 142
Download Subversion 142
Setting up the Subversion Repository 142
Creating a Subversion Project 143
Checking Out Files from Subversion 144
Updating and Committing Changes 144
Adding and Removing Files and Directories 145
Tagging & Exporting a Release 146
Branching & Merging a Project 147

Summary 149

Appendix A 150
Bibliography 150
Books 150
Web Resources 150

Appendix B 151
Setting Up Your Development Stack 151
AMP, XAMP/WAMP, MAMP, LAMP 151
Zend Server CE Download (MySQL, Apache, PHP, and ZendServer), System Requirements & Installation 151
MySQL Workbench 5.2 CE Download & Installation 155
Setting Up Your Integrated Development Environment (IDE) 156
Eclipse PDT (Helios Release) Download & Installation 156

Index 159

Introduction

A few years ago when I first sat down to evaluate the new features of PHP 5.3 I knew I was facing a challenging learning curve. It had been about 9 years since I had used PHP and at that time it was an easy to use, loosely typed, scripting language. My first thought while exploring the new features of PHP 5.3 was that “wow, little PHP is all grown up.”

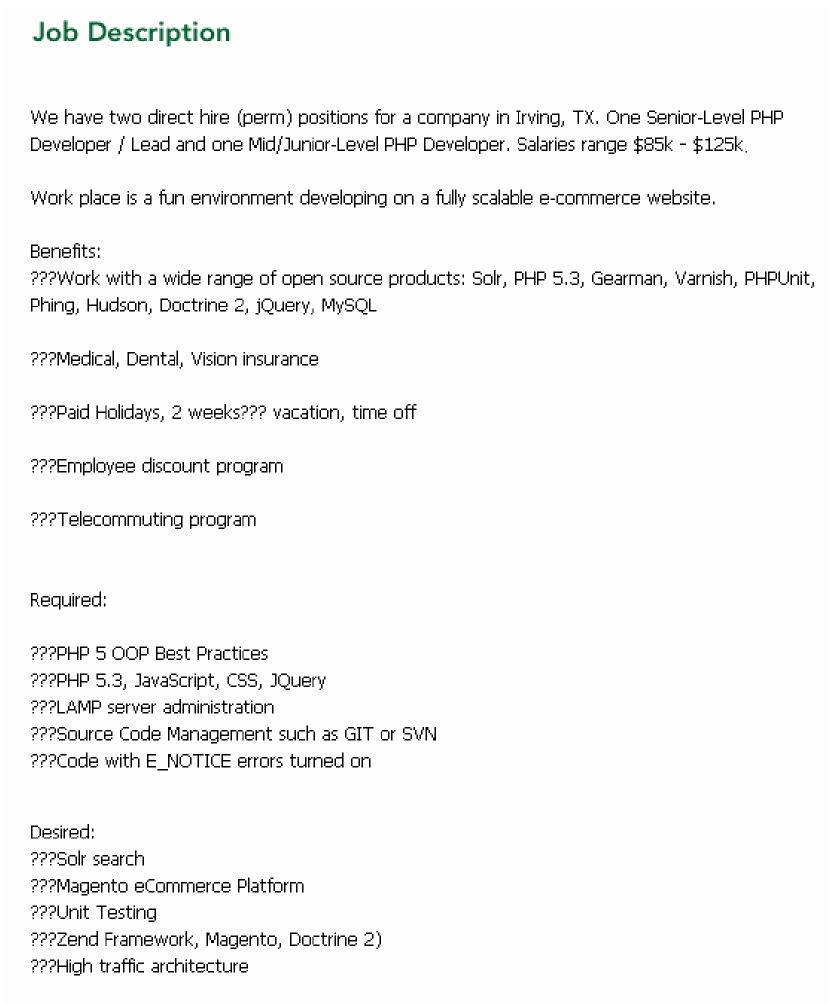
In an effort to get caught up in a hurry, I looked at several books and on-line forums and quickly grew frustrated with the same boring and monotonous way that these advanced concepts were explained and the poor examples that were used. Having an extensive background in object-oriented programming I continuously thought to myself “what if someone is trying to learn these concepts without much of an object-oriented background?”

The purpose of this book is to explain the advanced features and concepts of object-oriented PHP in a way that is simple and easy to understand and in a fashion that will hopefully help you remember and apply what you have learned.

If you are preparing for a job interview and you expect questions about advanced features of PHP 5+ this book will help you. In fact, you can be assured that most of the key concepts that are discussed in this book will come up in a job interview. I endeavored to include a few other features that you may be quizzed on in an interview as well.

Why Read This Book

This book is intended for developers who are new to Object Oriented PHP and perhaps Object Oriented Programming in general and are having difficulty grasping the new concepts. There are several reasons for reading this book beginning with the job description below. This book will help you understand and remember Object Oriented concepts as applied to PHP and give you a solid platform for continuous learning. If you ever find yourself in an interview for a job like the description below you will know what you are talking about and how to answer the fundamental questions about PHP that employers most likely will throw at you.


Job Description

We have two direct hire (perm) positions for a company in Irving, TX. One Senior-Level PHP Developer / Lead and one Mid/Junior-Level PHP Developer. Salaries range $85k - $125k.

Work place is a fun environment developing on a fully scalable e-commerce website.

Benefits:
???Work with a wide range of open source products: Solr, PHP 5.3, Gearman, Varnish, PHPUnit, Phing, Hudson, Doctrine 2, jQuery, MySQL

???Medical, Dental, Vision insurance

???Paid Holidays, 2 weeks??? vacation, time off

???Employee discount program

???Telecommuting program

Required:

???PHP 5 OOP Best Practices
???PHP 5.3, JavaScript, CSS, JQuery
???LAMP server administration
???Source Code Management such as GIT or SVN
???Code with E_NOTICE errors turned on

Desired:
???Solr search
???Magento eCommerce Platform
???Unit Testing
???Zend Framework, Magento, Doctrine 2)
???High traffic architecture

Figure 1-1. *Jr. PHP Developer Job Description*

This book focuses on the advanced features of Object Oriented PHP which can be applied to advanced development methods such as Extreme Programming, Design-by-Contract, Test Driven Development, Coding-to-the-Interface and Agile development. During the course of this brief book many simple examples will be used to illustrate key concepts and how they work. We will also build a small database driven application, a test harness for the application using PHPUnit, and add version control of our source code using Subversion (SVN).

In reference to the Job Description above, this book doesn't discuss JQuery or any other Javascript library. I do however encourage you to get a good JQuery book. Javascript was practically useless ten years ago as a cross browser, client-side development tool but has gained prominence and will continue to be pertinent for many years to come. When shopping for a JQuery book, keep in mind that JQuery's features are essentially broken down across eight major categories: Core Functionality, DOM Selection & Traversal, DOM Manipulation & CSS, Events, Effects & Animations, AJAX, User Interface and Extensibility through Plug-ins. It would be best to find a book that comprehensively covers each of these categories.

Why Learn Object Oriented PHP

There are several reasons to learn object-oriented programming. I will briefly touch on six primary advantages.

First of all, learning object-oriented programming, whether in PHP or any other OO-based language, will increase your value as a programmer which ultimately means you can make more money and have more flexibility on where you work.

Secondly, object-oriented principles are consistent across all of the object-oriented based languages and platforms. Once you have learned Object-Oriented PHP, grasping other languages will be much easier. Whether it is C#, Java, Visual Basic, Ruby on Rails or even C++, you will already know most of what you need to know. The syntax and semantics may differ slightly from language to language but the way they all work is essentially the same.

A third advantage to learning Object-Oriented programming is scalability. Using OO-based PHP for a small project may seem verbose and excessive when simple scripting can accomplish the same goal. However, if or when that small application grows bigger and bigger and becomes more complex the time spent writing the base code in Object-Oriented PHP will evidently prove to be a good investment.

A fourth advantage is that OO-based PHP creates re-usable blocks of code that can be used in your current project and in future projects. You can eventually accumulate a reusable and robust code library.

The fifth advantage to using OO-based PHP is that it is much easier to have multiple programmers working on the same project or code base because everything is segmented into objects. Various programmers can be assigned different objects to work on without worrying about touching or breaking code in other parts of the same project. This is considering that these objects are designed properly from the outset. It is this aspect of Object-Oriented PHP that makes PHP development in an SCRUM/Agile environment possible and allows robust tools like SVN and PHPUnit to be used. Learning how to use these tools in a team environment will certainly boost your value as a programmer.

The sixth reason for learning Object-Oriented PHP is that it allows you to leverage Object-Oriented based resources like PHP frameworks such as Zend and libraries such as PEAR. Learning how to use these resources effectively can help you build powerful applications faster and more efficiently. This is especially helpful if you are a freelance developer because you can produce robust applications faster and balance multiple projects easier which mean more income.

Chapter 1
PHP Objects & Classes

Initial Confusion

The real problem in explaining object-oriented programming is that there is no absolute linear or sequential path to doing so, the key concepts are all global. They say men are linear thinkers and women are global thinkers. Men tend to look for a starting point and put things in sequential order, doing a single task at a time. Women are global thinkers in that they are pervasive, considering the big picture. The fundamental concepts of object-oriented programming are global in that way versus straight procedural programming which is clearly linear.

This is the benefit of OOP however it doesn't help in explaining it to someone who is new to it. Most teachers will start with the 3 main advantages of OOP: **encapsulation**, **inheritance** and **polymorphism** and then elaborate on them. This can be confusing.

PHP Objects & Classes

At this point I would explain what an object is but you can't have an object without a class function. A class is essentially a blueprint for an object. It is a user-defined data type similar to a C-struct but more complex. A single instance of a class is an "object.”

A class defines characteristics that an object will have called properties. Properties describe an object and can be assigned values. For example, color is a property but “red” is a value assigned to color.

A class can also have member functions defined in them called methods. These functions manipulate the data values assigned to the properties within the class and control the behavior of the object created from the class. For example, a method can take a property value of an object, like the color of an object, and write it to the database, or visa versa.

Objects are at the very core of object-oriented programming so let me explain what an object is in the context of computer programming. **An object is simply data that is structured according to a user defined template that is defined in a class function.** Technically speaking, a class is a construct that defines constituent members and is used to create instances of itself. Simply speaking, a class function is essentially just a blueprint that defines the characteristics of an object, its properties, the way the object will behave, and its method functions. The properties describe and object and the methods process the data assigned to these properties.

An object can be almost anything you can imagine, a bird, a table, a house or a person. If the object is a person, then some common characteristics that can describe that person are name, gender, eye color, hair color, etc.

Before an object can exist it must have a class to define it. Our simple “Person” class will start off looking like this in PHP:

```
<?php

class Person
{

}

?>
```

This is as basic as it gets and all this does is define our class. At this point, our class is empty. It has no properties to describe the objects that will be created from it nor any methods that will give its objects the ability to do things and process data. Let’s add some basic characteristics that may describe a person as mentioned above:

```
class Person
{
        $name;
        $gender;
        $haircolor;
        $eyecolor;

}
```

Here we defined some properties that help describe a person but we can't do anything with these properties yet because we don't have any methods to set the value of these properties or any methods to get the values. Let's add some "setter" and "getter" functions to handle our properties. We will also add a method called **`__construct`** but we won't do anything with it at this time, that is, we won't implement it at this time. We'll talk about it in detail in Chapter 3 when we cover PHP Magic Methods:

```
<?php

class Person
{
        $name;
        $gender;
        $haircolor;
        $eyecolor;


        function setName($name)
        {
                $this->name = $name;
        }
        function getName()
        {
                return $this->name;
        }




        function setGender($gender)
        {
                $this->gender = $gender;
        }
        function getGender()
        {
                return $this->gender;
        }



        function setHairColor($haircolor)
        {
                $this->haircolor = $haircolor;
        }
        function getHairColor()
        {
                return $this->haircolor;
        }



        function setEyeColor($eyecolor)
        {
                $this->eyecolor = $eyecolor;
        }
        function getEyeColor()
        {
                return $this->eyecolor;
        }

        function __construct(){}

}
?>
```

Now the Person class above contains some properties that describe a person and some methods to set and get that information. Notice how the variable "**$this**" was used, we'll get back to it shortly.

Now that we have a person class, let's create some person objects from that class. Allow me to introduce some new friends, Mindi and Billy, they can be two instances (two objects) of the Person class. To create Mindi and Billy, we must instantiate them.

Instantiation

Let’s start with the term “instantiate.” To instantiate a class means to create an object based on that class, or to create an instance of that class. In PHP the way to do this is by using the **`new`** keyword like this:

$Mindi= **`new`** Person()
$Billy = **`new`** Person()

$Mindi and $Billy are now both instances (objects) of the class Person so their properties can be assigned specific values. Notice how I used the parenthesis after Person. This is because of the constructor I defined in the Person class, the first function just after the properties are defined. This constructor is not implemented in this example and I will explain this in detail in Chapter 3 when I discuss Magic Methods.

Let’s start with Mindi. We all know about her tattoo but let’s officially set her name and gender and some other attributes:

```
$Mindi->setName(‘Mindi’);
$Mindi->setGender(‘female’);
$Mindi->setHairColor(‘light brown’);
$Mindi->setEyeColor(‘blue’);
```

Now for Billy:

```
$Billy->setName(‘$Billy’);
$Billy ->setGender(‘male’);
$Billy ->setHairColor(‘brown);
$Billy ->setEyeColor(‘brown’);
```

$this Variable

The **$this** variable is a pointer to the object making the function call. That may not mean much to you so let’s explain it as it applies to Mindi. In the movie “Weird Science” two geeks created a metaphysical hot chic. That’s what we did when we instantiated $Mindi. Her blueprint was the Person class. We gave her dazzling blue eyes with the following method:

```
$Mindi->setEyeColor(‘blue’);
```

That method is defined in the Person class like so:

```
public function setEyeColor($eyecolor)
{
	$this->eyecolor = $eyecolor;
}
```

So **`$this`** was replaces with **`$Mindi`** at the moment `$Mindi` was instantiated because `$Mindi` is a copy of the Person class except with specific attributes instead of generic ones like $this. Hence the `$Mindi` object sees the `setEyeColor(‘blue’)` method like this:

```
public function setEyeColor(‘blue’)
{
	$Mindi->eyecolor = ‘blue’;
}
```

Access Modifiers

Access Modifiers are keywords that control visibility of class properties and methods. In PHP there are three of them:

1. Private
2. Protected
3. Public

Let’s describe Mindi in terms of her own class and let’s say that Mindi has a tattoo on her wrist. We could put it in PHP terms like this:

```
class Mindi {
$tattoo_location;

function setLocation() { }
}
```

When we do it this way, the tattoo is public by default. That means it can be seen anywhere in the program.

It's the same as writing it as:

```
class Mindi {
  public $tattoo_location;
}
```

This means class Billy or class Alex or class Dad can all see Mindi's tattoo because it has public visibility which is set by a public access modifier.

Now let's say Mindi instead has a tattoo that she doesn't want class mom or class dad to know about but wants to show off to her friends. She has this tattoo on her back so she can display it wearing a bikini top.

```
class Mindi {
  protected $tattoo_location;

  protected function setLocation(){}
}
```

Now Mindi can wear her bikini top around her friends and not her mom or dad. We can describe this in PHP by extending an invitation to the people she wants to show it to by using the `extends` key word:

```
class Billy extends Mindi {
      function viewTattoo(){}
}

class Alex extends Mindi....
class Anna extends Mindi....
class MathProfessor extends Mindi....
```

Since all of these classes extend class Mindi, they can see her tattoo. This is inheritance and all of these classes that extend Mindi are part of her inheritance hierarchy.

Now let's say she has her tattoo in a more intimate place (use your imagination) and she only wants her boyfriend Billy to see it and no one else. In PHP terms:

```
class Mindi {
private $tattoo_location;
private setLocation(){}

private letBillySee(){}
}
```

So to summarize the three PHP access Modifiers in a more formal way:

private restricts the access to the class itself. Only methods that are part of the same class can access private members. Let's look at an example below:

```
class Student {
       private $name;
       public $age;

       public function __construct($name, $age) {
              $this->name = $name;
              $this->age = $age;
       }
}

$tmpStudent = new Student ("Mindi","22");
echo "Name : " . $ tmpStudent->name;  //causes an error
```

The reason why data members are declared **private** is to avoid the outside programs to unintentionally modify the values without necessary validation. If you make a data member **private** you should provide public getter/setter methods to access the value of this data member. The *access modifier* **protected** allows the class itself and all of it's subclasses to access the data members and member functions. Global access is denied.

```
class Person {
       protected $name;
}

class Student extends Person {
       function setName($name) {
              //this works as $name is protected in Person
              $this->name = $name;
       }
}

$tmpStudent = new Student();
$tmpStudent->setName("Mindi");
$tmpStudent->name = "Mindi"; //this causes error as $name is protected and not public
```

public means that any code can access the member by its name.

```
class Student {
       private $name;
       public $age;

       public function __construct($name, $age) {
              $this->name = $name;
              $this->age = $age;
       }
}

$tmpStudent = new Student("Mindi","22");
echo "Age : " . $c->age; //prints 22
```

Now back to my previous point about object-oriented principles being "global" in the way a developer should think of them. I used Mindi's tattoo to explain *access modifiers* but I also had to demonstrate encapsulation and inheritance in order to do so.

Inheritance and Encapsulation are really just ways of regulating the ability of an object defined that is defined by a class.

Inheritance

Recall Mindi's tattoos, specifically the tattoo on her back? The tattoo on her back was described as protected in PHP terms meaning that it was visible within the Mindi class and to classes that *extended* the Mindi class. To inherit in PHP5, you should use the keyword `extends` in the class definition.

```
class Parent
{
}

class Child extends Parent
{

}
```

Inheritance passes "knowledge" down. It is a structured way of extending code and functionality into new programs and packages. Classes are created in hierarchies and inheritance allows the structure and methods in one class to be passed down the hierarchy. That means less programming is required when adding functions to complex systems. If a step is added at the bottom of a hierarchy, then only the processing and data associated with that unique step needs to be added. Everything else about that step is inherited. The ability to reuse existing objects is considered a major advantage of object oriented programming.

Method Overriding

Method overriding is when the function of base class is re-defined with the **same name, function signature and access modifier** (either public or protected) of the derived class.

The reason to override a method is to provide additional functionality over and above what has been defined in the base class. Imagine that you have a class by the name of Beverage from which you derive two child classes, Martini and Scotch. The Beverage class has methods defined to stir, mix, etc, but each of the specialized classes Martini and Scotch will have its own style of mixing and stirring and hence would need to override the stir and mix functionality.

Lets look at an example with Beverage:

```
class Beverage {
        public function stir() {
                echo "Stir method of Beverage Class called";
        }

        public function mix() {
                echo "Mix method of Beverage Class called";
        }

}

class Martini extends Beverage {
        public function stir() {
                echo "Stir method of the Martini Class called";
        }
        public function mix() {
                echo "Mix method of Martini Class called";
        }

}

class VodkaTonic extends Beverage {
        public function stir() {
                echo "Stir method of the VodkaTonic Class called";
        }
        public function mix() {
                echo "Mix method of VodkaTonic Class called";
        }
}

$m = new Martini();
$s = new VodkaTonic();

$m->stir();
echo "\n";
$vt->stir();
```

Output:
```
Stir method of the Martini Class called
Stir method of the VodkaTonic Class called
```

Invoking parent methods

First of all, "invoke" means to call, to request the action of something. I just wanted to get that brief explanation out of the way. Now, when you override a method of the base class, its functionality is completely hidden unless it has been explicitly invoked from the child class. To invoke a parent class method you should use the keyword **`parent`** followed by the **scope resolution operator "::** " followed by the name of the method as mentioned below:

```
        parent::function_name();
```

Look at the example below:

```
class Person {
        public function showData() {
                echo "This is Person's showData()\n";
        }
}

class Employee extends Person{
        public function showData() {
                parent::showData();
                echo "This is Employee's showData()\n";
        }
}

$c = new Employee();
$c->showData();

Output:
This is Person’s showData()
This is Employee’s showData()
```

In the above example, look at the way in which the `showData()` function in the Employee child class is invoking the Person parent class's `showData()` function. When the program executes the `showData()` method if the Employee class is called which in turn calls the `showData()` function of the parent class. After the parent class's `showData()` function completes its execution the remaining code in `showData()` function of the Employee class is executed.

Horizontal Inheritance – Using Traits

PHP does not support multiple inheritance; a class can only extend a single base class. What happens if you need code from more than one class? Until recently, developers had to accomplish this in a crude manner by copying and pasting code from additional classes that was needed outside of its base class. Now this can be done by way of a new feature introduced in PHP5.4 called a `trait` that provides a means of horizontal inheritance.

A `trait` looks like a class in the way it is structured but it cannot be instantiated. Properties and methods can be defined within a `trait` structure but they can only be used by a class that incorporates them with the `use` keyword which is called within the class that uses it, that is, it goes inside the curly braces. It isn't called in the class definition outside the curly braces like interfaces and the extending class (we will get to interfaces in Chapter 5).

```
trait Math
{
        public function add($num1,$num2)
        {
                return $num1 + $num2;
        }

        public function multiply($num1, $num2)
        {
                return $num1 * $num2;
        }
}
```

```
trait SalesTicket
{

        private $ticket = array();

        public function add($item)
        {
                array_push($this->ticket,$item);
        }

        public function getTicket()
        {
                return $this->ticket;
        }
}

class FoodOrder
{
        use Math,SalesTicket;
        {
        //Give priority to the add method from the Math trait
                Math::add insteadof SalesTicket;
        }

}

$order = new FoodOrder();
echo $order->add(5,7), "<br />";

Output:
12
```

Encapsulation

Encapsulation means what the term implies, to put in a “capsule,” to place functionality into a single package. Consider an object as a capsule or a box that has things in it. The access modifiers are like windows that control who gets to see inside with ‘public’ access being a wide open window that let's everyone see inside and ‘private’ access being fully shaded that doesn't let anyone see inside.

The control over the visibility of the contents of an object is a big part of how encapsulation works.

The contents of an object are properties and methods and because they are all packaged inside an object they make up a sort of mini-program that has control over how visible its contents, data, are. This is encapsulation.

Technically speaking, encapsulation refers to the creation of self-contained modules that bind processing functions to the data. Encapsulation ensures good code modularity, which keeps routines separate and less prone to conflict with each other. To reiterate, you can think of each object as a mini program that handles it’s own responsibilities as it’s own self contained capsule or pill.

Polymorphism

Now that you know that a class is a blueprint and that an object is constructed based on that class it is important to understand that many different objects can be created from the same class. I may have a single class for a table but I can create many different kinds of table objects from that class. I could have a round table, a square table, a long table a short table. I could have a blue table or a brown table. I could have a table made of oak or one made from plastic; all from the same base class.

Polymorphism simply means many forms. In computer programming, polymorphism is **the ability to create a variable, a function, or an object that has more than one form** in order to provide capability of adapting to what needs to be done. It is a mechanism by which objects of different types can process data through a single interface. Polymorphism is used to make applications more modular and extensible. Instead of messy, hard to read conditional statements describing different courses of action, you create interchangeable objects that you select based on what you need. That is the fundamental goal of polymorphism.

Polymorphism is a mechanism by which objects of different types can process data through a single interface.

Using a typical illustrative example, a program might define a type of object called Animal, containing a function called speak() and, through inheritance, derive object sub-types such as Cow, Dog, etc. that each have their own specific implementation of the speak() function. The mechanism of polymorphism allows such a program to call the speak() function without actually knowing whether it is calling it for a Cow, Dog, or any other derivative Animal, while executing the correct function for the specific type of Animal.

Polymorphism is not the same as method overloading or method overriding (in OOP, a method is a function that belongs to a class, while the class variables are referred to as its members) and should not be confused with these.

Method Overloading

Method overloading is where a class has two or more functions of the same name, but accepting a different number and/or data types of parameters.

Method Overriding

Method overriding is where a child class (one derived through inheritance) defines a function with the same name and parameters as a function in its parent class, but provides a different implementation of the function. This is not the same as polymorphism, as it does not necessarily involve late-binding, however overriding is part of the mechanism by which polymorphism is accomplished.

All this can sound confusing at first but a simple code example can help clarify:

```
class Animal {
   protected $name;

   public function __construct($name)
   {
      $this->name = $name;
   }

   public function getName()
   {
      return $this->name;
   }
}

class Cow extends Animal
{
   public function getName()
   {
      return 'Cow: '.parent::getName();
   }
}

$a = new Cow("Elsie Mae Mae");
echo $a->getName();
```

This is an example of overriding. It is different from polymorphism because we are declaring a Cow and directly calling a Cow function through Cow's interface. However, PHP is a loosely-typed language, meaning that the vast majority of data type handling is done implicitly by PHP, rather than the programmer. This means the difference between overriding and polymorphism in PHP can appear rather subtle. Consider a slightly different example:

```
<?php

class Animal {
   protected $name;

   public function __construct($name)
   {
      $this->name = $name;
   }

   public function getName()
   {
      return $this->name;
   }

   public function speak()
   {
      return $this->getName().' is speaking <br />';
   }
}

class Cow extends Animal
{
   public function getName()
   {
      return 'Cow: '.parent::getName();
   }
}


$a = new Cow("Elsie Mae Mae");
echo $a->speak();
$b = new Animal("something");
echo $b->speak();

?>
```

OUTPUT:

```
Cow: Elsie Mae Mae is speaking
something is speaking
```

Late Binding / Dynamic Binding

Late-binding (also known as dynamic binding) is the mechanism by which polymorphism is accomplished. In the example above, the `speak()` function is defined in the base class, but makes a call to `getName()`, a polymorphic function that has a specific implementation in each child class. Late-binding means that it is only at runtime, when the function is called, that the specific data type is bound to the request. In simple terms, even though `speak()` is defined only in the generic Animal class, it will call the child-specific implementation of `getName()`. This late-binding is a relatively new feature of PHP and is what makes true polymorphism possible.

```
Another example:

class Animal {
   protected $name;

   public function __construct($name)
   {
      $this->name = $name;
   }

   public function getName()
   {
      return $this->name;
   }
}

class Cow extends Animal
{
   public function getName()
   {
      return 'Cow: '.parent::getName();
   }
}

function Name(Animal $a)
{
   echo $a->getName();
}
```

```
$b = new Cow("Elsie Mae Mae");
Name($b);
```

This example is polymorphism, because the `Name()` function knows it is receiving an object of type Animal, but not whether it is a Cow, Dog or whatever else. Notice how we have specified type Animal as a parameter for the `Name()` function – specifying the data type of function parameters is normal and required in strongly-typed languages, but this type hinting is optional in PHP. Specifying a type hint will cause PHP to generate a fatal error if you attempt to pass the function a non-object, or object of a different type.

PHP is a dynamic programming language because it executes many common behaviors at run time that other languages might perform at compile time. It is loosely typed, or dynamically typed, in that a majority of the type checking is performed at run-time as opposed to being done at compile time. In dynamic typing values have types but variables do not, that it, a variable can refer to any value of any type.

Chapter 2
Magic Methods

The magic methods are reserved method names you can use in your classes to utilize special PHP functionality. Magic methods provide access to special PHP behavior. The naming convention used for magic methods is a lower/camel-case letters with two leading underscores.

__construct()

The constructor is a magic method that gets called when the object is instantiated, that is, at the moment an instance a class is created. It is usually the first thing in the class declaration but it does not need to be, it is a method like any other and can be declared anywhere in the class. Recall when I mentioned the parenthesis after the Person class name when we instantiated Mindi?

```
$Mindi = new Person();
```

The reference to the Person() class is the constructor for the $Mindi object. In this particular case the constructor is not implemented because there is nothing inside the parenthesis. By this I mean we don't initialize any of $Mindi's properties when we call the constructor. If we do not define a constructor function in our class then the PHP engine will assume that we are using an unimplemented constructor, that is:

```
public function __contruct(){}
```

Constructors also inherit like any other method. So if we consider our previous inheritance example from the Introduction to OOP, we could add a constructor to the Animal class like this:

```
// animal.php
class Animal
{
   public function __construct() {
        $this->created = time();
        $this->logfile_handle = fopen('/path/to/log.txt', 'w');
   }
}
```

Now we can create a class which inherits from the Animal class. Without adding anything into the Cow class, we can declare it and have it inherit from Animal, like this:

```
class Cow extends Animal {

}

$milky = new Cow();
echo $milky->created;
```

If we define a __construct method in the Cow class, then Cow objects will run that instead when they are instantiated. Since there isn't one, PHP looks to the parent class definition for information and uses that. So we can override, or not, in our new class – very handy.

Invoking parent Constructor and Destructor

We can get the parent PHP5 constructor and PHP5 Destructor to be invoked in the same way as invoking the parent method, refer to the example below:

```
class Person{
        public function __construct() {
                echo "This is Person's __construct()\n";
        }

        public function __destruct() {
                echo "This is Person's __destruct()\n";
        }
}

class Employee extends Person{
        public function __construct() {
                parent::__construct();
                echo "This is Employee's __construct()\n";
        }

        public function __destruct() {
                parent::__destruct();
                echo "This is Employee's __destruct()\n";
        }
}

$c = new Employee();

Output:
This is Person’s __construct()
This is Employee’s __construct()
This is Person’s __destruct()
This is Employee’s __destruct()
```

__destruct()

Did you spot the file handle that was also part of the constructor? We don’t really want to leave things like that lying around when we finish using an object and so the __destruct method does the opposite of the constructor. It gets run when the object is destroyed, either expressly by us or when we’re not using it any more and PHP cleans it up for us. For the Animal, our __destruct method might look something like this:

```
class Animal{

  public function __construct() {
    $this->created = time();
    $this->logfile_handle = fopen('/path/to/log.txt', 'w');
  }

  public function __destruct() {
    fclose($this->logfile_handle);
  }
}
```

__get(), __set(), __isset()

The setting and retrieval process of values of properties within a class that have no a explicit declaration comprises what's known as property overloading in PHP. The **`__set()`** function will be called automatically by the PHP engine each time a script attempts to assign a value to a property of a class that hasn't been explicitly declared. The **`__get()`** function will be invoked transparently when a script tries to retrieve the value of an undeclared class property. The **`__isset()`** function is also invoked when it is called on an undefined property. The purpose of **__isset()** is to test a property before working with it.

“User” example 1

Say that there's a basic class called "User," which represents, through software, a real-world user. The definition of this sample class would be something like this:

```
class User {
    private $firstname = 'John';
    private $lastname = 'Doe';
    private $email = 'janedoe@somedomain.com';
    // constructor (not implemented)
    public function __construct(){}

    // set user's first name
    public function setFirstName($firstname)
    {
           $this->firstname = $firstname;
    }

    // get user's first name
    public function getFirstName()
    {
           return $this->firstname;
    }
    // set user's last name
    public function setLastName($lastname)
    {
           $this->lastname = $lastname;
    }

    // get user's last name
    public function getLastName()
    {
    return $this->lastname;
    }
    // set user's email address
    public function setEmail($email)
    {
           $this->email = $email;
    }

    // get user's email address
    public function getEmail()
    {
           return $this->email;
    }
}
```

As you can see, the structure of the above "User" class is pretty easy to follow. It's only composed of a few setters and getters, used for setting and retrieving the values assigned to its three declared properties, that is `$firstname`, `$lastname` and `$email` respectively.

So far, nothing strange is happening here, right? What follows is a simple demonstration of how to use this class:

```
$user = new User();
$user->setFirstName('John');
$user->setLastName('Doe');
$user->setEmail('john@domain.com');

// display user data
echo 'First Name: ' . $user->getFirstName() . '<BR> Last Name: ' . $user->getLastName() . '<BR> Email: ' .
$user->getEmail();

/*
displays the following:
First Name: John
Last Name: Doe
Email: john@domain.com
```

“User” example 2

In the previous example, I created a basic "User" class, whose API allowed us to easily assign and retrieve the values of its declared properties. However, as I expressed earlier, it's possible to shorten its definition and make it more "dynamic" simply by concretely implementing the "__set()" and "__get()" methods.

To demonstrate this concept a bit more, I'm going to redefine this sample class to allow us not only to create new class properties at run time, but retrieve their respective values with extreme ease.

Now that you know what I'm going to do in the next few lines, please examine the modified version of the "User" class. It now looks like this:

```
class User
{
    // constructor (not implemented)
    public function _construct(){}

    // set undeclared property
    function __set($property, $value)
    {
            $this->$property = $value;
    }

    // get defined property
    function __get($property)
    {
            if (isset($this->$property))
            {
                    return $this->$property;
            }
    }
}
```

In the first case, the "__set()" function will create an undeclared property and assign a value to it, while in the last case, its counterpart "__get()" will retrieve this value if the property has been previously set.

The implementation of these two methods permits us to overload properties in a very simple way. It also makes the class's source code much shorter and more compact. The down side to this is that this process has some disadvantages that must be taken into account. First, and most importantly, the lack of an explicit declaration for each property of the class makes the code harder to read and follow. Second, it's practically impossible for an IDE like Eclipse PDT to keep track of those properties for either commenting or documenting them.

```
// example of usage of 'User' class with property overloading
$user = new User();
$user->firstname = 'Jane';
$user->lastname = 'Doe';
$user->email = 'janedoe@mydomain.com';
$user->address = 'My address 1111';

// display user data
echo 'First Name: ' . $user->firstname . ' Last Name: ' . $user->lastname . '<BR> Email: ' . $user->email .
'<BR>Address: ' . $user->address;
```

OUTPUT:

```
First Name: Jane
Last Name: Doe
Email: janedoe@mydomain.com
Address: My address 1111
```

__call()

If a class implements __call(), then if an object of that class is called with a method that doesn't exist __call() is used instead as a substitute. For example:

```
<?php
//This class demonstrate how the __call() Magic Method is used
class CallTest {
  private $v = array(1, 7, 9);

  public function __call($method, $args) {
    echo "The method \"$method\" was called.\n";
    var_dump($args);
    return $this->v;
  }
}

//test() doesn't exist
$example = new CallTest();
$a = $example->test('f', 'o', 'o');
var_dump($a);
?>
```

```
OUTPUT:
The method "test" was called.
array(3) {
  [0]=>
  string(1) "f"
  [1]=>
  string(1) "o"
  [2]=>
  string(1) "o"
}
array(3) {
  [0]=>
  int(1)
  [1]=>
  int(7)
  [2]=>
  int(9)
}
```

Now let's add a method called `test()` and run it again. Notice that this time `test()` was called instead of `__call()`:

```
<?php
//This class demonstrate how the __call() Magic Method is used
class CallTest {
  private $v = array(1, 7, 9);

  public function __call($method, $args) {
    echo "The method \"$method\" was called.\n";
    var_dump($args);
    return $this->v;
  }

  public function test($args) {
    echo "The method \"test\" exists now!\n";
    return NULL;
  }
}

//test() doesn't exist
$example = new CallTest();
$a = $example->test('f', 'o', 'o');
var_dump($a);
?>
```

```
OUTPUT:
The method "test" exists now!
NULL
```

__sleep()

The `__sleep()` **magic method** is called when the object of a class is about to be serialized. Serialization is the process of converting an object into a linear sequence of bytes for storage or transmission to another location over a network. This magic method __sleep() does not accept any parameter and returns an array. The array should contain a list of class members that should be serialized. This means that if you don't wish to serialize a particular class member, you should not include it in the array. Look at the example below:

```
class Employee {
```

```
          private $fname;
          private $date_of_birth;

          public function setFirstName($fname) {
                    $this->fname = $fname;
          }

          public function getFirstName() {
                    return $this->fname;
          }

          public function setBirthDate($dob) {
                    $this->date_of_birth = $dob;
          }

          public function getBirthDate() {
                    return $this->date_of_birth;
          }

          public function __sleep() {
                    return array("fname"); //because of this, only name is serialized
          }

}

$e = new Employee();
$e->setFirstName("Marsha");
$e->setBirthDate("09-12-1983");

$data = serialize($e)."\n";
echo $data."\n";
```

Output:

```
O:8:"Employee":1:{s:15:" Employee fname";s:6:"Marsha";}
```

In the above example, you can see that the serialized string data only contains the fname of the Employee Object. This is because the `__sleep()` magic method returned an array containing only the 'fname' data member.

`__wakeup()`

The `__wakeup()` magic method is the opposite of the `__sleep()` method and does not accept any parameter nor returns anything. It is responsible for setup operations after an object has been unserialized. Unserialized is the opposite of serialize which means to convert an array into a normal string that you can save in a file, pass in a URL, etc. To unserialize means to take a serialized string and convert it back to an array. The PHP functions `Serialize()` and `Unserialize()` are used to perform these operations. Look at the example below:

```
class Employee {
          private $fname;
          private $date_of_birth;

          public function setFirstName($fname) {
                    $this->fname = $fname;
          }

          public function getFirstName() {
                    return $this->fname;
          }

          public function setBirthDate($dob) {
                    $this->date_of_birth = $dob;
          }

          public function getBirthDate() {
                    return $this->date_of_birth;
          }

          public function __sleep() {
```

```
                    return array("fname");
          }

          public function __wakeup() {
               if($this->fname == "Marsha") {
                      $this->date_of_birth = "09-12-1983";
               }
          }
}

$e = new Employee();
$e->setFirstName("Marsha");
$e->setBirthDate("09-12-1983");

$data = serialize($e)."\n";
var_dump(unserialize($data));

OUTPUT:
object(Employee)#2 (2) {
  ["fname:private"]=>
  string(6) "Marsha"
  ["date_of_birth:private"]=>
  string(10) "09-12-1983"
}
```

In the above example, you can see that after the $e object has been serialized, that is converted to a common string, and the output stored in $data variable, we use the $data variable and pass it to the `unserialize()`. Before the object is unserialized and object created, the __`wakeup()` method is called.

__clone()

In PHP, objects are always assigned and passed around by reference. Let me explain with a simple example:

```
Class CopyClass
{
   public $name;
}

$first = new CopyClass();
$first->name="Number 1";
$second = $first;

echo "first = " . $first->name . "<BR><BR>";

$second->name="Number 2";

echo "first = " . $first->name . "<BR>";
echo "second = " . $second->name . "<BR>";

OUTPUT:
first = Number 1
first = Number 2
second = Number 2
```

Notice that the value of the `$first->name` property changed from "`Number 1`" to "`Number 2`" after I made a copy of the `$first` object called `$second` and set the value of `$second->name="Number 2."` This is because both the $first and $second objects actually refer to the same object, the $second object was assigned by reference.

The __`clone()` magic method can be used to make a copy of an object that doesn't refer to the object being copied but instead creates a new and completely distinct object. Let me demonstrate how __`clone()` works by using it in the previous example instead of setting `$second = $first`.

```
Class CopyClass
{
   public $name;
}

$first = new CopyClass();
$first->name="Number 1";
$second = clone $first;
echo "first = " . $first->name . "<BR><BR>";

$second->name="Number 2";

echo "first = " . $first->name . "<BR>";
echo "second = " . $second->name . "<BR>";

OUTPUT:
first = Number 1
first = Number 1
second = Number 2
```

This time the value of the `$first->name` did not change after we set a value for `$second->name`. That is because clone allowed us to make a copy of first that is a totally different object. Hence, now we have two objects not just two references to the same object like we had before.

Chapter 3
Abstract Classes & Methods

An integral part of polymorphism is the common interface. The common interface is a point of interaction between components. There are two ways to define a common interface in PHP: **interfaces and abstract classes**. Both have their uses, and you can mix and match them as you see fit in your class hierarchy.

There is more than one way to explain these features. Some prefer to describe the abstract class as a mix of the interface and the class. I would rather defer discussion about the PHP interface to Chapter 5 and explain the abstract class first.

The abstract class in PHP is a generic class that used solely as a base class, that is, an abstract class cannot ever be instantiated. An abstract class is a platform that is used to build more specific classes through inheritance. In other words, the methods and properties of the abstract class are implemented in the sub classes that inherit from the abstract class. You can't create an object from an abstract class but all of it's methods and properties have to be used by objects created from the classes that are based on the abstract class.

Before I confuse you any further, let's talk a little more about Mindi and her friend Vallery. The girls love to flaunt their stuff at the beach with their sexy but expensive swimwear.
Who'd have ever thought that such little clothing costs so much? Mind and Vallery are going to need jobs to maintain their flashy looks so let's help them out and give the girls jobs as waitresses.

It's a big help as a waiter or waitress to have good looks however looks aren't everything and any good waiter or waitress needs to ultimately know their menu. Let's go a step further and help the girls out by building a menu for them so they can do their job, make some money, buy those new bikinis and get back on the beach! Since a menu is comprised of menu items, let's start with the menu items.

Now since we want to give the girls flexibility on where they work and the type of food they serve, we want to create the ability to make any kind of menu that they'll need. We need to start off by being as generic or ***abstract*** as possible.

`abstract` Keyword

An abstract class or method is declared using the `abstract` keyword. To give us the most flexibility so that we can create as many kinds of menus and menuitems that we want we need to start with an abstract Menu class.

```
abstract class Menu
{

}
```

Well, we have an abstract class for our menu items but it doesn't do us any good because we don't have anything in it. Now let's think this through, for Mindi and Vallery's sake. Let's assume that there will be a database eventually so it follows that we will need a unique identifier for each of our menus and each of our menu items. There are several ways to do this but let's design our menu generator so that we have a general list of menu items that can be used on one or more menus (Lunch Menu, Dinner Menu, etc.). This implies a one-to-many entity relationship where there is one menuitemid to many menuids. Hence, the menu class will need a menuitemid field too. Let's also add name and description fields as well.

```
abstract class Menu
{
      private $menuid;
      private $menuitemid;
      private $menuname;
      private $description;

      public function setMenuID($menuid){$this->menuid = $menuid;}
      public function getMenuID(){return $this->menuid;}

      public function setMenuItemID($menuitemid){$this-> menuitemid = $ menuitemid;}
      public function getMenuItemID(){return $this-> menuitemid;}

      public function setMenuName($menuname){$this->menuname = $menuname;}
      public function getMenuName(){return $this->menuname;}

      public function setDescription($description){$this->description= $description;}
      public function getDescription(){return $this->description;}
}
```

It’s probably best that we don’t make the MenuItem class abstract because if we are going to make a general list, or table, of menu items that all have the same common structure we will want to instantiate the MenuItem class. Let’s give it some basic properties that we know each individual menu item will need, a unique ID, a name, description, price, serving size and a link to a picture. Let’s also give it some methods for setting and getting the values of these properties.

```
class MenuItem
{
      private $menuitemid;
      private $itemname;
      private $description;
      private $price;
      private $servingsize;
      private $picture;

      public function setID($menuitemid){$this-> menuitemid = $ menuitemid;}
      public function getID(){return $this-> menuitemid;}

      public function setPrice($price){$this->price = $price;}
      public function getPrice(){return $this->price;}

      public function setPicture($picture){$this->picture = $ picture;}
      public function getPicture (){return $this->picture;}

      public function setItemName($itemname){$this->itemname = $itemname;}
      public function getItemName(){return $this->itemname;}

      public function setDescription($description){$this->description= $description;}
      public function getDescription(){return $this->description;}

      public function setServingSize($servingsize){$this->servingsize = $servingsize;}
      public function getServingSize(){return $this->servingsize;}
}
```

Now that we have our base menu class, let’s build some specific menu classes that we will instantiate our actual menu objects from.

```
class MainMenu extends Menu
{
//...
}

class DrinkMenu extends Menu
{
//...
}

class LunchMenu extends Menu
{
//...
}

class KidsMenu extends Menu
{
//...
}

class DessertMenu extends Menu
{
//...
}
```

Let's examine the last two menu classes, the KidsMenu and DessertMenu classes. We probably don't want to leave these two classes open for extension by creating sub classes that can inherit from them, there is no need to. The Dessert Menu is usually the last thing anyone views when they are at a restaurant and the Kids Menu is self-explanatory. Just to save Mindi and Vallery any possible future confusion let's put a cap on the extensibility of these classes by making them final. We do this with the `final` keyword.

Abstract Methods

Recall from Chapter 2 in the discussion about polymorphism that you were introduced to the concept of Method Overriding. Well, Abstract methods are methods that must be overridden because they cannot be invoked directly. If you attempt to call an abstract method you will get an error. Abstract methods must be declared within an abstract class. If you declare an abstract method inside a non-abstract class this will cause a compile error.

In the following example we have an abstract class called Dog that has an abstract method called `bark()`:

```
<?php

abstract class Dog {

    private $name = null;
    private $gender = null;

    public function __construct($name) {
        $this->name = $name;
    }

    public function getName() {return $this->name;}
    public function setName($name) {$this->name = $name;}

    abstract public function bark();

}

// Override the abstract bark() method of the abstract Dog class

class Rottweiler extends Dog {

    public function bark() {
        echo "WOOOF!! WOOOF! WOOOF!" . "<br>";
    }

}

class Chihuahua extends Dog {

    public function bark() {
        echo "yip! yip! yip!" . "<br>";
    }
}


// This class causes a compilation error, because it fails to implement bark().
class Mutt extends Dog {
    // this will cause an error because this dog didn't override bark()
}


?>
```

OUTPUT:

```
Fatal error: Class Dog contains 1 abstract method and must therefore be declared abstract or implement the
remaining methods (Dog::bark)
```

`final` Keyword

```
final Class KidsMenu extends Menu
{
//...
}

final Class DessertMenu extends Menu
{
//...
              }
```

Inheritance allows for great flexibility within a class hierarchy. A class or a method can be overridden so that a call in a client method will achieve different effects according to which class instance it has been passed. Sometimes, like in the case of our Kids Menu and Dessert Menu, a class or a method should remain unchangeable. The `final` keyword stops inheritance. A final class cannot have sub-classes, or child classes and final methods cannot be overridden.

Chapter 4
PHP Interfaces

Interfaces are pure templates. The purpose of the interface is to give you flexibility by having your class be forced to implement multiple interfaces but still not allow multiple inheritance. As you recall from Chapter 2, there is no multiple inheritance in PHP. An interface acts as a contract in that any class that incorporates an interface commits to implementing all of the methods it defines. In other words, any classes implementing an interface must implement all methods defined by the interface.

PHP is a "loosely-typed" language, classes take on the type of the interface it implements as well as any class that it extends.

The use of interfaces allows a programming style called "*programming to the interface.*" The idea behind this is to base programming logic on the interfaces of the objects used rather than on internal implementation details. Programming to the interface reduces dependency on implementation specifics and makes code more reusable. It gives the programmer the ability to later change the behavior of the system by simply swapping the object used with another implementing the same interface.

An interface is similar to a class except that it cannot contain code. An interface can define method names and arguments, but not the contents of the methods. A class can implement multiple interfaces.

`interface` & `implements` Keywords

An interface is declared using the `interface` keyword and it is implemented using the keyword `implements`.

```
interface MyInterface {
}

class MyClass implements MyInterface {
}
```

Methods can be defined in the interface just like in a class, except without the body (the part between the braces).

```
interface MyInterface1 {
       public function doSomething();
}

interface MyInterface2 {
      public function doSomethingElse();
}

interface MyInterface3 {
       public function setName($name);
}
```

Multiple interfaces can be implemented by listing them separated with commas. All methods defined here will need to be included in any implementing classes exactly as described.

```
// VALID
class MyClass implements MyInterface1, MyInterface2, MyInterface3 {
    protected $name;
    public function doSomething() {
        // code that does something
    }
    public function doSomethingElse() {
        // code that does something else
    }
    public function setName($name) {
        $this->name = $name;
    }
}

// INVALID
class MyClass implements MyInterface1, MyInterface2, MyInterface3 {
    // missing doSomething()!

    private function doSomethingElse() {
        // this should be public!
    }
    public function setName() {
        // missing the name argument!
    }
}
```

If you have experience with C# this should be familiar to you. If not then you are probably wondering what is this good for? Recall when I stated that interfaces allow for a programming style called “programming to the interface” and that a PHP interface serves as a type of “contract?” Well you will most likely in your career encounter software architectural design methods that are domain driven, or use a design methodology called design-by-contract. The purpose for this type of design methodology is to make software code segments as modular and loosely-coupled as possible to minimize dependency between code segments. This makes it much easier for multiple developers to work on a single project without stepping on each others toes and causing conflicts in the software. This reduces the hassle of making code changes on a big project while reducing the risk of breaking down stream code in other places in the project. It also makes *unit testing* easier to accomplish so that classes can be tested by themselves as independent mini-programs.

To show first hand how the PHP interface is helpful, let’s continue on with our restaurant menu application. Recall that we developed a MenuItem class:

```
class MenuItem
{
      private $menuitemid;
      private $itemname;
      private $description;
      private $price;
      private $servingsize;
      private $picture;

      public function setID($menuitemid){$this-> menuitemid = $ menuitemid;}
      public function getID(){return $this-> menuitemid;}

      public function setPrice($price){$this->price = $price;}
      public function getPrice(){return $this->price;}

      public function setPicture($picture){$this->picture = $ picture;}
      public function getPicture (){return $this->picture;}

      public function setItemName($itemname){$this->itemname = $itemname;}
      public function getItemName(){return $this->itemname;}

      public function setDescription($description){$this->description= $description;}
      public function getDescription(){return $this->description;}

      public function setServingSize($servingsize){$this->servingsize = $servingsize;}
      public function getServingSize(){return $this->servingsize;}
}
```

We created an abstract Menu class:

```
abstract class Menu
{
      private $menuid;
      private $parentid;
      private $menuitemid;
      private $menuname;
      private $description;

      public function setMenuID($menuid){$this->menuid = $menuid;}
      public function getMenuID(){return $this->menuid;}

      public function setParent($parentid){$this->parentid = $parentid;}
      public function getParent(){return $this->parentid;}

      public function setMenuItemID($menuitemid){$this-> menuitemid = $ menuitemid;}
      public function getMenuItemID(){return $this-> menuitemid;}

      public function setMenuName($menuname){$this->menuname = $menuname;}
      public function getMenuName(){return $this->menuname;}
      public function setDescription($description){$this->description= $description;}
      public function getDescription(){return $this->description;}
}
```

We then created some classes that extended our Menu class:

```
class MainMenu extends Menu
{
//...
}

class DrinkMenu extends Menu
{
//...
}

class LunchMenu extends Menu
{
//...
}

final class DinnerMenu extends LunchMenu
{
//...
}

final class KidsMenu extends Menu
{
//...
}

final class DessertMenu extends Menu
{
//...
}
```

Now here is where Mindi and Vallery run into a dilemma. The Lunch Menu and the Dinner Menu have many of the same menu items but the serving sizes and prices should be different at dinner than they are at lunch. We can solve this by creating some interfaces that will ensure that dinner will be served up and priced up differently than lunch.

Now we can use the same menu for dinner that we use for lunch except we will adjust the serving sizes and prices for our dinner menu. We could start by extending our lunch menu to our dinner menu:

```
class DinnerMenu extends LunchMenu
{
//...
}
```

The next thing we need to do is to create the interfaces we need to force us to set up some business rules that will price and portion dinner items differently than lunch items. Let's go ahead and create an interface for Happy Hour too while we're at it.

```
interface DinnerPortion {
      public function setDinnerPortion();
}

interface DinnerPrices {
      public function setDinnerPrices();
}

interface HappyHourDrinkPrices {
      public function setHappyHourDrinkPrices();
```

```
  }
```

Now let’s put it all together:

```
 final class DinnerMenu extends LunchMenu implements DinnerPortion, DinnerPrices
 {
       public function setDinnerPortion($menuitemObject)
        {
              $adjusted_servingsize = 1;
                $base_servingsize = $menuitemObject->getServingSize();

                //Make the dinner portion 50% bigger than the lunch portion.
              $adjusted_servingsize = $base_servingsize * 1.5;

                              return  $adjusted_servingsize;

        }

        public function setDinnerPrices($menuitemObject)
        {
              $adjusted_price = 1;
                $base_price = $menuitemObject->getPrice();

                //Make the dinner price 25% more than the lunch price.
              $adjusted_price = $base_price * 1.25;

                              return  $adjusted_price;

        }
}
```

We’ll implement the HappyHourDrinkPrices interface on our DrinkMenu.

```
final class HappyHourMenu extends DrinkMenu implements HappyHourDrinkPrices
{
       public function setHappyHourDrinkPrices($drinkObject)
       {
              $adjusted_price = 1;
              $base_price = $drinkObject->getPrice();

              //Make the happy hour drink prices 30% less than regular price
              $adjusted_price = $base_price * 0.7;

                              return  $adjusted_price;

       }
}
```

The PHP interface is ultimately a mechanism to enforce discipline. Not only is it important to understand the interface as a contract but it is also important to understand it’s use as a language construct by thinking of them as means of classifying common traits or behaviors that were exhibited by potentially many non-related classes of objects. Let me attempt to drive this point home and demonstrate what it means to *program to the interface* with another example. This time let’s say that we are building a game application where a driver drives down a scary road and it equally threatened by two completely different things.

For example:

```
 class GrizzlyBear extends Animal {
   public function GrowlAtYou(){}
   public function ChaseYou(){}
   public function PounceOnYourCar(){}
   public function EatYou(){}
 }

 class Cop extends Person {
   public function FollowYou(){}
   public function PullYouOver(){}
   public function ArrestYou(){}
   public function BeatYouSenseless(){}
 }
```

Clearly, these two objects have nothing in common in terms of direct inheritance. But, you could say they are both threatening.

Let's say our game needs to have some sort of random thing that threatens the game player when they driving down a scary road. This could be a GrizzlyBear or a Cop or both, but how do you allow for both with a single function? And how do you ask each different type of object to "do their threatening thing" in the same way?

The key to realize is that both a GrizzlyBear and Cop share a common loosely interpreted behavior even though they are nothing alike in terms of modeling them. So, let's make an interface that both can implement:

```
 interface IThreat {
    public function BeThreatening();
 }


class GrizzlyBear extends Animal implements IThreat {
   function GrowlAtYou() { echo "Growwwwwwl!!"; }
   function ChaseYou(){}
   function PounceOnYourCar(){}
   function EatYou(){}

   function BeThreatening() {
      GrowlAtYou();
      ChaseYou();
      PounceOnYourCar();
   }
 }

 class Cop extends Person implements IThreat {
      function FollowYou(){}
      function PullYouOver(){}
      function ArrestYou(){}
      function BeatYouSenseless(){}

   function BeThreatening() {
      FollowYou();
      PullYouOver();
      ArrestYou();
   }
 }
```

We now have two classes that can each be threatening in their own way. And they do not need to derive from the same base class and share common inherent characteristics, they simply need to satisfy the contract of IThreat. That contract is simple, you just have to BeThreatening. In this regard, we can model the following:

```
class ScaryRoad {

  public function __construct() {}

   public function DriveCar(array $threatening_object) {
     //While driving down a scary road

      foreach ($threatening_object as $threat)
      {
        $threat->BeThreatening();
      }
   }
 }
```

```
//$cop1 = new Cop();
$cop1 = new Cop();
$grizzly1 = new GrizzlyBear();
$grizzly2 = new GrizzlyBear();

$road = new ScaryRoad();
$road->DriveCar($threats=array($cop1, $grizzly1, $grizzly2));
```

Here we have a scary road with a DriveCar() function that accepts a number of threats. Observe the use of the interface. This means that in our scenario, a member of the threats array could actually be a GrizzlyBear object or a Cop object.

The DriveCar method is called when a driver is driving down a scary road. In our application, that's when our threats do their work. Each threat is instructed to be threatening by way of the IThreat interface. In this way, we can easily have both GrizzlyBears and Cops be threatening in each of their own ways. We care only that we have something in the ScaryRoad object that is a threat, we don't really care what it is and they could have nothing in common with other.

Let’s look at a similar example:

```
class AttractiveStranger extends Person {
  public function LookAtYou(){}
  public function SmileAtYou(){}
  public function TalkToYou(){}
}

class Pepper extends Vegetable {
  public function BurnYourTongue(){}
  public function CauseBathroomEmergency(){}
}
```

Clearly, these two objects have nothing in common in terms of direct inheritance. But, you could say they both make you sweat.

Let's say our game needs to have some sort of random thing that makes our game player sweat when they are at a bar. This could be a AttractiveStranger or a Pepper or both, but how do you allow for both with a single function? And how do you ask each different type of object to "do their thing that makes you sweat" in the same way?

The key to realize is that both a AttractiveStranger and Pepper share a common loosely interpreted behavior even though they are nothing alike in terms of modeling them. So, let's make an interface that both can implement:

```
interface ISweat {
   function MakeYouSweat();
}


class AttractiveStranger extends Person implements ISweat {
  public function LookAtYou(){}
  public function SmileAtYou(){}
  public function TalkToYou(){}

  public function MakeYouSweat() {
    LookAtYou();
    SmileAtYou();
    TalkToYou();
  }
}

class Pepper extends Vegetable implements ISweat {
  public function BurnYourTongue();
  public function CauseBathroomEmergency();

  public function MakeYouSweat() {
    BurnYourTongue();
    CauseBathroomEmergency();
  }
}
```

We now have two classes that can each make you sweat in their own way. And they do not need to derive from the same base class and share common inherent characteristics -- they simply need to satisfy the contract of ISweat. In this regard, we can model the following:

```
class CollegeBar implements ISweat{

__construct()
{
   $attractivestranger = new AttractiveStranger();
   $hotpepper = new Pepper();

   $thing1 = $attractivestranger->MakeYouSweat();
   $thing2 = $hotpepper-> MakeYouSweat();

   $thingsThatMakeYouSweat = array($thing1, $thing2)

   SitAtBar($thingsThatMakeYouSweat);
}

   void SitAtBar() {
    // when you are sitting at the bar
        foreach ($thingsThatMakeYouSweat as $value)
        { . . . )

   }

 }
```

Here we have a college bar that accepts a number of people and a number of thingsThatMakeYouSweat. Observe the use of the interface. This means that in our scenario, a member of the thingsThatMakeYouSweat array could actually be a AttractiveStranger object or a Pepper object.

The SitAtBar method is called when a person is sitting at a bar. In our application, that's when our thingsThatMakeYouSweat do their work. Each thingsThatMakeYouSweat is instructed to make the person at the bar sweat by way of the ISweat interface. In this way, we can easily have both AttractiveStrangers and Peppers be threatening in each of their own ways. We care only that we have something in the CollegBar object that is a thingThatMakesYouSweat, we don't really care what it is and they could have nothing in common with other.

I hope that this helps to make the need and use of the PHP interface clear and also helps to explain what it really means to "program-to-the-interface."

Chapter 5
The `static` Modifier

The dictionary defines the work 'static' as pertaining to or characterized by a fixed or stationary condition, showing little or no change. The use of the word 'static' in Object-Oriented PHP means the same in that static properties and methods are never used in a dynamic way. Static properties and methods are defined in a class but are never encapsulated in objects that derived from that class. Let me repeat that, static properties and methods are accessible, if they are public, anywhere in the program without having to create an object from the class the static properties and methods are declared in. The static keyword is convenient for creating class-level utility methods and for sharing data between objects of the same type.

The use of the static keyword can be the source of confusion if you are learning about it for the first time. Some common questions that people have at first are: Are static properties the same as global variables? What is the difference between self and static keyword in static method? How do you call static properties and methods? Can you extend static methods? What are some practical uses for static properties and variables? These are all good questions and I will answer all of them through the course of this brief chapter. First, let me demonstrate how a static property compares to a dynamic property of an instantiated class. Let's revisit our Person class and create a new friend for Mindi and Vallery:

```
$tmpPerson = new Person();
$tmpPerson->Name = "Marsha";
```

We just created a new girl named Marsha by creating an instance of the Person class called Marsha. We then set a value to the 'Name' property of the tmpPerson object equal to "Marsha." Now, with the introduction of the static keyword in PHP, we can access methods and properties through the context of a class rather than only the object (note: these methods/properties need to be declared static first), that is, we can access the Name property without creating an object. For example:

```
class Person {

    static public $Name = "Marsha";

    static public function helloWorld() {

        print "Hello world from " . self::$Name;

    }

}
```

To declare a method or property as static, we must use the `static` keyword. When accessing properties from outside the class scope, we use the class name followed by two colons (::) and then the property name. The use of the twin colons is called the **Scope Resolution Operator**. For instance:

```
print Person::$Name . "\n";

Person::helloWorld();


Output:
Marsha
Hello world from Marsha
```

You should be able to see in the example above that you use the `self` keyword to access member variables or properties from inside a class.

As static methods are within the class scope, they cannot access any normal methods or properties (i.e. those that have not been declared static), as those would belong to the object, and not the class. A result of this is you cannot use the `$this` variable from within a static method because static methods are not invoked in the context of an object. Instead you would use `self`.

Let's get back to our restaurant menu application and add some static properties and methods that will be useful. Recall that we extended our abstract Menu class to create some specific menu classes. Let's add a static property to each of these classes to set the title that will be printed when we generate those specific menus. We will also employ static methods in a Database class that we will add to our application.

Static Properties

Look below in the MainMenu, DrinkMenu, LunchMenu, KidsMenu, DessertMenu, DinnerMenu, and HappyHourMenu classes. Let's add a `static public` property called `$title`. This static property will allows us to set the text to be used as the heading for our menus without creating the menu objects:

```
<?php
class MenuItem
{
		private $menuitemid;
		private $itemname;
		private $description;
		private $picture;
		private $servingsize;
		private $price;

		public function setID($menuitemid){$this-> menuitemid = $menuitemid;}
		public function getID(){return $this-> menuitemid;}

		public function setPrice($price){$this->price = $price;}
		public function getPrice(){return $this->price;}

		public function setPicture($picture){$this->picture = $picture;}
		public function getPicture (){return $this->picture;}

		public function setItemName($itemname){$this->itemname = $itemname;}
		public function getItemName(){return $this->itemname;}

		public function setDescription($description){$this->description= $description;}
		public function getDescription(){return $this->description;}

		public function setServingSize($servingsize){$this->servingsize = $servingsize;}
		public function getServingSize(){return $this->servingsize;}
}


abstract class Menu
{
		private $menuid;
		private $menuitemid;
		private $menuname;
		private $description;

		public function setMenuID($menuid){$this->menuid = $menuid;}
		public function getMenuID(){return $this->menuid;}

		public function setMenuItemID($menuitemid){$this-> menuitemid = $menuitemid;}
		public function getMenuItemID(){return $this-> menuitemid;}

		public function setMenuName($menuname){$this->menuname = $menuname;}
		public function getMenuName(){return $this->menuname;}

		public function setDescription($description){$this->description= $description;}
		public function getDescription(){return $this->description;}
}


class MainMenu extends Menu
```

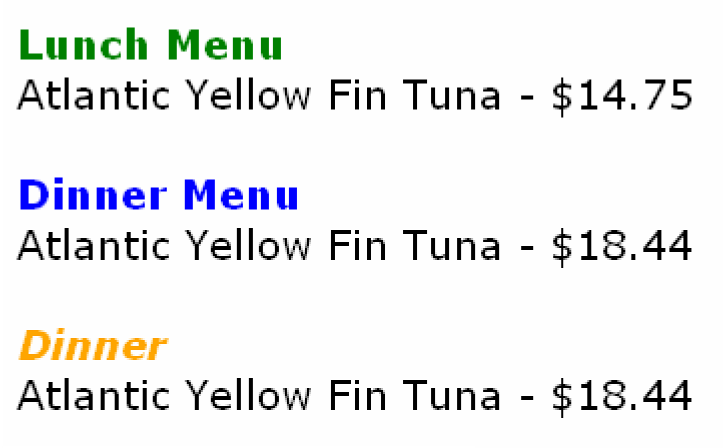
```
{
               static public $title="<b><font color=blue>Main Menu</font></b>";
}

class DrinkMenu extends Menu
{
               static public $title="<b><font color=yellow>Drink Menu</font></b>";
}

class LunchMenu extends Menu
{
               static public $title="<b><font color=green>Lunch Menu</font></b>";
}

Class KidsMenu extends Menu
{
               static public $title="<b><font color=orange>Kids Menu</font></b>";
}

Class DessertMenu extends Menu
{
               static public $title="<b><font color=red>Dessert Menu</font></b>";
}



interface DinnerPortion {
               public function setDinnerPortion($itemobject);
}

interface DinnerPrices {
               public function setDinnerPrices($itemobject);
}

interface HappyHourDrinkPrices {
               public function setHappyHourDrinkPrices($itemobject);
}



final class HappyHourMenu extends DrinkMenu implements HappyHourDrinkPrices
{
               static public $title="<b><font color=orange>Happy Hour Drink Menu</font></b>";

               public function setHappyHourDrinkPrices($menuitemObject)
               {
                       $adjusted_price = 1;
                       $base_price = $menuitemObject->getPrice();

                       //Make the dinner price 30% less than the normal price.
                       $adjusted_price = ($base_price * 0.7);

                       return  round($adjusted_price,2);
               }
}


final class DinnerMenu extends LunchMenu implements DinnerPortion, DinnerPrices
{
               static public $title="<b><font color=blue>Dinner Menu</font></b>";

               public function setDinnerPortion($menuitemObject)
               {
                       $adjusted_servingsize = 1;
                       $base_servingsize = $menuitemObject->getServingSize();

                       //Make the dinner portion 50% bigger than the lunch portion.
                       $adjusted_servingsize = $base_servingsize * 1.5;

                       return  $adjusted_servingsize;
               }

               public function setDinnerPrices($menuitemObject)
               {
                       $adjusted_price = 1;
                       $base_price = $menuitemObject->getPrice();
```

```
                        //Make the dinner price 25% more than the lunch price.
                        $adjusted_price = ($base_price * 1.25);

                        return  round($adjusted_price,2);
                }
}
```

Now let's apply the concepts we discussed in Chapters 2, 4, 5 and this chapter with a simple test. Remember that our Dinner Menu is extended from our Lunch Menu and that we implemented the DinnerPrices interface to make a distinction between the lunch prices and dinner prices. Also notice in the last segment of the following test code that I adjusted the title for the Dinner Menu, that is, I changed the `static public $title` property without creating an object to do it.

```
//Test
//Create Test Menu Item object
$tmpMenuItem = new MenuItem();
$tmpMenuItem->setItemName('Atlantic Yellow Fin Tuna');
$tmpMenuItem->setPrice('14.75');

echo LunchMenu::$title . "<BR>";
echo $tmpMenuItem->getItemName() . ' - $' . $tmpMenuItem->getPrice();
echo "<BR><BR>";

//Use the setDinnerPrice method we implemented in the DinnerMenu
// class that was declared in the DinnerPrices interface
$tmpDinnerMenu = new DinnerMenu();
echo DinnerMenu::$title . "<BR>";
echo $tmpMenuItem->getItemName() . ' - $' . $tmpDinnerMenu->setDinnerPrices($tmpMenuItem);
echo "<BR><BR>";

// Change the Title by changing the static $title variable in the DinnerMenu class.
DinnerMenu::$title="<font color=orange><i><b>Dinner</b></i></font>";
echo DinnerMenu::$title . "<BR>";
echo $tmpMenuItem->getItemName() . ' - $' . $tmpDinnerMenu->setDinnerPrices($tmpMenuItem);
echo "<BR><BR>";
?>
```

OUTPUT:



```
Lunch Menu
Atlantic Yellow Fin Tuna - $14.75

Dinner Menu
Atlantic Yellow Fin Tuna - $18.44

Dinner
Atlantic Yellow Fin Tuna - $18.44
```

Figure 6-1. *Eclipse PDT Debug Output*

Simplicity is often the best solution and answer to complex problems. In my experience I have encountered developers who will intentionally convolute their code into a big spaghetti mess intentionally for the sole purpose of making it hard for someone else to figure out. They of course do this in an attempt to make themselves more valuable and to assure job security. Let me assure you, if you learn this material and continue to develop your knowledge into design patterns and advanced software architecture, you will never have to pull that kind of stunt. Your value will be 'encapsulated' in your ability to understand a business model and requirements and react by developing from scratch, not from guarding some aging application like some troll.

With that said, let's discuss static methods. Static methods can be called directly from a class, not an object, just the same as static properties. As mentioned previously in this chapter, a common question surrounding the static modifier is what are static methods and properties good for? What are their practical uses?

For one, a database connection would be a good use for a static function. You don't need direct access to an entire DB object, you just need access to the connection resource. So you can call

```
$result = StaticClass::getDatabaseConnection()->query();
```

But if you need access to the class for storage later or multiple instances of the same object, then you would not want to go static.
Your class also now lives in a global scope, so you can access it from any class, in any scope, anywhere in your code base.

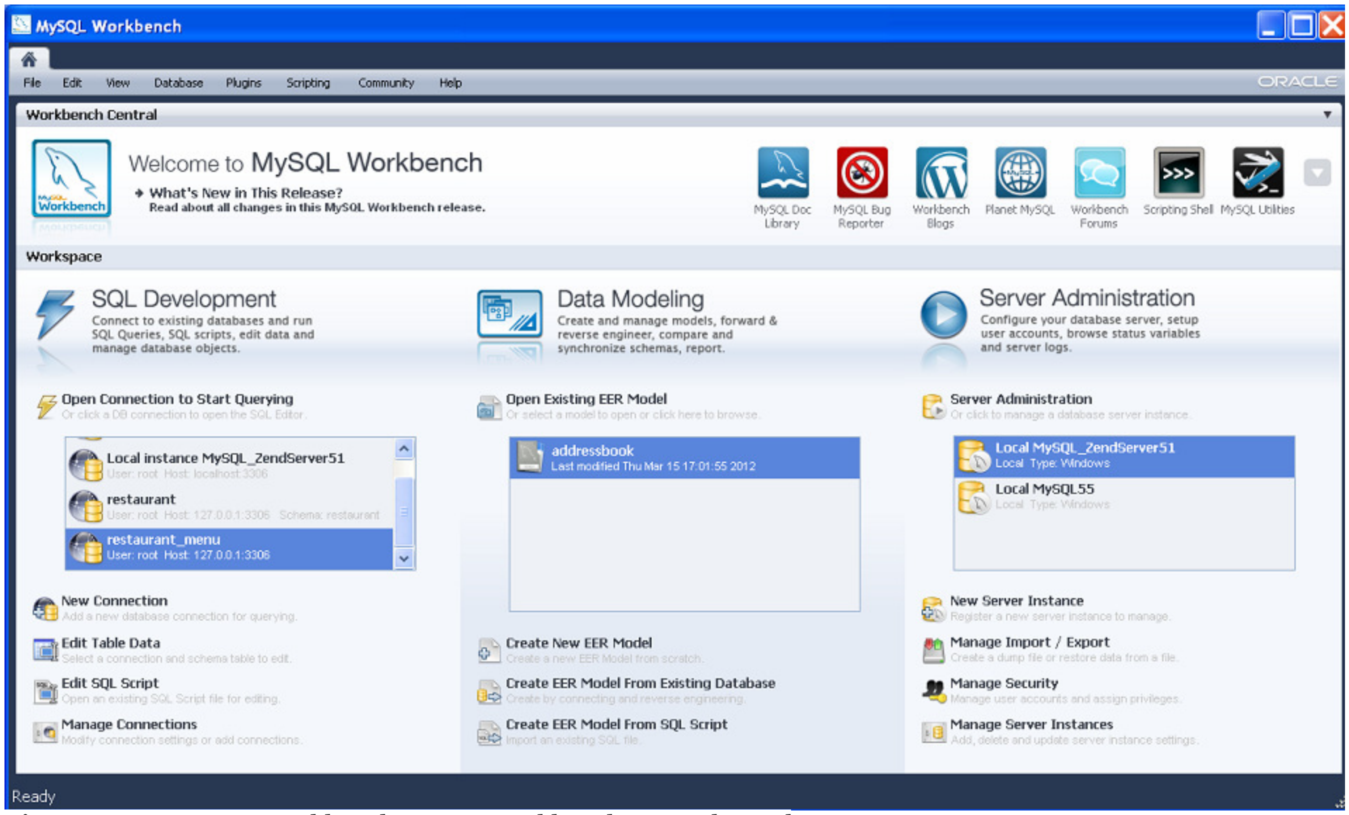
```
function getUsers()
{
     $users = StaticClass::getDatabaseConnection()->query('SELECT * FROM users');
}
```

Example: Use static methods to create instances of an object which might perhaps take different arguments.

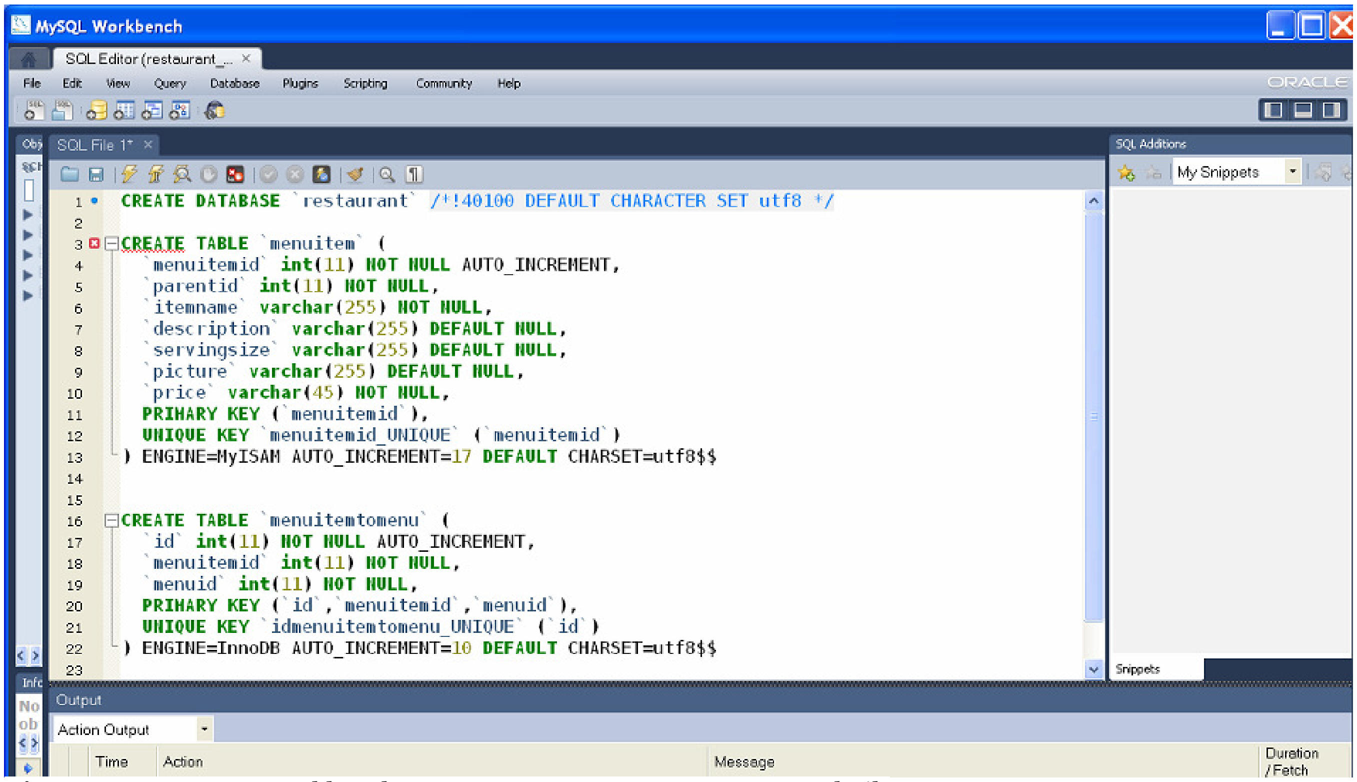
```
Class DBConnection
{
    public static function createFromConfiguration(Configuration $config)
    {
        $conn = new DBConnection();
        $conn->setDsn($config->getDBDsn());
        $conn->setUser($config->getDBUser());
        $conn->setPassword($config->getDBPass());

        return $conn;
    }

    public static function newConnection($dsn, $user, $password)
    {
        $conn = new DBConnection();
        $conn->setDsn($dsn);
        $conn->setUser($user);
        $conn->setPassword($password);

        return $conn;
    }
}
```

In the case of our restaurant menu application, we're going to need a database so let's create one with the following schema and then let's create a Database class to handle all of our database communication. I prefer to use MySQL Workbench Community Edition to perform my database administration. The most recent release is version 5.2. You can download and install MySQL Workbench for free at
http://dev.mysql.com/downloads/workbench/

Once you have MySQL WB 5.2 CE installed you will want to create a new connection by clicking on the "New Connection" link to open the Setup New Connection wizard and create a "localhost" connection.


MySQL Workbench
File Edit View Database Plugins Scripting Community Help
ORACLE
Workbench Central
Welcome to MySQL Workbench
What's New in This Release?
Read about all changes in this MySQL Workbench release.
MySQL Doc Library
MySQL Bug Reporter
Workbench Blogs
Planet MySQL
Workbench Forums
Scripting Shell
MySQL Utilities
Workspace
SQL Development
Connect to existing databases and run SQL Queries, SQL scripts, edit data and manage database objects.
Data Modeling
Create and manage models, forward & reverse engineer, compare and synchronize schemas, report.
Server Administration
Configure your database server, setup user accounts, browse status variables and server logs.
Open Connection to Start Querying
Or click a DB connection to open the SQL Editor.
Local instance MySQL_ZendServer51
User: root Host: localhost:3306
restaurant
User: root Host: 127.0.0.1:3306 Schema: restaurant
restaurant_menu
User: root Host: 127.0.0.1:3306
New Connection
Add a new database connection for querying.
Edit Table Data
Select a connection and schema table to edit.
Edit SQL Script
Open an existing SQL Script file for editing.
Manage Connections
Modify connection settings or add connections.
Open Existing EER Model
Or select a model to open or click here to browse.
addressbook
Last modified Thu Mar 15 17:01:55 2012
Create New EER Model
Create a new EER Model from scratch.
Create EER Model From Existing Database
Create by connecting and reverse engineering.
Create EER Model From SQL Script
Import an existing SQL file.
Server Administration
Or click to manage a database server instance.
Local MySQL_ZendServer51
Local Type: Windows
Local MySQL55
Local Type: Windows
New Server Instance
Register a new server instance to manage.
Manage Import / Export
Create a dump file or restore data from a file.
Manage Security
Manage user accounts and assign privileges.
Manage Server Instances
Add, delete and update server instance settings.
Ready
Figure 6-2. *MySQL Workbench 5.2 CE Workbench Control Panel*

```
CREATE DATABASE `restaurant` /*!40100 DEFAULT CHARACTER SET utf8 */

CREATE TABLE `menuitem` (
  `menuitemid` int(11) NOT NULL AUTO_INCREMENT,
  `parentid` int(11) NOT NULL,
  `itemname` varchar(255) NOT NULL,
  `description` varchar(255) DEFAULT NULL,
  `servingsize` varchar(255) DEFAULT NULL,
  `picture` varchar(255) DEFAULT NULL,
  `price` varchar(45) NOT NULL,
  PRIMARY KEY (`menuitemid`),
  UNIQUE KEY `menuitemid_UNIQUE` (`menuitemid`)
) ENGINE=MyISAM AUTO_INCREMENT=17 DEFAULT CHARSET=utf8$$


CREATE TABLE `menuitemtomenu` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `menuitemid` int(11) NOT NULL,
  `menuid` int(11) NOT NULL,
  PRIMARY KEY (`id`,`menuitemid`,`menuid`),
  UNIQUE KEY `idmenuitemtomenu_UNIQUE` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8$$
```


MySQL Workbench
SQL Editor (restaurant_...
File Edit View Query Database Plugins Scripting Community Help
SQL File 1*
SQL Additions
My Snippets
Snippets
Output
Action Output
Time Action Message Duration / Fetch

```
CREATE DATABASE `restaurant` /*!40100 DEFAULT CHARACTER SET utf8 */

CREATE TABLE `menuitem` (
  `menuitemid` int(11) NOT NULL AUTO_INCREMENT,
  `parentid` int(11) NOT NULL,
  `itemname` varchar(255) NOT NULL,
  `description` varchar(255) DEFAULT NULL,
  `servingsize` varchar(255) DEFAULT NULL,
  `picture` varchar(255) DEFAULT NULL,
  `price` varchar(45) NOT NULL,
  PRIMARY KEY (`menuitemid`),
  UNIQUE KEY `menuitemid_UNIQUE` (`menuitemid`)
) ENGINE=MyISAM AUTO_INCREMENT=17 DEFAULT CHARSET=utf8$$


CREATE TABLE `menuitemtomenu` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `menuitemid` int(11) NOT NULL,
  `menuid` int(11) NOT NULL,
  PRIMARY KEY (`id`,`menuitemid`,`menuid`),
  UNIQUE KEY `idmenuitemtomenu_UNIQUE` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8$$

```

Figure 6-3. *MySQL Workbench 5.2 CE RestuarantMenuManager.sql File*

```
<?php
//Database.php
require_once('config.php');

class Database
{
    public $connection;
                private function Database()
                {
                   $databaseName = $GLOBALS['configuration']['db'];
                   $serverName = $GLOBALS['configuration']['host'];
                   $databaseUser = $GLOBALS['configuration']['user'];
                   $databasePassword = $GLOBALS['configuration']['pass'];
                   $databasePort = $GLOBALS['configuration']['port'];

                   $this->connection = mysql_connect ($serverName.":".$databasePort, $databaseUser,
$databasePassword);
                        mysql_set_charset('latin1',$this->connection);
                        if ($this->connection)
                        {
                                if (!mysql_select_db ($databaseName))
                                {
                                throw new Exception('Cannot find: "' . $databaseName . '"');
                                }
                        }
                        else
                        {
                                throw new Exception('Cannot connect to the database.');
                        }
                }

                public static function Connect()
                {
                        static $database = null;
                        if (!isset($database))
                        {
                                $database = new Database();
                        }
                        return $database->connection;
                }

                public static function Reader($query, $connection)
```

```
		{
			$cursor = mysql_query($query, $connection);
			return $cursor;
		}

		public static function Read($cursor)
		{
			return mysql_fetch_assoc($cursor);
		}

		public static function Query($query, $connection)
		{
			$result = mysql_query($query, $connection);
			return mysql_num_rows($result);
		}

		public static function InsertOrUpdate($query, $connection)
		{
			$result = mysql_query($query, $connection);
			return intval(mysql_insert_id($connection));
		}
}
?>
```

At this point, let's "refactor" the menuitem class and give it the essence of abstraction without actually making it abstract by using the `__set()`, `__get()` and `__isset()` Magic Methods from Chapter 3.

```
Class MenuItem
{
		// constructor (not implemented)
		public function _construct(){}

		// set undeclared property
		function __set($property, $value)
		{
		$this->$property = $value;
		}

		// get defined property
		function __get($property)
		{
			if (isset($this->$property))
			{
				return $this->$property;
			}
		}
}
```

The MenuItem class is now cleaner and more flexible and will work smoothly with the restaurant database schema from above. Notice that we do not need a Menu table, only a MenuItem table and a bridge table called MenuItemToMenu that relates a specific menu to a menuitem. This design allows for a unique menuitem to be used on multiple menus and if you will recall from Chapter 4, the use of interfaces allows us to vary the prices and servingsizes of the menuitems in response to the particular menus that uses the menuitem. We will come back to this restaurant menu application in Chapter 7 where add exception handling capability as we put everything together.

Another good practical use of static methods and properties is to employ the Singleton design pattern.
The Singleton pattern is a design pattern that restricts the instantiation of a class to one object.

Singletons are useful in several ways. Primarily, Singletons are used often as a more preferred method of making data available across an application as an alternative to global variables. There are several reasons for this. For one, a class that relies on global variables becomes cumbersome to reuse from one application to another without first ensuring that the new application defines the same global variables. A second reason is that global variables will usually consume resources where a class that uses the Singleton pattern permits lazy allocation and initialization. A third reason that Singletons are preferred over globals is that globals clutter the global namespace, or the containing namespace in PHP5.3+. We will discuss the namespace feature in Chapter 8.

The Singleton pattern is also used to implement the Abstract Factory, Builder, Prototype and Façade design patterns. Design patterns are beyond the scope of this book but I have included several useful references for design patterns in the Appendix. I highly recommend reviewing these references as design patterns are an integral part of software development.

The singleton pattern is implemented by creating a class with a method that creates a new instance of the class if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made private. Also note observe that the `$instance` property is set to `private static`. In this example class, the `$instance` property will hold and return the instance of the Singleton object.

Singleton:

```
class SingletonClass {
    private static $instance;
    private function __construct() { }
    public function __clone() {
        trigger_error('Clone is not allowed.', E_USER_ERROR);
    }
    public static function init() {
        if (!isset(self::$instance)) {
            $c = __CLASS__;
            self::$instance = new $c;
        }
        return self::$instance;
    }
    // other public, dynamic methods for singleton
}

$singleton = SingletonClass::init();
```

Design patterns are beyond the scope of this book however I have provided some references in the Appendix that I highly recommend reviewing. This next example of how static properties and methods can be useful is to track the number of objects or instances of a class. This example extremely rudimentary, all it does is increment the `public static` property `$instance` every time the class constructor is called at the time of instantiation. Likewise it decrements `$instance` every time the destructor is called. This is possible because `public static $instance` persists in the context of the `ountInstancesOfMyself` class, not in the objects created from that class. It is in this way, and in the way we used our `public static $title` variables in the menu classes of our restaurant application that public static properties are used in a 'global variable' type fashion.

```
Class CountInstancesOfMyself {
    public static $instances = 0;
    public function __construct() {
        CountInstancesOfMyself::$instances++;
    }
    public function __destruct() {
        CountInstancesOfMyself::$instances--;
    }
}
$a = new CountInstancesOfMyself();
$b = new CountInstancesOfMyself();
$c = new CountInstancesOfMyself();

echo CountMe::$instances;
```

Output:
```
3
```

Late Static Binding

Late Static Binding describes a feature available in PHP 5.3 which can be used to reference the called class in a context of static inheritance. Ultimately, late static binding can be explained in these simple terms**: the `static` keyword vs. the `self` keyword.**

Both keywords are similar except `self` resolves to the containing class and `static` resolves to the invoked class.

Allow me to explain with a simple example. Say we have an empty house or apartment and we want to put some things in it. Let's model the stuff we want to put in our empty place as an abstract HouseholdObject and create some sub classes to give us the basic things our bachelor pad needs:

```
Abstract class HouseholdObject{

}


class Couch extends HouseholdObject {
      Public static function create() {
            return new Couch();
      }
}


class FlatScreenTV extends HouseholdObject {
      Public static function create() {
            return new FlatScreenTV ();
      }
}

class Refrigerator extends HouseholdObject {
      Public static function create() {
            return new Refrigerator ();
      }
}
```

Since the `create()` function is common to all of the classes that are extended from the `HouseholdObject` base class, it is more practical to move the `create()` function into the base class and then extend it to the sub classes so we only have to type it once. Notice in the `create()` function that we are using the `self()` keyword as a reference to the class:

```
Abstract class HouseholdObject{
      Public static function create() {
            return new self();
      }

}

class Couch extends HouseholdObject {

}

class Table extends HouseholdObject {

}

class Refrigerator extends HouseholdObject {
}
```

The problem with returning `self()` as a reference to the class is that when the `create()` function is called by the sub classes, using the scope resolution operator, we get an error:

```
Couch::create();
Table::create();
```

```
Output:
Fatal error: Cannot instantiate abstract class HouseholdObject
```

This is because we used the `self` keyword and it refers to the base class, the abstract HouseholdObject class. When the create() function is extended to the Couch, Table and Refrigerator classes and we call it to create a Couch or Table or Refrigerator it tries to create a HouseholdObject instead. The create() function is not referring to the class that is calling it, or invoking it, it is referring to the abstract class instead. As you hopefully recall, an abstract class cannot be instantiated, so the PHP engine throws an error.

Now, using late static binding let's try this again except this time the `create()` function will return a `static` reference. This static reference will refer to the class which will be calling the `create()` function, not the base class `HouseholdObject`:

```
Abstract class HouseholdObject{
      Public static function create() {
            Return new static();
      }
}

Class Couch extends HouseholdObject {

}

Class Table extends HouseholdObject {

}

Class Refrigerator extends HouseholdObject {

}
```

Chapter 6

PHP Error Control & Exception Handling

The topic of exception and error handling may be the least exciting topic of this book yet it is the one that may be the most critical because it can save your ass as a developer if you give it the attention it deserves. Errors are going to occur, they are inevitable. Chance however favors the prepared mind and as a developer if you give careful consideration to the problems that will most likely occur, you will have a much improved chance of releasing successful software. Incorporating useful tools like javascript form validation will help to prevent user induced errors. Developing your code to handle predicted errors and exceptions will help you to prevent an all out disaster. The release of PHP 5 has included some advantageous exception handling ability in addition to the PHP error reporting and configuration directives. We will review this exception handling ability first.

Numerous configuration directives determine PHP's error-reporting behavior. I will go ahead and discuss several of these last. If you find yourself getting bored then at least remember what they are and where they are in this book so you can reference them when you need them.

The Built in Exception Class

In PHP 5+ an exception is a special object that is instantiated from PHP's `Exception` class or any class that extends the PHP `Exception` class. The purpose of objects that are of type `Exception` are to contain and report information about errors. The constructor of the Exception class optionally accepts two arguments, a message string and an error code. The Exception class contains several public methods that are beneficial in evaluating error situations. I will quickly introduce the Exception class then I will provide an example of each methods use.

```
<?php
class Exception
{
    protected $message = 'Unknown exception';   // exception message
    private   $string;                          // __toString cache
    protected $code = 0;                        // user defined exception code
    protected $file;                            // source filename of exception
    protected $line;                            // source line of exception
    private   $trace;                           // backtrace
    private   $previous;                        // previous exception if nested exception

    public function __construct($message = null, $code = 0, Exception $previous = null);

    final private function __clone();           // Inhibits cloning of exceptions.

    Final public  function getMessage();        // Gets the message string that was passed to the constructor
    final public  function getCode();           // Gets the error code that was passed to the constructor
    final public  function getFile();           // Gets the source filename
    final public  function getLine();           // Gets the source line
    final public  function getTrace();          // Gets a multidimensional array of the backtrace()
    final public  function getPrevious();       // Gets previous exception
    final public  function getTraceAsString();  // Gets a formatted string of the data returned by trace

    /* Overrideable */
    public function __toString();               // formatted string for display
}
?>
```

Let's first reference the Database class that we developed for our restaurant menu application to interact with the database. Notice that we use the Exception class to print a custom message when connection attempt to the database fails either because the database with the name we specified in the configuration file is not found or it cannot connect because some other connection credential was wrong, which is also set in the configuration file.

Throwing an Exception

```
<?php
//Database.php

require_once('config.php');

class Database
{
       public $connection;

       private function Database()
       {
```

```
                $databaseName = $GLOBALS['configuration']['db'];
                $serverName = $GLOBALS['configuration']['host'];
                $databaseUser = $GLOBALS['configuration']['user'];
                $databasePassword = $GLOBALS['configuration']['pass'];
                $databasePort = $GLOBALS['configuration']['port'];

                $this->connection = mysql_connect ($serverName.":".$databasePort, $databaseUser,
$databasePassword);
                mysql_set_charset('latin1',$this->connection);
                if ($this->connection)
                {
                        if (!mysql_select_db ($databaseName))
                        {
                                throw new Exception('I cannot find the specified database "'.$databaseName.'".
Please edit configuration.php.');
                        }
                }
                else
                {
                        throw new Exception('I cannot connect to the database. Please edit configuration.php with
your database configuration.');
                }
        }

        public static function Connect()
        {
                static $database = null;
                if (!isset($database))
                {
                        $database = new Database();
                }
                return $database->connection;
        }

        public static function Reader($query, $connection)
        {
                $cursor = mysql_query($query, $connection);
                return $cursor;
        }

        public static function Read($cursor)
        {
                return mysql_fetch_assoc($cursor);
        }

        public static function NonQuery($query, $connection)
        {
                mysql_query($query, $connection);
                $result = mysql_affected_rows($connection);
                if ($result == -1)
                {
                        return false;
                }
                return $result;

        }

        public static function Query($query, $connection)
        {
                $result = mysql_query($query, $connection);
                return mysql_num_rows($result);
        }

        public static function InsertOrUpdate($query, $connection)
        {
                $result = mysql_query($query, $connection);
                return intval(mysql_insert_id($connection));
        }
}
```

Figure 7-1. *Try-Throw-Catch*

Typically you will set up your exception handling in your code within the construct of a try-catch block. The code that you intend to execute will reside inside a `try` block and the code that handles exceptions to normal execution of your code will go in the `catch` block. The `try` and `catch` keywords are called marker clauses and they define the scope of the exception handler. The unique aspect of the `catch` block is that it accepts an `Exception` object as an input argument that allows you to use the methods in the Exception class as you can see below:

```
class MyClass
{
    public function doSomething()
    {
       //...
        try
        {
            // try something
        }
        catch (Exception $e)
        {
            // handle/report exception
        }
    }
}
```

Now let's add a `finally` block to facilitate the output of the example function below and let's add one more feature to this construct called `throw` which will initiate the exception:

```
class MyClass
{
    public function doSomething()
    {

       $filename = '/path/to/test.txt';

        try
        {
          if !(file_exists($filename))
          {
               throw new Exception("file does not exist.");
          }
        }
        catch (Exception $e)
        {
               echo "Exception caught: " . $e->getMessage() . "<\ br>";
               echo "Exception code: " . $e->getCode() . "<\ br>";
        }
        finally
       {
               //create the file if it doesn't exist.
               $handle = fopen($filename, 'w');
```

```
                fclose($handle);
        }
    }
}
```

Setting the Desired Error Sensitivity Level

Pay strict attention to the following! The following code example shows you how to set your error reporting. Insert these two lines at the top of your code body. The second line is called the `error_reporting` directive. The `error_reporting` directive determines the reporting sensitivity level.

```
<?php

ini_set('display_errors',1);
error_reporting(E_ALL);

//code body ...

?>
```

Another way to do it is to edit your **php.ini** file and include this option:

```
error_reporting = E_ALL
```

To turn error reporting off for a single document, include this line:

```
error_reporting(0);
```

Fourteen separate levels are available, and any combination of these levels is valid. See the following table for a complete list of these levels. Note that each level is inclusive of all levels residing below it. For example, the `E_ALL` level reports any messages resulting from the 13 other levels residing below it in the table.

Error Reporting Sensitivity Levels:

Error Level	Description
E_ALL	All errors and warnings
E_COMPILE_ERROR	Fatal compile-time errors
E_COMPILE_WARNING	Compile-time warnings
E_CORE_ERROR	Fatal errors that occur during PHP's initial start
E_CORE_WARNING	Warnings that occur during PHP's initial start
E_ERROR	Fatal run-time errors
E_NOTICE	Run-time notices
E_PARSE	Compile-time parse errors
E_RECOVERABLE_ERROR	Near-fatal errors (introduced in PHP 5.2)
E_STRICT	PHP version portability suggestions (introduced in PHP 5.0)
E_USER_ERROR	User-generated errors
E_USER_NOTICE	User-generated notices
E_USER_WARNING	User-generated warnings
E_WARNING	Run-time warnings

Table 7-1. *PHP's Error-Reporting Sensitivity Levels*

Introduced in PHP 5, `E_STRICT` suggests code changes based on the core developers' determinations as to proper coding methodologies and is intended to ensure portability across PHP versions. If you use deprecated functions or syntax, use references incorrectly, use `var` rather than a scope level for class fields, or introduce other stylistic discrepancies, `E_STRICT` calls it to your attention.

Be aware that in PHP 6 you will need to set the `error_reporting` to `E_ALL` in order to view the user-generated warnings because `E_STRICT` is integrated into `E_ALL`.

In order to report all errors, which is handy during the development phase, consider setting the directive like this:

```
error_reporting = E_ALL
```

At times when you are only concerned about fatal run-time, parse, and core errors., you could use logical operators to set the directive as follows:

```
error_reporting E_ERROR | E_PARSE | E_CORE_ERROR
```

Now in order to set reporting on all errors except for user-generated ones, you would do the following:

```
error_reporting E_ALL & ~(E_USER_ERROR | E_USER_WARNING | E_USER_NOTICE)
```

Also not that the `error_reporting` directive uses the tilde character (~) to represent the logical operator `NOT`.

Displaying Startup Errors

Enabling the display_startup_errors directive will display any errors encountered during the initialization of the PHP engine. Like display_errors, you should have this directive enabled during testing and disabled when the site is live.

Logging Errors

Errors should be logged in every instance because such records provide the most valuable means for determining problems specific to your application and the PHP engine. Therefore, you should keep `log_errors` enabled at all times. Exactly to where these log statements are recorded depends on the error_log directive.

Identifying the Log File

Errors can be sent to the system syslog or can be sent to a file specified by the administrator via the `error_log` directive. If this directive is set to syslog, error statements will be sent to the syslog on Linux or to the event log on Windows.

If you're unfamiliar with the syslog, it's a Linux-based logging facility that offers an API for logging messages pertinent to system and application execution. The Windows event log is essentially the equivalent of the Linux syslog. These logs are commonly viewed using the Event Viewer.

Setting the Maximum Log Line Length

The `log_errors_max_len` directive sets the maximum length, in bytes, of each logged item. The default is 1,024 bytes. Setting this directive to 0 means that there is no maximum length set.

Ignoring Repeated Errors

Enabling `ignore_repeated_errors` causes PHP to disregard repeated error messages that occur within the same file and on the same line.

Ignoring Errors Originating from the Same Location

Enabling `ignore_repeated_source` causes PHP to disregard repeated error messages emanating from different files or different lines within the same file.

Storing Most Recent Error in a Variable

Enabling `track_errors` causes PHP to store the most recent error message in the variable `$php_errormsg`. Once registered, you can do as you please with the variable data, including output it, save it to a database, or do any other task suiting a variable.

If you've decided to log your errors to a separate text file, the Web server process owner must have adequate permissions to write to this file. In addition, be sure to place this file outside of the document root to lessen the likelihood that an attacker could happen across it and potentially uncover some information that is useful for surreptitiously entering your server.

The `define_syslog_variables()` function initializes the constants necessary for using the `openlog()`,`closelog()`, and `syslog()` functions. Its prototype follows:

void define_syslog_`variables(void)`

You need to execute this function before using any of the following logging functions.

Opening the Logging Connection

The `openlog()` function opens a connection to the platform's system logger and sets the stage for the insertion of one or more messages into the system log by designating several parameters that will be used within the log context. Its prototype follows:

```
int openlog(string ident, int option, int facility)
```

Several parameters are supported, including the following:

ident: Identifies messages. It is added to the beginning of each entry. Typically this value is
set to the name of the program. Therefore, you might want to identify PHP-related messages such as "PHP" or "PHP5."

`option`: Determines which logging options are used when generating the message. A list of available options is offered in Table 8-2. If more than one option is required, separate each option with a vertical bar. For example, you could specify three of the options like so:
`LOG_ODELAY | LOG_PERROR| LOG_PID`.

`facility`: Helps determine what category of program is logging the message. There are several categories, including `LOG_KERN`,`LOG_USER`,`LOG_MAIL`,`LOG_DAEMON`,`LOG_AUTH`,`LOG_LPR`, and `LOG_LOCALN`, where *N* is a value ranging between 0 and 7. Note that the designated facility determines the message destination. For example, designating `LOG_CRON` results in the submission of subsequent messages to the cronlog, whereas designating `LOG_USER` results in the transmission of messages to the `messages` file. Unless PHP is being used as a command-line interpreter, you'll likely want to set this to `LOG_USER`. It's common to use `LOG_CRON` when executing PHP scripts from a crontab. See the syslog documentation for more information about this matter.

Logging Options

Option	Description
LOG_CONS	If an error occurs when writing to the syslog, send output to the system console.
LOG_NDELAY	Immediately open the connection to the syslog.
LOG_ODELAY	Do not open the connection until the first message has been submitted for logging. This is the default.
LOG_PERROR	Output the logged message to both the syslog and standard error.
LOG_PID	Accompany each message with the process ID (PID).

Table 7-2. *Error Logging Options*

Closing the Logging Connection

The **`closelog()`** function closes the connection opened by `openlog()`. Its prototype follows:

```
int closelog(void)
```

The **`syslog()`** function is responsible for sending a custom message to the syslog. Its prototype follows:

```
int syslog(int priority, string message)
```

The first parameter, `priority`, specifies the `syslog` priority level, presented in order of severity here:

`LOG_EMERG`: A serious system problem, likely signaling a crash

`LOG_ALERT`: A condition that must be immediately resolved to avert jeopardizing system integrity

`LOG_CRIT`: A critical error, which could render a service unusable but does not necessarily place the system in danger

`LOG_ERR`: A general error

`LOG_WARNING`: A general warning

`LOG_NOTICE`: A normal but notable condition

`LOG_INFO`: A general informational message

`LOG_DEBUG`: Information that is typically only relevant when debugging an application

The second parameter, `message`, specifies the text of the message that you'd like to log. If you'd like to log the error message as provided by the PHP engine, you can include the string `%m`in the message. This string will be replaced by the error message string (`strerror`) as offered by the engine at execution time.

Now that you've been acquainted with the relevant functions, here's an example:

```
<?php
    define_syslog_variables();
    openlog("CHP7", LOG_USER);
    syslog(LOG_WARNING,"Chapter 7 example warning.");
    closelog();
?>
```

Referring back to our restaurant menu application, let's tie everything together and adding some exception handling code to the rest of the application. We already have a separate file called Database.php that contains our Database class which of course handles the interaction between the "restaurant" database and the rest of the application. The rest of the applications code is segregated into different files that divide the application between three main parts that can be described as, the "Model," the "View" and the "Controller."

The code that constitutes the "Model" is in MenuModel.php. This file contains all the classes that define the models for all of the specific menu objects that we want (i.e.: DrinkMenu, KidsMenu, DinnerMenu, etc.), the abstract menu class which is the base class for the specific menu classes and the menuitem class:

```
<?php
//MenuModel.php

require_once('/path/to/Database.php');

class MenuItem
{
        // constructor (not implemented)
        public function _construct(){}

        // set undeclared property
        function __set($property, $value)
        {
        $this->$property = $value;
        }

        // get defined property
        function __get($property)
        {
                if (isset($this->$property))
                {
                        return $this->$property;
                }
        }
}


abstract class Menu
{
        private $menuid;
        private $parentid;
        private $menuitemid;
        private $menuname;
        private $description;

        public function setMenuID($menuid){$this->menuid = $menuid;}
        public function getMenuID(){return $this->menuid;}

        public function setMenuItemID($menuitemid){$this-> menuitemid = $menuitemid;}
        public function getMenuItemID(){return $this-> menuitemid;}

        public function setMenuName($menuname){$this->menuname = $menuname;}
        public function getMenuName(){return $this->menuname;}

        public function setDescription($description){$this->description= $description;}
        public function getDescription(){return $this->description;}

        function getAllMenuItems($menuid)
        {
                $connection = Database::Connect();
                $this->query = "select mitm.*, mi.itemname, mi.description,
                                      mi.price, mi.picture, mi.servingsize
                                      from `menuitemtomenu`
                                      as mitm left join `menuitem`
                                      as mi on mitm.menuitemid = mi.menuitemid
                                      where mitm.menuid = $menuid";

                $menuitemList = Array();
                $cursor = Database::Reader($this->query, $connection);
                while ($row = Database::Read($cursor))
                {
                        $menuitem = new MenuItem;
                        $menuitem->menuitemid = $row['menuitemid'];
                        $menuitem->itemname = $row['itemname'];
                        $menuitem->price = $row['price'];
                        $menuitem->servingsize = $row['servingsize'];
                        $menuitem->description = $row['description'];
                        $menuitemList[] = $menuitem;
                }
                return $menuitemList;
        }

}

class MainMenu extends Menu
{
        static public $title="<b><font color=blue>Main Menu</font></b>";
        const id = 1;

        static public function menutime()
        {
                return  time();
        }
}

class DrinkMenu extends Menu
{
        static public $title = "<b><font color=lightblue>Drink Menu</font></b>";
        const id = 2;
}

class LunchMenu extends Menu
{
        static public $title="<b><font color=green>Lunch Menu</font></b>";
        const id = 3;
}

class KidsMenu extends Menu
{
        static public $title="<b><font color=yellow>Kids Menu</font></b>";
        const id = 5;
}

class DessertMenu extends Menu
{
        static public $title="<b><font color=red>Dessert Menu</font></b>";
        const id = 6;
}

class AppetizerMenu extends Menu
{
        static public $title="<b><font color=purple>Appetizer Menu</font></b>";
        const id = 7;
}


interface AdjustPortion {
        public function setDinnerPortion($itemobject);
}

interface AdjustPrice {
        public function setDinnerPrices($itemobject);
        public function setHappyHourDrinkPrices($itemobject);
}



final class HappyHourMenu extends DrinkMenu implements AdjustPrice
{
        static public $title="<b><font color=orange>Happy Hour Drink Menu</font></b>";

        public function setHappyHourDrinkPrices($menuitemObject)
        {
                $adjusted_price = 1;
                $base_price = $menuitemObject->getPrice();

                //Make the dinner price 30% less than the normal price.
                $adjusted_price = ($base_price * 0.7);

                return  round($adjusted_price,2);
        }

        public function setDinnerPrices($menuitemobject) {  }
}


final class DinnerMenu extends LunchMenu implements AdjustPortion, AdjustPrice
{
        static public $title="<b><font color=blue>Dinner Menu</font></b>";
        //DinnerMenu inherits ID from LunchMenu

        public function setDinnerPortion($menuitemServingSize)
        {
                $adjustment = 1;
                $adjusted_servingsize = "";
                $portion = explode(" ", $menuitemServingSize);

                //Make the dinner portions 50% bigger than the lunch portion.
                foreach ($portion as $subportion)
                {
                        if (is_numeric($subportion))
                        {
                        $adjustment = $subportion * 1.5;
                        $adjusted_servingsize = $adjusted_servingsize . " " . round($adjustment,2);
                        }
                        else
                        {
                        $adjusted_servingsize = $adjusted_servingsize . " " . $subportion;
                        }
                }

                return  $adjusted_servingsize;
        }


        public function setDinnerPrices($menuitemPrice)
        {
                $adjusted_price = 1;

                //Make the dinner price 25% more than the lunch price.
            try
            {
                  if ($menuitemPrice != 0)
                  {
                     $adjusted_price = ($menuitemPrice * 1.25);
                     return  round($adjusted_price,2);
                  }
                  else
                  {
                     throw new Exception('MenuItem Price is $0.0');
                  }
            }
            catch (Exception $e)
            {
                  echo "Caught exception: " .  $e->getMessage() . "\n";
            }
        }

        //Override the base method to use the AdjustPrice interface
        public function GetAllMenuItems($menuid)
        {
                $connection = Database::Connect();
                $this->query = "select mitm.*, mi.itemname, mi.description,
                                       mi.price, mi.picture, mi.servingsize
                                       from `menuitemtomenu`
                                       as mitm left join `menuitem`
                                       as mi on mitm.menuitemid = mi.menuitemid
                                       where mitm.menuid = $menuid";

            try
            {

            // Note: Recall that the throw new Exception(...) statement is already in the Database class in
Database.php.

                $menuitemList = Array();
                $cursor = Database::Reader($this->query, $connection);
                while ($row = Database::Read($cursor))
                {
                        $menuitem = new MenuItem;
                        $menuitem->menuitemid = $row['menuitemid'];
                        $menuitem->itemname = $row['itemname'];
                        $menuitem->price = self::setDinnerPrices($row['price']);
                        $menuitem->description = $row['description'];
                        $menuitem->servingsize = self::setDinnerPortion($row['servingsize']);
                        $menuitemList[] = $menuitem;
                }
                return $menuitemList;
            }
            catch(Exception $e)
            {
                echo 'Caught exception: ',  $e->getMessage(), "\n";
            }
        }

        public function setHappyHourDrinkPrices($menuitemObject){}
}


//DBMapper handles most of the communication between the model and the database
class DBMapper extends Menu
{

        public function Erase($menuitemid)
        {
           try
           {
                $connection = Database::Connect();
                $this->query = "DELETE mi.*, mitm.* FROM `menuitem` AS mi
                                LEFT JOIN `menuitemtomenu` AS mitm ON mitm.menuitemid = mi.menuitemid
                                WHERE mi.menuitemid = $menuitemid";

                $rows = Database::Query($this->query, $connection);
           }
           catch(Exception $e)
           {
                echo "Caught exception in Erase Method: " . $e->getMessage(), "\n";
           }
        }


        public function Save($data) {

           try
           {

                $connection = Database::Connect();

                if ( (isset($data["menuid"])) && (isset($data["menuitemid"])) )
                {

                // If the record doesn’t exist then we will INSERT it.
                $this->query = "select `menuitemid`, `menuid`
                                              from `menuitemtomenu`
                                              where `menuitemid`= " . $data['menuitemid'] .
```

```
                                        " and `menuid`= " . $data['menuid'] . " LIMIT 1";

            // Note: Recall that the throw new Exception(...) statement is already in the
            // Database class in Database.php

            $rows = Database::Query($this->query, $connection);
            if ($rows == 0)
                    {
                            $this->query = "insert into `menuitemtomenu` (`menuitemid`,`menuid`) values (
                            ". $data['menuitemid'] . ", " . $data['menuid'] . " )";
                    }
                    $insertId = Database::InsertOrUpdate($this->query, $connection);
            }
            else
            {

                    $menuitem = new MenuItem;
                    $menuitem->itemid = $data['menuitemid'];
                    $menuitem->itemname = $data['itemname'];
                    $menuitem->description = $data['description'];
                    $menuitem->servingsize = $data['servingsize'];
                    $menuitem->picture = $data['picture'];
                    $menuitem->price = $data['price'];

            // Check to see if the record already exists in the menuitem table.
            // If so we will UPDATE it, if not we will INSERT it.

                    $this->query = "select `menuitemid` from `menuitem`
                                    where `menuitemid`='" . $menuitem->itemid ."' LIMIT 1";

            // Note: Recall that the throw new Exception(...) statement is already in the
            // Database class in Database.php.
                    $rows = Database::Query($this->query, $connection);

                    if ($rows > 0)
                    {
                            $this->query = "update `menuitem` set
                            `itemname`='" . mysql_real_escape_string($menuitem->itemname) . "',
                            `description`='" . mysql_real_escape_string($menuitem->description) . "',
                            `servingsize`='" . $menuitem->servingsize . "',
                            `picture`='" . $menuitem->picture . "',
                            `price`='" . $menuitem->price . "' where `menuitemid`='".$menuitem->itemid."'";
                    }
                    else
                    {
                            $this->query = "insert into `menuitem`
                            (`itemname`,`description`,`servingsize`,`picture`,`price`) values (
                            '".mysql_real_escape_string($menuitem->itemname)."',
                            '".mysql_real_escape_string($menuitem->description)."',
                            '".$menuitem->servingsize."',
                            '".$menuitem->picture."',
                            '".$menuitem->price."' )";
                    }
                    $insertId = Database::InsertOrUpdate($this->query, $connection);
            }

        }
        catch (Exception $e)
        {
            echo "Caught exception in Save method: " . $e->getMessage(), "\n";
        }
    }


    public function getMenuItemTable()
    {
            $connection = Database::Connect();
            $this->query = "select menuitemid, itemname, description,
                                    price, picture, servingsize
                                    from `menuitem`";

            $menuitemList = Array();
            $cursor = Database::Reader($this->query, $connection);
            while ($row = Database::Read($cursor))
            {
                    $menuitem = new MenuItem;
                    $menuitem->menuitemid = $row['menuitemid'];
                    $menuitem->itemname = $row['itemname'];
                    $menuitem->price = $row['price'];
                    $menuitem->servingsize = $row['servingsize'];
                    $menuitem->description = $row['description'];
                    $menuitemList[] = $menuitem;
            }
            return $menuitemList;
    }

}
?>
```

Now to set the error reporting for an entire application, just insert the following two lines in the index.php file. The following command will report all errors:

```
<?php
//index.php

ini_set('display_errors',1);
error_reporting(E_ALL);

//...more code

?>
```

Chapter 7
Model – View – Controller

Understanding the Model-View-Controller Design Pattern

Model-View-Controller (MVC) design pattern expresses the separation of a software architecture's domain model, presentation, and actions based on user input into three distinct areas:

Model

The Model is how the underlying data is structured. The model manages the behavior and data of the application domain.

View

The View is what is presented to the user or consumer. The View is a visual representation of the model that manages the display of information by requesting information about the Model's state.

Controller

The Controller is the element that performs the processing based on user input. The Controller gives instructions to the Model to change the Model's state.

Separating these three elements makes it easier to achieve loose coupling, because it makes it possible for the controller to work with multiple different Model and View components. Both the View and the Controller depend on the Model. The Model, on the other hand, does not depend on either the View or the Controller. This distinct relationship is one the key benefits of the separation. This separation allows the model to be developed and tested independent of the visual presentation.

PHP has a several popular and high quality MVC frameworks to choose from, each with their own assortment of pros and cons to consider. However, in order the avoid any unnecessary confusion, we won't dive into any one of the popular frameworks at this point, instead we will develop our own custom MVC framework. As you recall from Chapter 7, we already have developed our data model for our restaurant menu management application which is in the MenuModel.php file and contains the base Menu class, MenuItem class, the DBMapper class which extends the base Menu class and all of the derived sub-Menu classes which also extend the base Menu class. Now we will integrate this into a custom MVC framework and add our View and Controller elements.

The MVC URL Structure & URL Mapping

To begin, you have no doubt seen the very recognizable URL structure that applications developed in the MVC framework operate on:

http://domain/controller/action/id

To do this for our Restaurant Menu Management (MVC framework) application, we'll need to utilize .htaccess URL re-writing with Apache.

We'll need to create a file named **.htaccess** (note the period at the start). This file will sit in our site's root directory. The purpose of this file is to configure Apache and its modules on a per site/directory basis. In our case, **.htaccess** will look like this:

```
Options +FollowSymLinks
RewriteEngine on
RewriteRule ^([a-zA-Z]*)/?([a-zA-Z]*)?/?([a-zA-Z0-9]*)?/?$
index.php?controller=$1&action=$2&id=$3 [NC,L]
```

The first two lines are preparing Apache for our new rewrite rule, which we're defining using the RewriteRule command. This command first has a regular expression and is then followed by a generic URL with some variables in it. Effectively what this rule is stating is that any URL that a user requests that matches this regular expression should be converted to the specified true URL at a web server level. The user never sees this conversion. Any matches inside the regular expression's round brackets are given a variable from left to right. The first variable is $1, the second is $2, etc. This specific rule also allows for anything after the controller to be left off the URL (the question marks handle this).

The index.php File

Our web server will point to index.php in the root directory as it will be the single point of entry for all requests. The index.php file is where we will set up the error reporting for the application. The index.php file will establish an "interpreter" object which is responsible for translating the requested URL into an instance of the relevant controller class, and then executing the requested action (method in that controller class) on that controller instance. We will come back to the index.php file after I have explained a few other things first.

```
<?php
//index.php

ini_set('display_errors',1);
error_reporting(E_ALL);


//reference the general classes
require("helperclasses/Interpretor.php");
require("helperclasses/BaseController.php");
require("helperclasses/Template.php");
require_once("helperclasses/Database.php");

// reference the data model classes here
require("models/MenuModel.php");

//reference the model classes here

//reference the controller classes here


//create the controller and execute the action
$interpreter = new Interpreter($_GET);
$controller = $interpreter->CreateController();
$controller->ExecuteAction();

?>
```

MVC Folder Structure

All controllers, models and views are individual files organized in relevant folders of the same name. The views folder has a subfolder for each controller. Controllers and models files are individual PHP classes. Views files are PHP capable, but in this example they will be pure HTML. Our basic folder structure will be the following:

```
/htdocs
     /models
     /views
     /controllers
     /helperclasses
     /images
     /database
```

We obviously will place our MenuModel.php file in the models folder. Our Database.php file will go in the helperclasses folder since it contains our Database class which facilitates the communication with the MySQL database.   The database directory will be used to store the actual .sql files that can be used to create our database, create the database schema (and load it with some test data).

Since the .sql files contain some test data already, the first thing we are going to do is develop a way to present or view that data. So let's build the model, controller and view files that we need to show our restaurant menu. Once we have done this we will be able to add the ability to edit and add menu items as well following the same pattern. Let's proceed with the development of our first **Custom MVC Application – A Restaurant Menu Management Application.**

We will call our first MVC element "show." To create this element we will need a "show" model, a "show" controller and a "show" view. The "show" controller will instantiate an instance of the "show" model which will provide the information to present in the "show" view but how does the "show" controller know how to do what it has to do based on what the user does? The user interacts with our application via a web browser. The web browser sends a request to a URL. Recall that an MVC URL looks like this:

http://domain/controller/action/id

Or more specific to our example:

http://localhost/show/Display

Our controller needs to receive interpreted input from the URL and execute commands accordingly. Since we will also have controllers to handle edit and add capabilities we will need a base class that receives URL input and relays commands which can be extended. It really shouldn't be the job of the base controller to interpret URL command so we will also need another utility class that does this work. Let's call this class Interpreter and put is in a utility class file called Interpreter.php

Let's start with the Interpreter class:

```
class Interpreter {
       private $controller;
       private $action;
       private $urlvalues;
       //store the URL values on object creation


       public function __construct($urlvalues)
       {
              $this->urlvalues = $urlvalues;
              if ($this->urlvalues['controller'] == "")
              {
                     $this->controller = "show";
              }
              else
              {
                     $this->controller = $this->urlvalues['controller'];
              }
              if ($this->urlvalues['action'] == "")
              {
                     $this->action = "index";
              }
              else
              {
                     $this->action = $this->urlvalues['action'];
              }
       }
```

An object instantiated from this class will break down the URL commands (with assistance from .htaccess) and interpret which specific controller is being requested and what action (method) in that controller class is being called. I hope you remember some key things from the previous chapters that may prompt you to ask some questions here. Unless we are calling a static method from our controller class (which won't work if we are to create several controllers that do different things), we need to create an instance of our controller object. It follows that we need to be able to instantiate a controller object specific to what the URL command is. Hence the Interpreter class needs a method to do this:

```
       //instantiate the requested controller as an object
       public function CreateController()
       {
              //does the class exist?
              if (class_exists($this->controller))
              {
                     $parents = class_parents($this->controller);
                     //does the class extend the controller class?
                     if (in_array("BaseController",$parents))
                     {
                            //does the class contain the requested method?
                            if (method_exists($this->controller,$this->action))
                            {
                                   return new $this->controller($this->action,$this->urlvalues);
                            }
                            else
                            {
                                   throw new Exception("Bad Method: " . $this->urlvalues);
```

```
                        }
                }
                else
                {
                        throw new Exception("Bad Controller: " . $this->urlvalues);
                }
        }
        else
        {
                throw new Exception("Bad Controller: " . $this->urlvalues);
        }
}
```

Let’s put all of this in a utility class file called Interpret.php and save it in the helperclasses folder

```
<?php
//Interpreter.php

class Interpreter {
        private $controller;
        private $action;
        private $urlvalues;
        //store the URL values on object creation


        public function __construct($urlvalues)
        {
                $this->urlvalues = $urlvalues;
                if ($this->urlvalues['controller'] == "")
                {
                        $this->controller = "show";
                }
                else
                {
                        $this->controller = $this->urlvalues['controller'];
                }
                if ($this->urlvalues['action'] == "")
                {
                        $this->action = "index";
                }
                else
                {
                        $this->action = $this->urlvalues['action'];
                }
        }


        //instantiate the requested controller as an object
        public function CreateController()
        {
           try
           {
                //does the class exist?
                if (class_exists($this->controller))
                {
                        $parents = class_parents($this->controller);
                        //does the class extend the controller class?
                        if (in_array("BaseController",$parents))
                        {
                                //does the class contain the requested method?
                                if (method_exists($this->controller,$this->action))
                                {
                                        return new $this->controller($this->action,$this->urlvalues);
                                }
                                else
                                {
                                        return new Exception("Bad Method: " . $this->urlvalues);
                                }
                        }
                        else
                        {
                                return new Exception("Bad Controller: " . $this->urlvalues);
                        }
                }
                else
                {
                        return new Exception("Bad Controller: " . $this->urlvalues);
                }
            }
            catch (Exception $e)
            {
                echo 'Caught exception: ',  $e->getMessage(), "\n";
            }
        }
}

?>
```

Now let’s write our base controller class that will do the general work of relaying commands that come from the URL via the interpreter and put it in it’s own utility class file:

```
<?php
//BaseController.php

abstract class BaseController {
        protected $urlvalues;
        protected $action;

        public function __construct($action, $urlvalues) {
                $this->action = $action;
                $this->urlvalues = $urlvalues;
        }

        public function ExecuteAction($input) {
                return $this->{$this->action}($input);
        }

        protected function ReturnView($viewmodel) {
                $viewlocation = 'views/' . get_class($this) . '/' . $this->action . '.php';
                require($viewlocation);
        }
}

?>
```

While we are discussing the utility classes let's go ahead and introduce the last one we need. This one will assist in writing dynamic data to our presentation files by reviewing a "dumb" html template and replacing keys in that file with dynamic data that flows from the model. We’ll name this class Template and also put it in it’s own PHP file:

```
<?php

class Template
{
public $file;
        public $vars=array();

        public function set($key, $value)
        {
          $this->vars[$key] = $value;
        }

        public function display()
        {

           try
           {
               if(file_exists($this->file))
               {
                 $output = file_get_contents($this->file);
                 foreach($this->vars as $key => $value)
                 {
                   $output = preg_replace('/{'.$key.'}/', $value, $output);
                 }
                 echo $output;
               }
               else
               {
                 throw new Exception("Missing template --" . $this->file);
```

```
                }

              }

    catch (Exception $e)
    {
                    echo "Exception caught: " . $e->getMessage();
            }

        }
}

?>
```

Notice that the `display()` method uses the PHP regular expression function `preg_replace()` to replace any text in the HTML view file that is enclosed by curly braces:

```
$output = preg_replace('/{'.$key.'}/', $value, $output);
```

This `$key` and corresponding `$value` are stored in the `vars` array . Key /value pairs are added to the `vars` array via the `set($key, $value)` function. If this doesn't make sense to you yet it will after we look at the markup for the view. Let's write up a simple HTML page that will "show" the menu:

```
<!-View markup to Show the menu data -->

<--
Header
-->
<html>
<head>
</head>
<body>
<img src="images/logo.jpg">
<table width='900'>

<--
This section will print once per sub menu object, if we create a DrinkMenu,
AppetizerMenu, LunchMenu and DessertMenu this section will be printed 4 times to display the title
of the corresponding menu
-->
<tr><td colspan='3'><center><h1><font style='Georgia'><i>{Title}</i></font></h1></center></td></tr>
<tr>
<th width='*' align='left'><b>Menu Item</b></th>
<th width='250' align='left'><b>Serving Size</b></th>
<th width='40' align='left'><b>Price</b></th>
</tr>
<tr><td colspan='3'>&nbsp;</td></tr>

<--
This section will be used in a loop and will print once per menuitem object
-->
<tr>
<td width='*'><b>{itemname}</b></td>
<td width='250'>{servingsize}</td>
<td width='40'>${price}</td>
</tr>
<tr><td colspan='3'><div style='word-wrap: break-word;'>{description}</div></td></tr>
<tr><td colspan='3'>&nbsp;</td></tr>
</table>
<br><br>

<--
Footer
-->
</body>
</html>
```

Take note of the "keys" which are enclosed by the curly braces: `{Title}, {itemname}, {servingsize}, {price}` and `{description}`. The Display function in the Template class will use the preg_replace method to find and replace this pattern of text with an actual correlating value that is gets from the data model which in turn gets the value from the database.

View

Although it is acceptable to include PHP in your View files it is better to avoid it if possible and maintain a clean presentation layer that consist only of pure markup language. The essence of MVC is to keep major aspects of an application completely separated. In order to achieve that with our Restaurant Menu Management application we need to break up the HTML above into separate template files because a couple of sections of this page will be used in different iterative calls, as the comments in the HTML indicate. In this particular view which will “show” the menu, we need to break up this markup into four separate HTML view files as such:

```
<!-
header_template.html
-->
<html>
<head>
</head>
<body>
<img src="images/logo.jpg">
<table width='900'>
<!- ------ -->


<!-
label_template.html
-->
<tr><td colspan='3'><center><h1><font style='Georgia'><i>{Title}</i></font></h1></center></td></tr>
<tr>
<th width='*' align='left'><b>Menu Item</b></th>
<th width='250' align='left'><b>Serving Size</b></th>
<th width='40' align='left'><b>Price</b></th>
</tr>
<tr><td colspan='3'>&nbsp;</td></tr>
<!- ------ -->


<!-
menu_template.html
-->
<tr>
<td width='*'><b>{itemname}</b></td>
<td width='250'>{servingsize}</td>
<td width='40'>${price}</td>
</tr>
<tr><td colspan='3'><div style='word-wrap: break-word;'>{description}</div></td></tr>
<tr><td colspan='3'>&nbsp;</td></tr>
</table>
<br><br>
<!- ------ -->


<!-
footer_template.html
-->
</body>
</html>
<!- ------ -->
```

We will then save these view files in a folder called “show/templates” in the views directory so that our folder structure now looks like this:

```
/htdocs
      /models
      /views
            /show
                  /templates
```

```
        /controllers
        /helperclasses
        /images
        /database
```

The Display Class

Now that our presentation templates are set up we need one more thing to complete our view for the Show feature, a utility class called `Display()`. The Display class will be responsible for assembling the template files and writing dynamic data where required that comes from the corresponding model. The Display class will use objects instantiated from the Template class to write this dynamic data to the static template files. We'll put this class in a file called Display.php and save it in the show/views directory.

```
<?php


$display = new Display();
$display->Header();
$display->Body($viewmodel);
$display->Footer();

class Display
{

        public function Header()
        {
                require_once('views/show/templates/header_template.html');
        }

        public function Body($viewmodel)
        {
            foreach ($viewmodel as $menu)
            {
                $template_header= new Template();
                $template_header->file = "views/show/templates/label_template.html";
                $template_header->set('Title', $menu['title']);

                $template_header->display();

                $template_body = new Template();
                $template_body->file = "views/show/templates/menu_template.html";

                foreach ($menu['data'] as $menuitem)
                {
                        $template_body->set('itemname', strtoupper( $menuitem->itemname) );
                        $template_body->set('description', $menuitem->description);
                        $template_body->set('servingsize', $menuitem->servingsize);
                        $template_body->set('price', $menuitem->price);

                        $template_body->display();
                }
            }
        }

        public function Footer()
        {
                        require_once('views/show/templates/footer_template.html');
        }
}

?>
```

As you can see the Display() class for the Show feature consists of only three methods, `Header()`, `Body()` and `Footer()`. All of the information we need from the data model to present a restaurant menu to the user is in the input variable `$viewmodel`, which to serve our purposes here, is passed to the `Body()` method because that is where all of the dynamic formatting will occur in this particular scenario.

Now we can proceed to link everything together with a model and controller to show the menu. The controllers contain methods that directly correspond to actions given by the user in the URL, hence the action in the URL is actually the name of the method in the controller which is to be invoked. The name of the controller that is called, which is actually a class that contains the action (method), is also in the URL. Controller actions/methods are responsible for creating an instance of the relevant model which in turn facilitates the transfer of data to the corresponding view. The controller and it's related model should be kept in files that have identical names but stored in different directories. In this case we want to present the menu data so we will have a controller file called "show.php" which is kept in the controllers directory and a model file called "show.php" which is kept in the models directory.

Controller

As it's name implies, the Controller controls the communication between the major elements of an MVC application. It serves as a "switchboard operator" that relays requests and responses between the model and the view. In regards to our restaurant menu application, the "Show" controller extends the BaseController class which receives the action from the Interpreter and executes that action with the `ExecuteAction()` function. The "show" controller extends the BaseController class which receives the action from the Interpreter and executes that action with the `ExecuteAction()` function.

Keep in mind that the URL command that will "show" the menu is:

http://localhost/show/Display

The Interpreter evaluates this URL and determines that the `Display()` method of the Show controller class needs to be invoked so it creates an instance of the Show class that's in /htdocs/controllers/show.php and executes the `Display()` method. The `Display()` method does three things. First it creates an instance of the corresponding ShowModel called `$viewmodel`. Next it invokes the Display() method of the `$viewmodel` object. Thirdly it passes the response from the ShowModel to the corresponding View in the view/show/Display.php file via the `ReturnView()` method that's defined in the BaseController class. As you can see above in the Display.php file, the corresponding view for the Show feature is setup by creating an instance of the Display() class in the Display.php file and invoking it's `Header(), Body()` and `Footer()` methods.

Here is our "Show" controller:

```
<?php
/* *** controllers/show.php *** */


class Show extends BaseController {

        public function Display() {

                $viewmodel = new ShowModel();
                $this->ReturnView($viewmodel->Display());

        }
}

?>
```

Now we need to build a model, called ShowModel, that corresponds to the controller above.

Model

As you can see below, the model that corresponds to the "Show" controller is defined in a class called ShowModel in a file named show.php that is in the models directory:

```
<?php
/* *** models/show.php *** */


class ShowModel {

 public function Display() {

  $drinks = new DrinkMenu();
  $appetizers = new AppetizerMenu();
  $lunch = new LunchMenu();
  $dinner = new DinnerMenu();

  $arr_drinks=array("title"=>DrinkMenu::$title,"data"=>$drinks->getAllMenuItems(DrinkMenu::id));
  $arr_appetizers=array("title"=>AppetizerMenu::$title,"data"=>$appetizers->getAllMenuItems(AppetizerMenu::id));
  $arr_lunch=array("title"=>LunchMenu::$title,"data"=>$lunch->getAllMenuItems(LunchMenu::id));
  $arr_dinner=array("title"=>DinnerMenu::$title,"data"=>$dinner->getAllMenuItems(DinnerMenu::id));

  return array($arr_drinks, $arr_appetizers, $arr_lunch, $arr_dinner);

  }

}
?>
```

This is where we finally put our data model to use by creating some sub menu objects which extend the abstract Menu class. Recall that the abstract Menu class has a method called `getAllMenuItems` which creates MenuItem objects and loads them with relevant data from the menuitem table in the database. Also recall that we override this method in the DinnerMenu class.

Note that according to our business design, the LunchMenu and DinnerMenu will use the same MenuItems in our application because we employed interfaces that enforce this business rule. Remember that these interfaces are used for the purpose of adjusting the prices and serving sizes between lunch and dinner menus which both have the same menu items. As a result we override the getAllMenuItems($menuitemid) method of the abstract Menu class in the DinnerMenu class. Had we not designed our application this way we normally would have included the getAllMenuItems() method in the DBMapper class instead of including it in the base Menu class. We did it the way that we did in order to demonstrate the use of interfaces.

For starters, I am just going to add sub menus for drinks, appetizers, lunch and dinner by creating instances of the DrinkMenu, AppetizerMenu, LunchMenu and DinnerMenu classes respectively. Each of these objects will call the `getAllMenuItems()` method to retrieve relevant data which correlates to the `const id` of each object's class. The `$dinner` object calls the `getAllMenuItems()` method that it overrides in order to implement the interfaces that are referenced by the DinnerMenu class. When you look at the output be sure to compare prices and serving sizes between matching menu items on the Lunch and Dinner Menus.

There is one last thing we have to do before all of this will work. We have to add references to the controllers/show.php and models/show.php files to the index.php file. The index.php file is the point of entry for URL requests and is also the assembly file that ties everything together. Our data model and helper files are all referenced here too. It also is where we create an instance of the Interpreter class and pass it the $_GET array that contains the query string values in the URL. The instance of the Interpreter class then creates the appropriate controller object based on the URL input. The last thing the index.php file does is invoke the ExecuteAction method of the controller object which in turn executes the action in the URL:

```
<?php
ini_set('display_errors',1);
error_reporting(E_ALL);


//reference the general classes
require("helperclasses/Interpreter.php");
require("helperclasses/BaseController.php");
require("helperclasses/Template.php");
require_once("helperclasses/Database.php");

//reference the data model
require("models/MenuModel.php");

//reference the model classes
```

```
require("models/show.php");

//reference the controller classes
require("controllers/show.php");

//create the controller and execute the action
$interpreter = new Interpreter($_GET);
$controller = $interpreter->CreateController();

$controller->ExecuteAction();

?>
```

And that's it! Let's take a look at the output:

WILDOCEAN

Fresh Exquisite Seafood & Fine Dining

Drink Menu

Menu Item	Serving Size	Price
CHERRY SODA Refreshing soda blended with candied cherry syrup	16 oz	$1.59
FRESH BREWED COFFEE Fresh Ground, Breshly Brewed Coffee	8 oz	$0.59
ICE WATER Ice Cold Water with Lemon	12 oz	$0.29
JALAPEÑO-LIME SPRITZER A refreshing chilled spritzer with a kick	16 oz	$1.99

Appetizer Menu

Menu Item	Serving Size	Price
OYSTERS ROCKEFELLER 6 oysters baked, topped with Pernod flavored spinach and Parmesan cheese.	6 oysters	$12.55
CHILLED SHUCKED OYSTERS Our fresh live oysters are from the Lousiana Gulf Coast and are the best live oysters to be found	12 oysters	$13.99
GRILLED LOUISIANA OYSTERS 6 oysters on the half shell, topped with Parmesan cheese and butter, char-grilled until plump.	6 oysters	$9.99
OYSTERS BIENVILLE 6 oysters baked, topped with Bay shrimp in a rich white wine cream sauce.	6 oysters	$9.99
PEEL-N- EAT SHRIMP Large Gulf shrimp steeped in beer, fresh lemon and country herbs. Chilled and served with our homemade red sauce.	12 shrimp	$9.99
GULF SHRIMP SKEWER Seven large shrimp skewered and prepared to your liking. Grilled or Blackened	7 shrimp	$8.99
CRAB BISQUE Sherry infused cream and crab stock with lump crab meat. Served with fresh made buttery corn bread.	1 bowl	$7.50

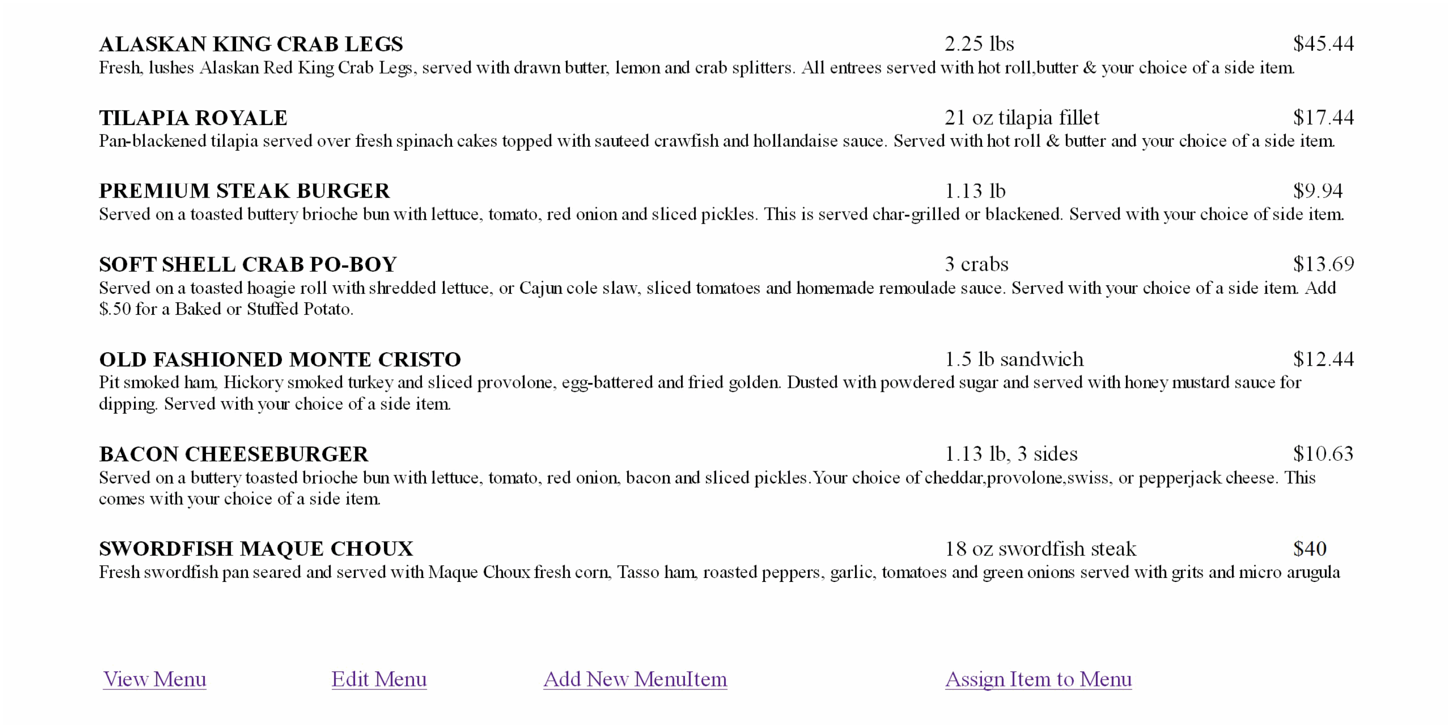
Lunch Menu

Menu Item	Serving Size	Price
GULF AMBERJACK SANDWICH Served seasoned and grilled or pan-blackened on a toasted buttery brioche bun with lettuce, tomato, red onion and sliced pickles. Served with your choice of a side item .	10 oz fillet	$11.95

HAWAIIAN COCONUT SHRIMP	8 shrimp	$14.25
Butterflied jumbo shrimp hand dipped in our sweet coconut milk batter, then rolled in toasted coconut flakes and deep-fried golden. Served with apricot dipping sauce. Served with hot roll, butter, & your choice of a side item.		
NEW YORK STRIP STEAK	12 oz	$19.95
Our hand-cut 12 ounce New York Strip, seasoned just right! Cooked by our grillmaster to your specification. Served with hot roll, butter, & your choice of a side item.		
SOUTHERN STYLE CRABCAKES	2 cakes	$13.95
Three golden brown crabcakes served over a garnish of rice pilaf with homemade remoulade sauce. All entrees served with hot roll, butter, & your choice of a side item.		
OYSTERS BENEDICT	8 oysters	$10.95
Fried gulf oysters, sliced tomatoes and poached eggs on toasted English muffins, topped with creamy hollandaise sauce. All Benedicts served with your choice of home-style potatoes, buttery grits or fresh fruit.		
TWIN JUMBO SOFT SHELL CRAB	2 crabs	$20.95
Two soft shell crabs dipped in our Cajun breading and fried golden. Served over Creole shrimp stuffing & topped with basil buerre blanc. This comes with a hot roll, butter & your choice of a side item.		
ALASKAN KING CRAB LEGS	1.5 lbs	$36.35
Fresh, lushes Alaskan Red King Crab Legs, served with drawn butter, lemon and crab splitters. All entrees served with hot roll,butter & your choice of a side item.		
TILAPIA ROYALE	14 oz tilapia fillet	$13.95
Pan-blackened tilapia served over fresh spinach cakes topped with sauteed crawfish and hollandaise sauce. Served with hot roll & butter and your choice of a side item.		
PREMIUM STEAK BURGER	0.75 lb	$7.95
Served on a toasted buttery brioche bun with lettuce, tomato, red onion and sliced pickles. This is served char-grilled or blackened. Served with your choice of side item.		
SOFT SHELL CRAB PO-BOY	2 crabs	$10.95
Served on a toasted hoagie roll with shredded lettuce, or Cajun cole slaw, sliced tomatoes and homemade remoulade sauce. Served with your choice of a side item. Add $.50 for a Baked or Stuffed Potato.		
OLD FASHIONED MONTE CRISTO	1 lb sandwich	$9.95
Pit smoked ham, Hickory smoked turkey and sliced provolone, egg-battered and fried golden. Dusted with powdered sugar and served with honey mustard sauce for dipping. Served with your choice of a side item.		
BACON CHEESEBURGER	.75 lb, 2 sides	$8.50
Served on a buttery toasted brioche bun with lettuce, tomato, red onion, bacon and sliced pickles.Your choice of cheddar,provolone,swiss, or pepperjack cheese. This comes with your choice of a side item.		
SWORDFISH MAQUE CHOUX	12 oz swordfish steak	$32
Fresh swordfish pan seared and served with Maque Choux fresh corn, Tasso ham, roasted peppers, garlic, tomatoes and green onions served with grits and micro arugula		

Dinner Menu

Menu Item	**Serving Size**	**Price**
GULF AMBERJACK SANDWICH	15 oz fillet	$14.94
Served seasoned and grilled or pan-blackened on a toasted buttery brioche bun with lettuce, tomato, red onion and sliced pickles. Served with your choice of a side item .		
HAWAIIAN COCONUT SHRIMP	12 shrimp	$17.81
Butterflied jumbo shrimp hand dipped in our sweet coconut milk batter, then rolled in toasted coconut flakes and deep-fried golden. Served with apricot dipping sauce. Served with hot roll, butter, & your choice of a side item.		
NEW YORK STRIP STEAK	18 oz	$24.94
Our hand-cut 12 ounce New York Strip, seasoned just right! Cooked by our grillmaster to your specification. Served with hot roll, butter, & your choice of a side item.		
SOUTHERN STYLE CRABCAKES	3 cakes	$17.44
Three golden brown crabcakes served over a garnish of rice pilaf with homemade remoulade sauce. All entrees served with hot roll, butter, & your choice of a side item.		
OYSTERS BENEDICT	12 oysters	$13.69
Fried gulf oysters, sliced tomatoes and poached eggs on toasted English muffins, topped with creamy hollandaise sauce. All Benedicts served with your choice of home-style potatoes, buttery grits or fresh fruit.		
TWIN JUMBO SOFT SHELL CRAB	3 crabs	$26.19
Two soft shell crabs dipped in our Cajun breading and fried golden. Served over Creole shrimp stuffing & topped with basil buerre blanc. This comes with a hot roll,		



ALASKAN KING CRAB LEGS 2.25 lbs $45.44
Fresh, lushes Alaskan Red King Crab Legs, served with drawn butter, lemon and crab splitters. All entrees served with hot roll,butter & your choice of a side item.

TILAPIA ROYALE 21 oz tilapia fillet $17.44
Pan-blackened tilapia served over fresh spinach cakes topped with sauteed crawfish and hollandaise sauce. Served with hot roll & butter and your choice of a side item.

PREMIUM STEAK BURGER 1.13 lb $9.94
Served on a toasted buttery brioche bun with lettuce, tomato, red onion and sliced pickles. This is served char-grilled or blackened. Served with your choice of side item.

SOFT SHELL CRAB PO-BOY 3 crabs $13.69
Served on a toasted hoagie roll with shredded lettuce, or Cajun cole slaw, sliced tomatoes and homemade remoulade sauce. Served with your choice of a side item. Add $.50 for a Baked or Stuffed Potato.

OLD FASHIONED MONTE CRISTO 1.5 lb sandwich $12.44
Pit smoked ham, Hickory smoked turkey and sliced provolone, egg-battered and fried golden. Dusted with powdered sugar and served with honey mustard sauce for dipping. Served with your choice of a side item.

BACON CHEESEBURGER 1.13 lb, 3 sides $10.63
Served on a buttery toasted brioche bun with lettuce, tomato, red onion, bacon and sliced pickles.Your choice of cheddar,provolone,swiss, or pepperjack cheese. This comes with your choice of a side item.

SWORDFISH MAQUE CHOUX 18 oz swordfish steak $40
Fresh swordfish pan seared and served with Maque Choux fresh corn, Tasso ham, roasted peppers, garlic, tomatoes and green onions served with grits and micro arugula

View Menu   Edit Menu   Add New MenuItem   Assign Item to Menu

Figure 8-1. Browser *output from http://localhost/show/Display*

Develop the Functionality to Add New Menu Items

So now if we want to add additional capability to the restaurant menu management application we just follow the same pattern. Our application doesn't have add or edit capability yet nor a way to assign menu items to menus. Recall that we already developed a `Save()` method in the DBMapper class of the data model. The `Save()` method receives the $_POST array as input and has the ability to determine if the input it receives needs to be inserted into the database if it doesn't exist or updated in the database if it does. It also determines which database table it needs to work with. Our database is very simple, the schema only includes two tables:

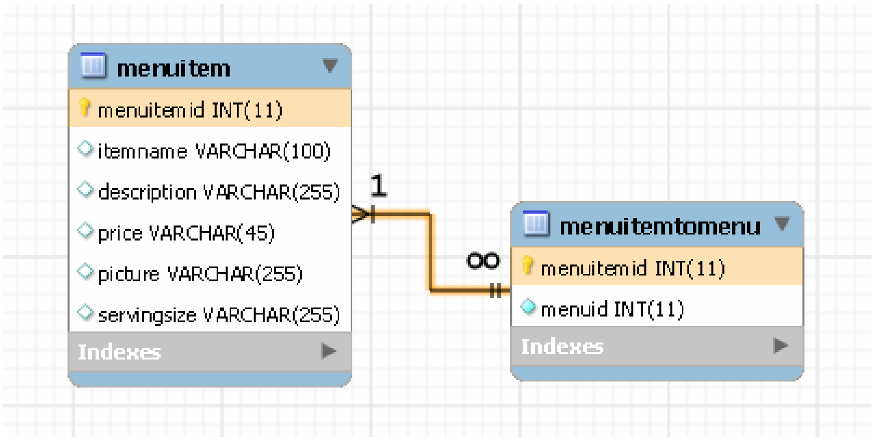


Figure 8-2. *Restaurant Database Entity Relationship Diagram*

In order to develop the capability to add new menu items to the database, we need to be aware of the data that we want to capture from the user. In bigger applications that adhere to domain driven design this is usually determined from the data model. For our purposes here we'll just refer to the database schema. Keeping this in mind, let's type up an HTML document that includes a form that the user can use to add a new menu item to the menuitem table.

```
<html>
<head>
<title>Add New Menu Item</title>
</head>
<body>
<form method="POST" action="index.php/add/Save">
<table>
<tr><td>Item Name:</td><td><input type="text" name="itemname"></td></tr>
<tr><td>Description:</td><td><input type="text" name="description"></td></tr>
<tr><td>Serving Size:</td><td><input type="text" name="servingsize"></td></tr>
<tr><td>Picture:</td><td><input type="text" name="picture"></td></tr>
<tr><td>Price:</td><td><input type="text" name="price"></td></tr>
<tr><td><input type="submit" name="submit" value="submit"></td></tr>
</table>
</form>
</body>
</html>
```

Observe the tag line that opens the form:

```
<form method="POST" action="index.php/add/Save">
```

We intend to post the user input data from this form to the index.php file which in turn will make the $_POST array available to the appropriate controller. Based on the action in the form tag line we will be calling this new controller "Add" and it will invoke a method called `Save()`. This leads into what we need to do next, create the new controller. We'll first need to extend the BaseController class so that we have access to it's methods. Just like we did for the Show controller, we'll create a method called `Display()` that serves as the relay between the model and the view. The `Save()` method of this controller will take the $_POST array that contains the user inputs and pass them to the `Save()` method in the DBMapper class of MenuModel.php which in turn makes sure the input gets written to the database. The `Save()` redirects the user to the edit page which we have not developed yet. Let's name the file that contains this controller add.php and save it off into the controllers directory.

Controller for the Add Menu Item Feature

```
<?php
// controller/add.php

class Add extends BaseController {

        protected function Display() {

                $viewmodel = new AddModel();
                $this->ReturnView($viewmodel->Display());
        }

        protected function Save()
        {
                $saveitem = new DBMapper();
                $saveitem->Save($_POST);
                header('Location: index.php/edit/Display');
        }
}

?>
```

Our model for this feature will be simple since we aren't sending any data to the view because our "Add Menu Item" form doesn't need anything from the data model. Keeping with our naming convention let's create an AddModel class. To stay consistent with the way we did things for the Show feature, let's give the AddModel class a `Display()` method which returns nothing because the View requires nothing from the model in this scenario:

Model for the Add Menu Item Feature

```
<?php
// models/add.php
```

```
class AddModel {

        public function Display()
        {
               return;
        }
}

?>
```

View for the Add MenuItem Feature

To be consistent with the way we split the view between a Header, Body and Footer, let's split up the Add New Menu Item HTML document into three separate files accordingly. Let's also add some navigation links in the footer that we can use to choose which view we want to see in the browser. Notice that new `add` controller and `ShowAddForm()` action/method is used in the navigation link that brings up the new Add Menu Item page. We should also go ahead and copy/paste these navigation links into the views/show/templates/footer_template.html file.

View Templates for the Add MenuItem Feature

```
<!–
header_addmenuitem.html
-->
<html>
<head>
<title>Add New Menu Item</title>
</head>
<!– ------ -->



<!–
body_addmenuitem.html
-->
<body>
<form method="POST" action="index.php/add/Save">
<table>
<tr><td>Item Name:</td><td><input type="text" name="itemname"></td></tr>
<tr><td>Description:</td><td><input type="text" name="description"></td></tr>
<tr><td>Serving Size:</td><td><input type="text" name="servingsize"></td></tr>
<tr><td>Picture:</td><td><input type="text" name="picture"></td></tr>
<tr><td>Price:</td><td><input type="text" name="price"></td></tr>
<tr><td><input type="submit" name="submit" value="submit"></td></tr>
</table>
</form>
<!– ------ -->



<!–
footer_addmenuitem.html
-->
</ br>
</ br>
<center>
<table width="100%">
<tr>
<td><a href="index.php/show/ShowMenu">View Menu</a></td>
<td><a href="index.php/add/ShowAddForm">Add New MenuItem</a></td>
</tr>
</table>
</center>
</body>
</html>
<!– ------ -->
```

Now let's create the folder path "add/templates" in the views directory and save these three files in it.

Display Manager for the Add MenuItem Feature

Just like we did for the Show Menu feature, we will build a `Display()` class that will manage the assembly of the view for the Add Menu Item feature. This `Display()` class will reside in a file called Display.php in the add/views folder. As was mentioned above, the view for this Add Menu Item feature doesn't need anything from the data model to build an input form so this Display class will be simple.

So let's create a class called `Display()` and give it three simple methods to format our view: `Header()`, `Body()` and `Footer()`. We'll put this class in a PHP file called Display.php, create an instance of this class, invoke it's methods and save it to the views/add folder so when this file is referenced by the controller it will assemble the view:

```
<?php
// views/add/Display.php

$display = new Display();
$display->Header();
$display->Body();
$display->Footer();


class Display
{
        public function Header()
        {
                require_once('views/add/templates/header_addmenuitem.html');
        }

        public function Body()
        {

                require_once('views/add/templates/body_addmenuitem.html');
        }

        public function Footer()
        {
                require_once('views/add/templates/footer_addmenuitem.html');
        }
}

?>
```

Now all that's left to do is to add references to our new model and controller to the index.php file:

```
<?php
ini_set('display_errors',1);
error_reporting(E_ALL);


//require the general classes
require("helperclasses/Interpreter.php");
require("helperclasses/BaseController.php");
require("helperclasses/Template.php");
require_once("helperclasses/Database.php");

//require the data model
require("models/MenuModel.php");

//require the model classes
require("models/show.php");
require("models/add.php");

//require the controller classes
require("controllers/show.php");
require("controllers/add.php");

//create the controller and execute the action
$interpreter = new Interpreter($_GET);
$controller = $interpreter->CreateController();

$controller->ExecuteAction();

?>
```

Now we can add new menu items to the database from the user interface. We'll also need a way to assign new menu items to one or more available menus so let's write out the markup for a page that will allow the user to do this:

```
<html>
<head>
<title>Assign Menu Items to Menus</title>
</head>
<body>
<form method="POST" action="index.php/assign/Save">
<table>
<tr><td>Select a Single Menu Item:</td></tr>
<tr>
<td>
<select name="menuitemid">

<option value="{menuitemid}">{itemname}</option>

</select>
</td>
</tr>
<tr><td>&nbsp;</td></tr>
<tr><td>Choose the Menus you want to add the Menu Item to</td></tr>
<tr><td>
<select name="menuid" multiple="multiple">

<option value="{menuid}">{menuname}</option>

</select>
</td>
</tr>
</table>
<br>
<br>
<input type="submit" value="UPDATE">
</form>
<br>
<br>
<br>
<br>
<br>
<br>
<center>
<table width="100%">
<tr>
<td><a href="index.php/show/ShowMenu">View Menu</a></td>
<td><a href="index.php/add/ShowAddForm">Add New MenuItem</a></td>
<td><a href="index.php/assign/ShowAssignForm">Assign Item to Menu</a></td>
</tr>
</table>
</center>
</body>
</html>
</body>
</html>
```

What we have done in the html document above is create a select list that we will want to load all available menu items into and another select list that will contain all of the available menus. The value in the first select list will be the menuitemid of the menuitem that is selected by the user. We know that since we are loading a select list with dynamic data that a loop is going to be involved with both select lists so let's segregate the segments of this page that will be used iteratively and the header, footer and other static segments of this HTML page. The resulting view will be split between five template files, we will save them in the templates folder in the views/assign directory.

View Templates for the Assign MenuItem-to-Menu Feature

```
<!- header_assignmenuitem.html -->
<html>
<head>
<title>Assign Menu Items to Menus</title>
```

```
</head>
<body>
<form method="POST" action="index.php/assign/Save">
<table>
<tr><td>Select a Single Menu Item:</td></tr>
<tr>
<td>
<select name="menuitemid">
<!- ----------------- -->


<!- body_assignmenuitem1.html -->
<option value="{menuitemid}">{itemname}</option>
<!- ----------------- -->


<!- body_assignmenuitem2.html -->
</select>
</td>
</tr>
<tr><td>&nbsp;</td></tr>
<tr><td>Choose the Menus you want to add the Menu Item to</td></tr>
<tr><td>
<select name="menuid" multiple="multiple">
<!- ----------------- -->


<!- body_assignmenuitem3.html -->
<option value="{menuid}">{menuname}</option>
<!- ----------------- -->


<!- footer_assignmenuitem1.html -->
</select>
</td>
</tr>
</table>
<br>
<br>
<input type="submit" value="UPDATE">
</form>
<br>
<br>
<br>
<br>
<br>
<br>
<center>
<table width="100%">
<tr>
<td><a href="index.php/show/Display">View Menu</a></td>
<td><a href="index.php/add/Display">Add New MenuItem</a></td>
<td><a href="index.php/assign/Display">Assign Item to Menu</a></td>
</tr>
</table>
</center>
</body>
</html>
</body>
</html>
<!- ----------------- -->
```

Notice that the footer now includes a navigation link to the Assign Menuitem to Menu page. Let's proceed with writing out a controller for the assign feature and put it in a file called assign.php and save it to the controllers directory

Controller for the Assign MenuItem-to-Menu Feature

```
<?php
// controllers/assign.php

class Assign extends BaseController {

       protected function Display() {
```

```
                $viewmodel = new AssignModel();
                $this->ReturnView($viewmodel->Display());
        }

        protected function Save()
        {
                $assignitem = new DBMapper();
                $assignitem->Save($_POST);
                header('Location: index.php?controller=show&action=Display');
        }
}

?>
```

Now do to the nature of our design, we do not have a menu table in the database. Instead we create our sub menus in our data model by creating sub menu classes that extends the abstract menu class (i.e.: LunchMenu, DinnerMenu, KidMenu, etc.). Hence, we must manually add our sub menus and their IDs to an array in the `Display()` method that will be passed on to the display and load the select list that lists the menus. The `ShowAssignForm()` gets the menuitems from the database via the data model. The Save() method is basically the same as it is in the “add” controller, we just redirect the view back to the Edit form which we are about to create after we develop the AssignModel and add the Assign controller and model references to index.php.

So let’s proceed with developing the AssignModel. Just like with the ShowModel and AddModel, the AssignModel class will include a `Display()` method. This `Display()` method for the Assign MenuItem-to-Menu Feature will need to fetch the menuitem table from the database in addition to having an array to correspond to the available sub menu classes we created in the data model. The view for this feature will need this information to populate the select lists on the user interface.

Model for the Assign MenuItem-to-Menu Feature

```
<?php

class AssignModel {

        public function Display()
        {
                $loadtable = new DBMapper();
                $menuitems=$loadtable->getMenuItemTable();
                $arr_menuitems = array("menuitem"=>$menuitems);

                $menus = array("2"=>"Drinks","3"=>"Lunch & Dinner","5"=>"Kids","6"=>"Dessert","7"=>"Appetizers");
                $arr_menus = array("menu"=>$menus);

                return array($arr_menuitems, $arr_menus);
        }

}


?>
```

The Display() method of the AssignModel class uses the getMenuItemTable() method of the DBMapper class in MenuModel.php to grab the entire menuitem table. Both the data from the menuitem table and the array that has the menu names and IDs are each packaged in different arrays which are in turn wrapped in an array themselves which will be shipped to the view as the return value of this `Display()` method. The controller passes this return value to the view in a variable called `$viewmodel`.

Display Manager for the Assign MenuItem-to-Menu View

The corresponding Display() method in the view's Display class will un-wrap these nested arrays from the AssignModel with nested `foreach` loops so that this data can be presented according to the format set up by the markup language of the corresponding template files. Just like with the Show and Add features, we will instantiate the Display() class in the views/assign/Display.php file and the `Body()` method will receive this packaged data from the AssignModel as input and do the un-wrapping:

```
<?php

$display = new Display();
$display->Header();
$display->Body($viewmodel);
$display->Footer();


class Display{

        public function Header()
        {
                require_once('views/assign/templates/header_assignmenuitem.html');
        }

        public function Body($viewmodel)
        {

                $template_body1 = new Template();
                $template_body1->file = "views/assign/templates/body_assignmenuitem1.html";

                foreach ($viewmodel as $selectlist)
                {
                        foreach ($selectlist['menuitem'] as $menuitem)
                        {
                           $template_body1->set('menuitemid', $menuitem->menuitemid);
                           $template_body1->set('itemname', $menuitem->itemname);
                           $template_body1->display();
                        }
                }


                require('views/assign/templates/body_assignmenuitem2.html');


                $template_body3 = new Template();
                $template_body3->file = "views/assign/templates/body_assignmenuitem3.html";

                foreach ($viewmodel as $selectlist)
                {
                        foreach ($selectlist['menu'] as $menuid => $menuname)
                        {
                           $template_body3->set('menuname', $menuname);
                           $template_body3->set('menuid', $menuid);
                           $template_body3->display();
                        }

                }
        }

        public function Footer()
        {
                require_once('views/assign/templates/footer_assignmenuitem.html');
        }
}



?>
```

We employed objects instantiated from the Template class twice in the Body() method because we have two select lists which will need to be dynamically loaded. All together the Header(), Body() and Footer() assemble a complete view that incorporates relevant information from the database. Now we just need to tie it into the application via the index.php file by requiring the files that contain the model and controller:

```
<?php
ini_set('display_errors',1);
error_reporting(E_ALL);


//require the general classes
require("helperclasses/Interpreter.php");
require("helperclasses/BaseController.php");
require("helperclasses/Template.php");
require_once("helperclasses/Database.php");

//require the data model
require("models/MenuModel.php");

//require the model classes
require("models/show.php");
require("models/add.php");
require("models/assign.php");

//require the controller classes
require("controllers/show.php");
require("controllers/add.php");
require("controllers/assign.php");


//create the controller and execute the action
$interpreter = new Interpreter($_GET);
$controller = $interpreter->CreateController();

$controller->ExecuteAction();

?>
```

Lastly we will develop the capability to edit and delete menu items. Again, referring back to the DBMapper class in the ModelMenu.php file, you can see that we have already developed an `Erase()` method and the `Save()` method that has the capability to update existing records in the database. It follows that our EditModel class of our "Edit" controller will need methods that use these DBMapper methods along with a Display() method that relays communication between the model and view.

This time, just to demonstrate that you don't have to start with the View when developing new capability in an application, we'll build our controller first based on the assumptions we just made about what it will need to do. Beginning with extending the BaseController class, let's create the methods `Display()`, `Save()` and `Erase()`. We'll make the assumption that the user will want to be immediately re-directed back to the Edit page to verify that the edit/delete action worked so let's put those re-direct commands in both the `Save()` and `Erase()` methods.

Controller for the Edit/Delete MenuItem Feature

```
<?php
// controllers/edit.php

class Edit extends BaseController {

        public function Display()
        {
                $viewmodel = new EditModel();
                $this->ReturnView($viewmodel->Display());
        }

        public function Save()
        {
                $updateitem = new DBMapper();
                $updateitem->Save($_POST);
                header('Location: index.php/edit/Display');
        }

        public function Erase()
        {
                $eraseitem = new DBMapper();
                $eraseitem->Erase($_GET['id']); //menuitemid passed through the URL
                header('Location: index.php/edit/Display');
        }

}

?>
```

Now we need a model for the Edit feature. We know that we will want to the ability to edit all of the properties of a menuitem and that we will want to see what the current values are for those properties for all available menuitems in the database. Hence we will need to pull in the menuitem table again as we did with the Assign MenuItem-to-Menu feature so let's to our Edit model.

Model for the Edit MenuItem Feature

```
<?php

class EditModel {

        public function Display()
        {

            $loadtable = new DBMapper();
            $data=$loadtable->getMenuItemTable();

            return($data);
        }

}


?>
```

The EditModel will return the array that it receives from the `getMenuItemTable()` method of the DBMapper class and the Edit controller will pass it along to the Edit view where it will presumably be received by the Edit view's Display management object. Before we build our Display() class for our view, let's write out the markup for this view so we can better understand how we will need to set up our formatting in the Display() class:

General markup for the Edit MenuItem Page:

```
<html>
<head>
<title>Edit Menu Items</title>
</head>
<body>
<table>
<tr>
<th>ID</th>
<th>Item Name</th>
<th>Description</th>
<th>Price</th>
<th>Serving Size</th>
<th>&nbsp;</th>
<th>&nbsp;</th>
<form method="POST" action="index.php/edit/Save">
<tr>
<td><input type="hidden" name="menuitemid" value="{menuitemid}">{menuitemid}</td>
<td><textarea cols="35" rows="1" name="itemname">{itemname}</textarea></td>
<td><textarea cols="55" rows="1" name="description">{description}</textarea></td>
<td><input type="text" name="price" value="{price}"></td>
<td><input type="text" name="servingsize" value="{servingsize}"></td>
<td><input type="submit" value="UPDATE"></td>
<td><a href="index.php/edit/Erase/{menuitemid}">DELETE</a></td>
</tr>
</form>
</table>
</ br>
</ br>
<center>
<table width="100%">
<tr>
<td><a href="index.php/show/Display">View Menu</a></td>
<td><a href="index.php/edit/Display">Edit Menu</a></td>
<td><a href="index.php/add/Display">Add New MenuItem</a></td>
<td><a href="index.php/assign/Display">Assign Item to Menu</a></td>
</tr>
</table>
</center>
</body>
</html>
</body>
</html>
```

Notice that there is a navigation link to the Edit Menu page. This footer section should be copy and pasted to the footer sections in the other views as well to keep things consistent. In the HTML above we have our keys in place and we set up controls for UPDATE and DELETE. The UPDATE feature will POST user changes to index.php which will in turn make the $_POST data available to the Edit controller. The DELETE feature calls the index.php file and tells it that it wants the Edit" controller to invoke the `Erase()` method on the menuitemid that it passes to it through the URL.

View Templates for the Edit MenuItem Feature

Let's break up the HTML for the Edit view into three template files that will be saved to views/edit/templates:

```
<!-- HEADER TEMPLATE -->
<!-- header_editmenuitem.html -->
<html>
<head>
<title>Edit Menu Items</title>
</head>
<body>
<table>
<tr>
<th>ID</th>
<th>Item Name</th>
<th>Description</th>
<th>Price</th>
<th>Serving Size</th>
<th>&nbsp;</th>
<th>&nbsp;</th>
</tr>
<!-- HEADER TEMPLATE -->


<!-- BODY TEMPLATE -->
<!-- body_editmenuitem.html -->
<form method="POST" action="index.php/edit/Save">
<tr>
<td><input type="hidden" name="menuitemid" value="{menuitemid}">{menuitemid}</td>
<td><textarea cols="35" rows="1" name="itemname">{itemname}</textarea></td>
<td><textarea cols="55" rows="1" name="description">{description}</textarea></td>
<td><input type="text" name="price" value="{price}"></td>
<td><input type="text" name="servingsize" value="{servingsize}"></td>
<td><input type="submit" value="UPDATE"></td>
<td><a href="index.php/edit/Erase/{menuitemid}">DELETE</a></td>
</tr>
</form>
<!-- BODY TEMPLATE -->


<!-- FOOTER TEMPLATE -->
<!-- footer_editmenuitem.html -->
</table>
</ br>
</ br>
<center>
<table width="100%">
<tr>
<td><a href="index.php/show/Display">View Menu</a></td>
```

```
<td><a href="index.php/edit/Display">Edit Menu</a></td>
<td><a href="index.php/add/Display">Add New MenuItem</a></td>
<td><a href="index.php/assign/Display">Assign Item to Menu</a></td>
</tr>
</table>
</center>
</body>
</html>
</body>
</html>
<!-- FOOTER TEMPLATE -->
```

Now we can proceed with the Display manager for the Edit feature:

Display Manager for the Edit View

```
<?php


$display = new Display();
$display->Header();
$display->Body($viewmodel);
$display->Footer();


class Display {

        public function Header()
        {
                require_once('views/edit/templates/header_editmenuitem.html');
        }

        public function Body($viewmodel)
        {

                $template_body = new Template();
                $template_body->file = "views/edit/templates/body_editmenuitem.html";

                foreach ($viewmodel as $menuitem)
                {
                        $template_body->set('menuitemid', $menuitem->menuitemid);
                        $template_body->set('itemname', $menuitem->itemname);
                        $template_body->set('description', $menuitem->description);
                        $template_body->set('servingsize', $menuitem->servingsize);
                        $template_body->set('price', $menuitem->price);

                        $template_body->display();
                }
        }

        public function Footer()
        {
                require_once('views/edit/templates/footer_editmenuitem.html');
        }
}


?>
```

Now, we just need to update the index.php file with references to the controllers/edit.php and models/edit.php files and we are in business:

```
<?php
ini_set('display_errors',1);
error_reporting(E_ALL);


//require the general classes
require("helperclasses/Interpreter.php");
require("helperclasses/BaseController.php");
require("helperclasses/Template.php");
```

```
require_once("helperclasses/Database.php");

//require the data model
require("models/MenuModel.php");

//require the model classes
require("models/show.php");
require("models/add.php");
require("models/assign.php");
require("models/edit.php");

//require the controller classes
require("controllers/show.php");
require("controllers/add.php");
require("controllers/assign.php");
require("controllers/edit.php");

//create the controller and execute the action
$interpreter = new Interpreter($_GET);
$controller = $interpreter->CreateController();

$controller->ExecuteAction();

?>
```

The complete code base for the restaurant menu management application that we just developed, and database files are available for download at:

http://www.php-this.com/downloads/restmenuapp_mvc

`Verification Code:` **`ST32919`**

In the code base that you can download I also included Integrate_Display.php files in the view folders which bypass the use of templates altogether and incorporate the view in a single file that integrates PHP with the markup language. The clean method of using templates is the proper way, or the purest way to adhere to MVC design but sometimes it may be necessary to use the integrated view approach.

I hope I have helped you to understand how extensive this MVC application is. You can build additional capability to manage employees and sales tickets, inventory, etc. The point is that with this framework you can easily add to the database schema and data model to build out an entire restaurant management system using this design pattern without intruding on the work we have done here.

Chapter 8
The Reflection API

PHP Reflection API is comprised of a set of methods that allow you to analyze the inner structure of classes and interfaces very easily. It makes it simple to do such things as determine if a reflected class defines a given method or not, or if the class declares a specified property, in addition to checking whether that property is public, protected or private.

```
// create instance of 'DinnerMenu' class
$dinner = new DinnerMenu();


// create instance of Reflection class and pass in 'User' class as argument
$reflect_dinner = new ReflectionClass('DinnerMenu');


// get name of reflected class
echo $reflect_dinner->getName();
```

OUTPUT:

```
DinnerMenu
```

As you can see above, I first created an instance of the DinnerMenu class only for illustrative purposes, and second, a new reflector object has been engendered from the native "ReflectionClass" class. This one is the backbone of the entire reflection API. It takes as an as input argument the name of the class that needs to be reflected.

In this example, the string "DinnerMenu" has been passed to the constructor of the "ReflectionClass" class. This process returns a reflector object that will allow you to retrieve valuable information about the "DinnerMenu" class.

Since I'm just starting to explore the reflection API, the first thing that the above example does is invoke a method called "getName()," which returns the name of the reflected class. Using the getName() method of the Reflection Class may seem pointless at first since we already new the name of the reflected class but there may be situations where the name of the reflected class will not be known in advance. In this example we used the getName() method as an introduction to the methods of the Reflection Class. Let's look at other methods that will allow us to get additional information about that class. The ones that will be discussed in the following section will return the name of the constant defined within the "DinnerMenu" class.

Now let's look at how to use a couple of additional methods to get the names and values of the constants declared by the reflected class.

```
// create instance of 'DinnerMenu' class
$dinner = new DinnerMenu();


// create instance of Reflection class and pass in 'User' class as argument
$reflect_dinner = new ReflectionClass('DinnerMenu');


// get constant in reflected class
echo $reflect_dinner->getConstant('id');
```

OUTPUT:

```
3

// get an array of constants in reflected class
print_r($reflect_dinner->getConstants());
```

OUTPUT:

```
Array ( [id] => 3 )
```

It is apparent in the example above that retrieving the values assigned to the constants defined by a given class is only a matter of calling the "getConstant()" and "getConstants()" methods of the reflection API, and nothing else. The first of these methods takes the name of the constant that needs to be parsed, while the second one simply returns an array of constants, populated with their corresponding values.

```
// create instance of 'DinnerMenu' class
$dinner = new DinnerMenu();


// create instance of Reflection class and pass in 'User' class as argument
$reflect_dinner = new ReflectionClass('DinnerMenu');
```

```
Reflection::export($reflect_dinner);
```

OUTPUT:

```
Class [ <user> final class DinnerMenu extends LunchMenu implements AdjustPortion, AdjustPrice ] {
  @@ C:\temp\restaurant.php 222-269

  - Constants [1] {
    Constant [ integer id ] { 3 }
  }

  - Static properties [1] {
    Property [ public static $title ]
  }

  - Static methods [0] {
  }

  - Properties [0] {
  }

  - Methods [12] {
    Method [ <user, prototype AdjustPortion> public method setDinnerPortion ] {
      @@ C:\temp\restaurant.php 227 - 235

      - Parameters [1] {
        Parameter #0 [ <required> $menuitemServingSize ]
      }
    }

    Method [ <user, prototype AdjustPrice> public method setDinnerPrices ] {
      @@ C:\temp\restaurant.php 237 - 245

      - Parameters [1] {
        Parameter #0 [ <required> $menuitemPrice ]
      }
    }

    Method [ <user, overwrites Menu, prototype Menu> public method GetAllMenuItems ] {
      @@ C:\temp\restaurant.php 248 - 265

      - Parameters [1] {
        Parameter #0 [ <required> $menuid ]
      }
    }

    Method [ <user, prototype AdjustPrice> public method setHappyHourDrinkPrices ] {
      @@ C:\temp\restaurant.php 267 - 267

      - Parameters [1] {
        Parameter #0 [ <required> $menuitemObject ]
      }
    }

    Method [ <user, inherits Menu> public method setMenuID ] {
      @@ C:\temp\restaurant.php 124 - 124

      - Parameters [1] {
        Parameter #0 [ <required> $menuid ]
      }
    }

    Method [ <user, inherits Menu> public method getMenuID ] {
      @@ C:\temp\restaurant.php 125 - 125
    }

    Method [ <user, inherits Menu> public method setMenuItemID ] {
      @@ C:\temp\restaurant.php 127 - 127

      - Parameters [1] {
        Parameter #0 [ <required> $menuitemid ]
      }
    }

    Method [ <user, inherits Menu> public method getMenuItemID ] {
      @@ C:\temp\restaurant.php 128 - 128
    }
    Method [ <user, inherits Menu> public method setMenuName ] {
      @@ C:\temp\restaurant.php 130 - 130

      - Parameters [1] {
        Parameter #0 [ <required> $menuname ]
      }
    }

    Method [ <user, inherits Menu> public method getMenuName ] {
      @@ C:\temp\restaurant.php 131 - 131
    }

    Method [ <user, inherits Menu> public method setDescription ] {
      @@ C:\temp\restaurant.php 133 - 133

      - Parameters [1] {
        Parameter #0 [ <required> $description ]
      }
    }

    Method [ <user, inherits Menu> public method getDescription ] {
      @@ C:\temp\restaurant.php 134 - 134
    }
  }
}
```

Now let's try the same procedure on the ReflectionClass itself so we can gain full insight into the rest of the methods that the PHP5 Reflection API offers:

```
$see_inside_reflection_class = new ReflectionClass(ReflectionClass);
Reflection::export($see_inside_reflection_class);
```

```
OUTPUT:
Class [ <internal:Reflection> class ReflectionClass implements Reflector ] {

  - Constants [3] {
    Constant [ integer IS_IMPLICIT_ABSTRACT ] { 16 }
    Constant [ integer IS_EXPLICIT_ABSTRACT ] { 32 }
    Constant [ integer IS_FINAL ] { 64 }
  }

  - Static properties [0] {
  }

  - Static methods [1] {
    Method [ <internal:Reflection> static public method export ] {

      - Parameters [2] {
        Parameter #0 [ <required> $argument ]
        Parameter #1 [ <optional> $return ]
      }
    }
  }

  - Properties [1] {
    Property [ <default> public $name ]
  }

  - Methods [40] {
    Method [ <internal:Reflection> final private method __clone ] {
    }

    Method [ <internal:Reflection, ctor> public method __construct ] {

      - Parameters [1] {
        Parameter #0 [ <required> $argument ]
      }
    }

    Method [ <internal:Reflection> public method __toString ] {
    }

    Method [ <internal:Reflection> public method getName ] {
    }
```

```
Method [ <internal:Reflection> public method isInternal ] {
}

Method [ <internal:Reflection> public method isUserDefined ] {
}

Method [ <internal:Reflection> public method isInstantiable ] {
}

Method [ <internal:Reflection> public method getFileName ] {
}

Method [ <internal:Reflection> public method getStartLine ] {
}

Method [ <internal:Reflection> public method getEndLine ] {
}

Method [ <internal:Reflection> public method getDocComment ] {
}

Method [ <internal:Reflection> public method getConstructor ] {
}

Method [ <internal:Reflection> public method hasMethod ] {

  - Parameters [1] {
    Parameter #0 [ <required> $name ]
  }
}

Method [ <internal:Reflection> public method getMethod ] {

  - Parameters [1] {
    Parameter #0 [ <required> $name ]
  }
}

Method [ <internal:Reflection> public method getMethods ] {

  - Parameters [1] {
    Parameter #0 [ <optional> $filter ]
  }
}

Method [ <internal:Reflection> public method hasProperty ] {

  - Parameters [1] {
    Parameter #0 [ <required> $name ]
  }
}

Method [ <internal:Reflection> public method getProperty ] {

  - Parameters [1] {
    Parameter #0 [ <required> $name ]
  }
}

Method [ <internal:Reflection> public method getProperties ] {

  - Parameters [1] {
    Parameter #0 [ <optional> $filter ]
  }
}

Method [ <internal:Reflection> public method hasConstant ] {

  - Parameters [1] {
    Parameter #0 [ <required> $name ]
  }
}

Method [ <internal:Reflection> public method getConstants ] {
}
```

```
Method [ <internal:Reflection> public method getConstant ] {

  - Parameters [1] {
    Parameter #0 [ <required> $name ]
  }
}

Method [ <internal:Reflection> public method getInterfaces ] {
}

Method [ <internal:Reflection> public method getInterfaceNames ] {
}

Method [ <internal:Reflection> public method isInterface ] {
}

Method [ <internal:Reflection> public method isAbstract ] {
}

Method [ <internal:Reflection> public method isFinal ] {
}

Method [ <internal:Reflection> public method getModifiers ] {
}

Method [ <internal:Reflection> public method isInstance ] {

  - Parameters [1] {
    Parameter #0 [ <required> $object ]
  }
}

Method [ <internal:Reflection> public method newInstance ] {

  - Parameters [1] {
    Parameter #0 [ <required> $args ]
  }
}

Method [ <internal:Reflection> public method newInstanceArgs ] {

  - Parameters [1] {
    Parameter #0 [ <optional> array $args ]
  }
}

Method [ <internal:Reflection> public method getParentClass ] {
}

Method [ <internal:Reflection> public method isSubclassOf ] {

  - Parameters [1] {
    Parameter #0 [ <required> $class ]
  }
}

Method [ <internal:Reflection> public method getStaticProperties ] {
}

Method [ <internal:Reflection> public method getStaticPropertyValue ] {

  - Parameters [2] {
    Parameter #0 [ <required> $name ]
    Parameter #1 [ <optional> $default ]
  }
}

Method [ <internal:Reflection> public method setStaticPropertyValue ] {

  - Parameters [2] {
    Parameter #0 [ <required> $name ]
    Parameter #1 [ <required> $value ]
  }
}

Method [ <internal:Reflection> public method getDefaultProperties ] {
}
```

```
    Method [ <internal:Reflection> public method isIterateable ] {
    }

    Method [ <internal:Reflection> public method implementsInterface ] {

      - Parameters [1] {
        Parameter #0 [ <required> $interface ]
      }
    }

    Method [ <internal:Reflection> public method getExtension ] {
    }

    Method [ <internal:Reflection> public method getExtensionName ] {
    }
  }
}
```

As you scroll past all of the methods available you will see the getName(), getConstant(), and getConstants() methods that we previously discussed. It would be beneficial to just play with these methods on your own to become more acquainted with them. As you will discover, they can be a very helpful and powerful utility in unit testing.

Chapter 9
The PEAR Library

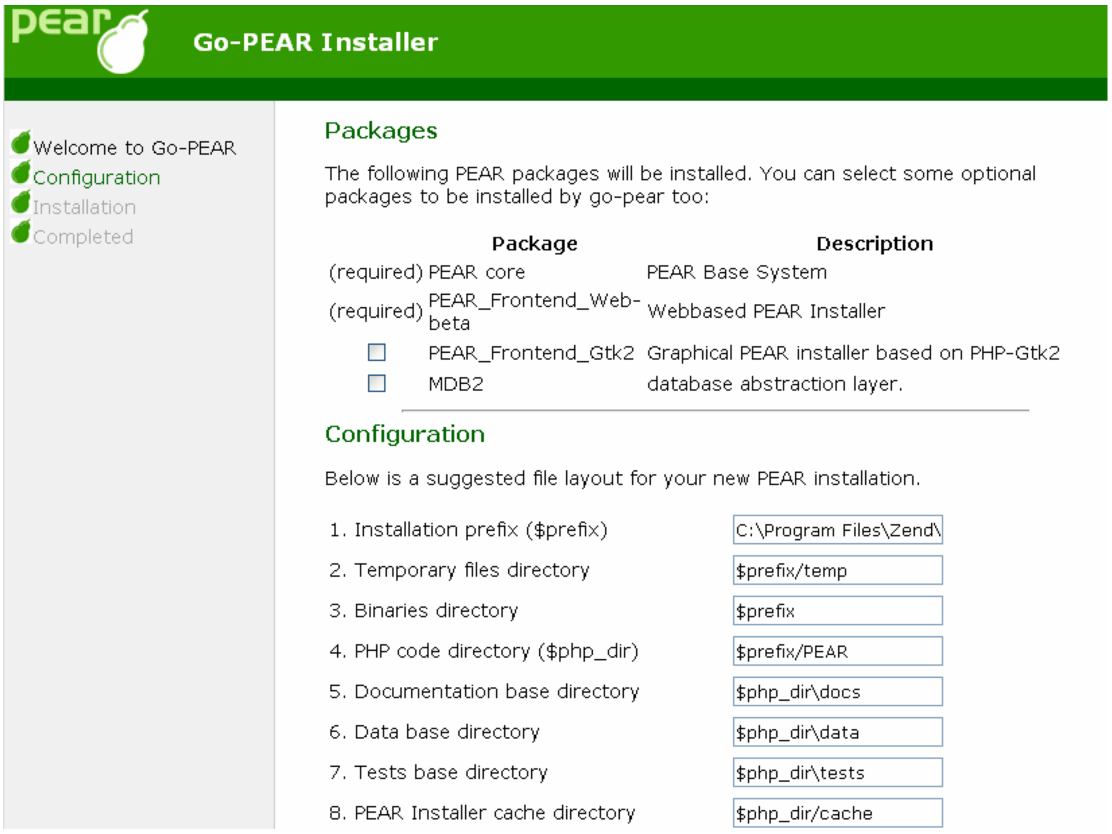
PEAR stands for PHP Extension and Application Repository, which is a framework and distribution system for reusable PHP components and classes. Aside from the DB abstraction packages, PEAR library contains huge amount of useful classes for work with XML, CURL etc. Using PEAR can save you great amount of time to code something that other people already coded, tested, and used. For example, if you need a HTML form validation routine, PEAR has it in its Validate Package. The PEAR repository is centrally managed at http://pear.php.net, so when you use an official PEAR package you can be sure of it's quality and reliability.

Full list of maintained packages is available here.

Install PEAR in a Windows Zend Framework Environment

1. Open this page in any browser http://pear.php.net/go-pear and save it as a PHP file in C:\Program Files\Zend\Apache2\htdocs\go-pear.php

2. Open a browser and go to http://localhost/go-pear.php



Packages

The following PEAR packages will be installed. You can select some optional packages to be installed by go-pear too:

	Package	Description
(required)	PEAR core	PEAR Base System
(required)	PEAR_Frontend_Web-beta	Webbased PEAR Installer
☐	PEAR_Frontend_Gtk2	Graphical PEAR installer based on PHP-Gtk2
☐	MDB2	database abstraction layer.

Configuration

Below is a suggested file layout for your new PEAR installation.

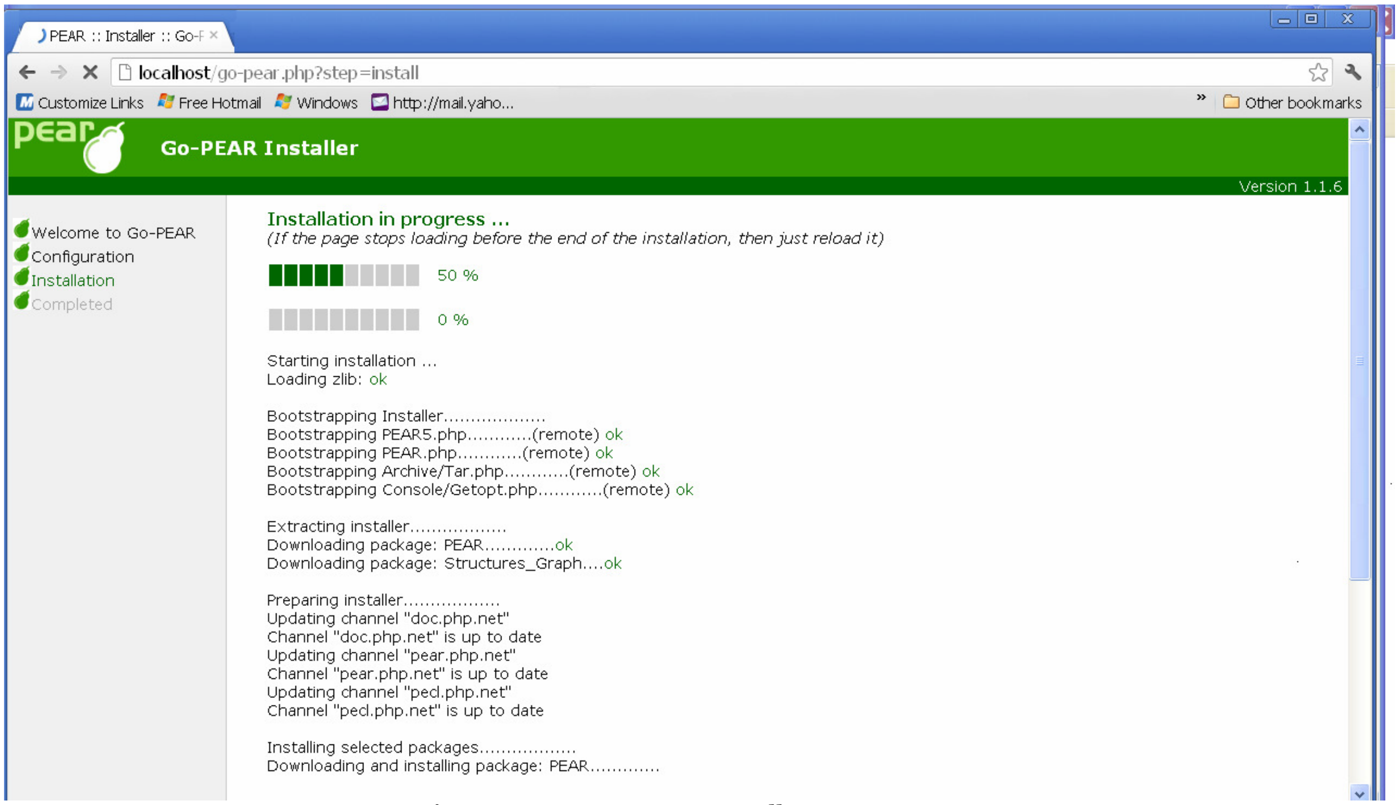
1. Installation prefix ($prefix)	C:\Program Files\Zend\
2. Temporary files directory	$prefix/temp
3. Binaries directory	$prefix
4. PHP code directory ($php_dir)	$prefix/PEAR
5. Documentation base directory	$php_dir\docs
6. Data base directory	$php_dir\data
7. Tests base directory	$php_dir\tests
8. PEAR Installer cache directory	$php_dir/cache

Figure 10-1. *Go-PEAR Installation Settings*

3. When installing PEAR make sure you set:

"1. Installation prefix ($prefix)" = "**C:\Program Files\Zend\ZendServer\bin**"

4. Add the PEAR installation path (**C:\Program Files \Zend\ZendServer\bin\PEAR**) to your "Path" environment variable located in MyComputer(right-click)->Properties->Advanced System Settings->Environment Variables. Name the variable "**PEAR**".



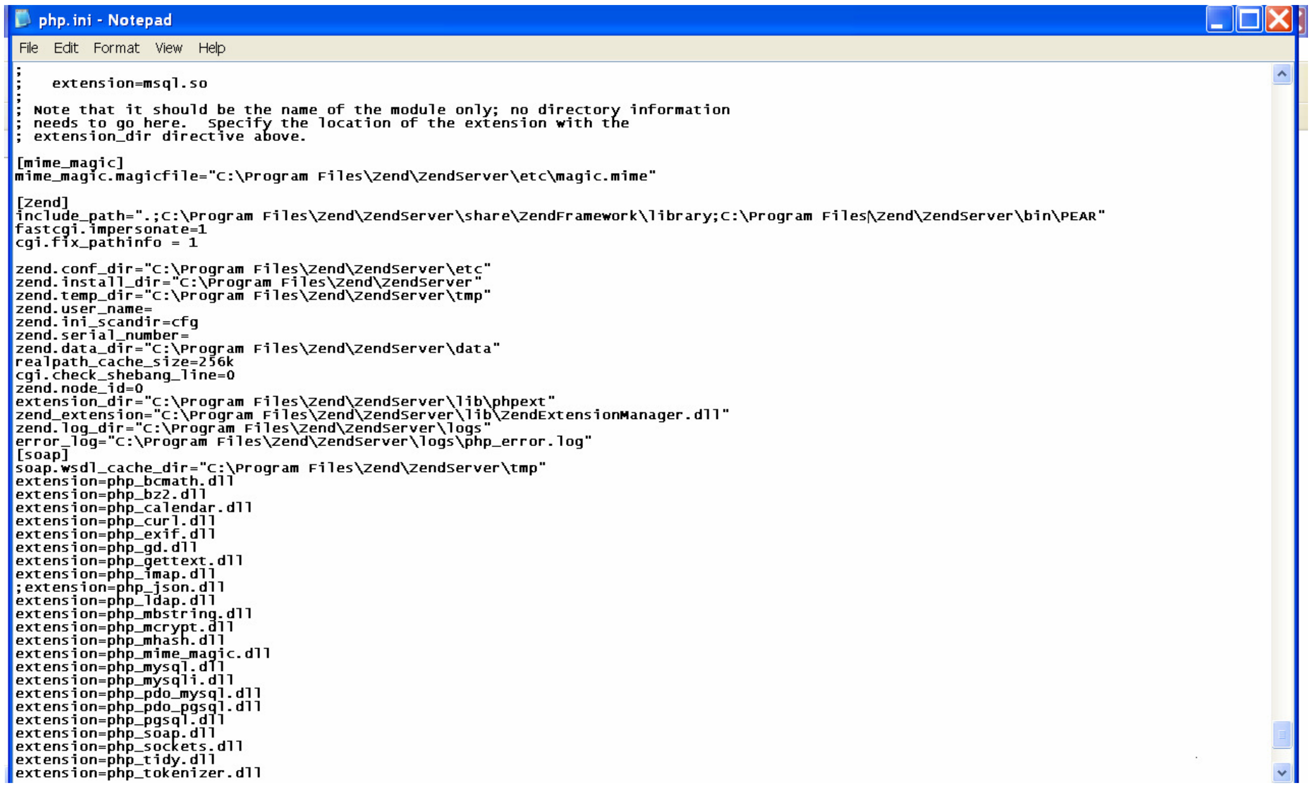
Installation in progress ...
(If the page stops loading before the end of the installation, then just reload it)

```
50 %

0 %

Starting installation ...
Loading zlib: ok

Bootstrapping Installer..................
Bootstrapping PEAR5.php............(remote) ok
Bootstrapping PEAR.php............(remote) ok
Bootstrapping Archive/Tar.php............(remote) ok
Bootstrapping Console/Getopt.php............(remote) ok

Extracting installer.................
Downloading package: PEAR.............ok
Downloading package: Structures_Graph....ok

Preparing installer.................
Updating channel "doc.php.net"
Channel "doc.php.net" is up to date
Updating channel "pear.php.net"
Channel "pear.php.net" is up to date
Updating channel "pecl.php.net"
Channel "pecl.php.net" is up to date

Installing selected packages.................
Downloading and installing package: PEAR.............
```

Figure 10-2. *Go-PEAR Installer Progress Page*

5. Go to C:\Program Files\Zend\ZendServer\etc\php.ini and add the PEAR path to your include path

```
[Zend]
include_path=".; C:\Program
Files\Zend\ZendServer\share\ZendFramework\library;C:\Program
Files\Zend\zendserver\bin\PEAR"
```



```
;
;   extension=msql.so
;
; Note that it should be the name of the module only; no directory information
; needs to go here.  Specify the location of the extension with the
; extension_dir directive above.

[mime_magic]
mime_magic.magicfile="C:\Program Files\Zend\ZendServer\etc\magic.mime"

[Zend]
include_path=".;C:\Program Files\Zend\ZendServer\share\ZendFramework\library;C:\Program Files\Zend\ZendServer\bin\PEAR"
fastcgi.impersonate=1
cgi.fix_pathinfo = 1

zend.conf_dir="C:\Program Files\Zend\ZendServer\etc"
zend.install_dir="C:\Program Files\Zend\ZendServer"
zend.temp_dir="C:\Program Files\Zend\ZendServer\tmp"
zend.user_name=
zend.ini_scandir=cfg
zend.serial_number=
zend.data_dir="C:\Program Files\Zend\ZendServer\data"
realpath_cache_size=256k
cgi.check_shebang_line=0
zend.node_id=0
extension_dir="C:\Program Files\Zend\ZendServer\lib\phpext"
zend_extension="C:\Program Files\Zend\ZendServer\lib\ZendExtensionManager.dll"
zend.log_dir="C:\Program Files\Zend\ZendServer\logs"
error_log="C:\Program Files\Zend\ZendServer\logs\php_error.log"
[soap]
soap.wsdl_cache_dir="C:\Program Files\Zend\ZendServer\tmp"
extension=php_bcmath.dll
extension=php_bz2.dll
extension=php_calendar.dll
extension=php_curl.dll
extension=php_exif.dll
extension=php_gd.dll
extension=php_gettext.dll
extension=php_imap.dll
;extension=php_json.dll
extension=php_ldap.dll
extension=php_mbstring.dll
extension=php_mcrypt.dll
extension=php_mhash.dll
extension=php_mime_magic.dll
extension=php_mysql.dll
extension=php_mysqli.dll
extension=php_pdo_mysql.dll
extension=php_pdo_pgsql.dll
extension=php_pgsql.dll
extension=php_soap.dll
extension=php_sockets.dll
extension=php_tidy.dll
extension=php_tokenizer.dll
```

Figure 10-3. *Modified php.ini File*

6. RESTART your PC

Install PEAR in a Windows XAMPP Environment

1. Go to folder c:\xampp\php and find the file pear.bat.

2. Open that file in your text editor. Find the following line in that file.

```
IF “%PHP_PEAR_INSTALL_DIR%”==”" SET “PHP_PEAR_INSTALL_DIR=\xampp\php\pear”
```

3. Add following line above this line.

```
SET “PHP_PEAR_INSTALL_DIR=C:\xampp\php\PEAR”
```

4. Now include c:\xampp\php to PATH environment variable.

5. In windows vista you can find it here.

Start->Control Panel->System and Maintenance->System->Advanced System Settings->Environment Variable

6. Now go to command prompt and type command pear. You will see all the possible options for that command.

7. Use following command to install pear package.

```
pear install –o 'path of package'
```

Install PEAR in a Windows WAMP Environment

1. Open a command prompt

2. Go to your PHP folder (ex: c:\wamp2\bin\php\php5.9.2-2)

3. Run "go-pear" to install PEAR

Install PEAR on Mac OS

Instead of upgrading Mac OS X's PEAR install, install your own copy in `/usr/local/`.

Open /Applications/Utilities/Terminal.app and enter this command:

```
$ cd /usr/local
```

Next, we begin the PEAR installation process. Enter this command into the Terminal:

```
$ curl http://pear.php.net/go-pear | sudo php
```

Enter your administrator password if prompted and answer any questions that follow. You should be okay if you accept the default answers for each question. When the script finishes, PEAR should be installed in `/usr/local/bin/`. The PEAR library should be accessible at `/usr/local/PEAR/`.

Update system PATH

Now we need to add our custom pear install to our system PATH. Create or edit your bash profile in vi (or TextMate):

```
$ vi ~/.bash_profile
```

Ensure this file includes the following line of text:

```
PATH=/usr/local/bin:$PATH
```

Save the file and restart the Terminal application for this change to take effect. Next, we should verify that pear works.

Run this command in the Terminal:

```
which pear
```

This command should respond with: `/usr/local/bin/pear`.

Update PHP include path

Now we need to tell PHP where our PEAR library is located by adding pear to the PHP include path in the `/etc/php.ini` file.

This file does not exist by default on Mac OS X 10.6. To create this file, run the following command in the Terminal:

```
$ sudo cp /etc/php.ini.default /etc/php.ini
```

Next, we need to edit /etc/php.ini file in vi or TextMate and update the PHP include path. Locate the following line in `/etc/php.ini`:

```
;include_path = ”/php/includes”
```

Remove the ”;” from the beginning of the line and add the PEAR library path.

```
include_path = ”/usr/local/PEAR:/php/includes”
```

Save /etc/php.ini. Restart the Apache web server by unchecking and rechecking System Preferences > Sharing > Web Sharing.

Next, view http://localhost/~[your_user_name]/index.php in a web browser. Search for “include_path” and verify the path now includes ”`/usr/local/PEAR`”. PEAR is now installed.

Install PEAR on Linux

PEAR packages are installed in a configurable location. On a Linux (or Unix) system this will usually be /usr/local/lib/php

Type the command at the command line:

```
$ pear
```

If you get:

```
$ pear – command not found
```

then it means PEAR is not installed in your Linux server. To install it just follow the steps below at the command prompt.

Download the installation php file from **http://pear.php.net/go-pear.**

```
$ wget http://pear.php.net/go-pear
```

Rename to php file

```
$ cp go-pear go-pear.php
```

Run the php script from command line.

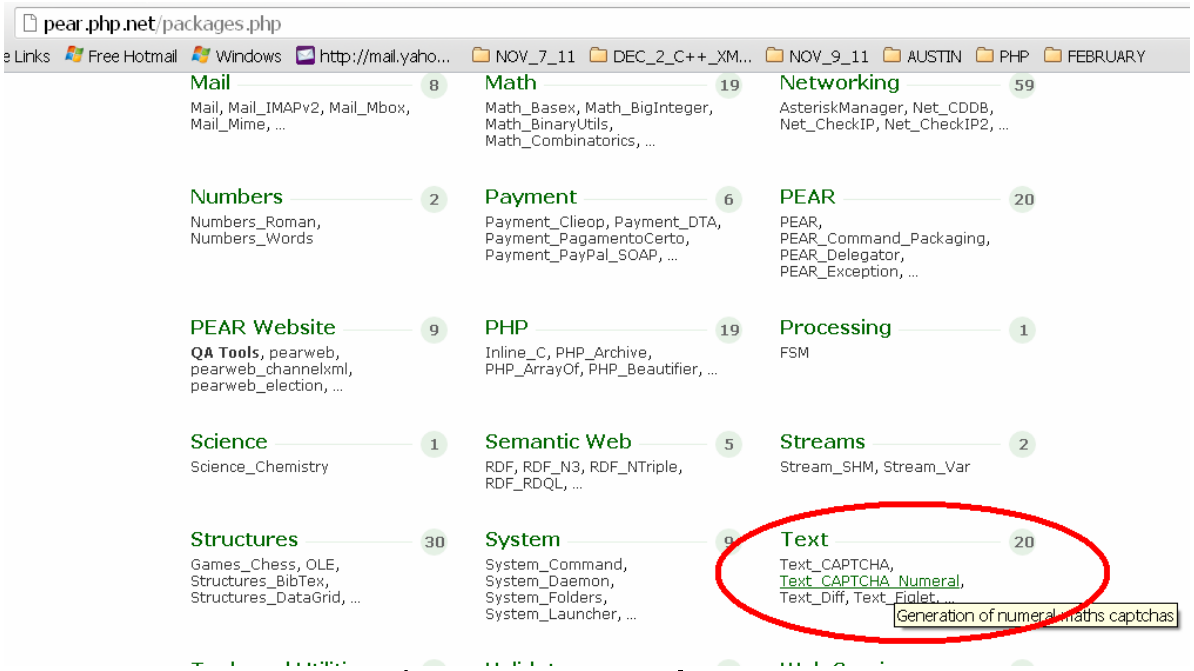
```
$ php go-pear.php
```

and you will see the installation begin. Once the binary is installed, you can install php various extensions using this command.

```
$ pear install package
```

Using PEAR - EXAMPLE:

Text -> CAPTCHA_Numeral



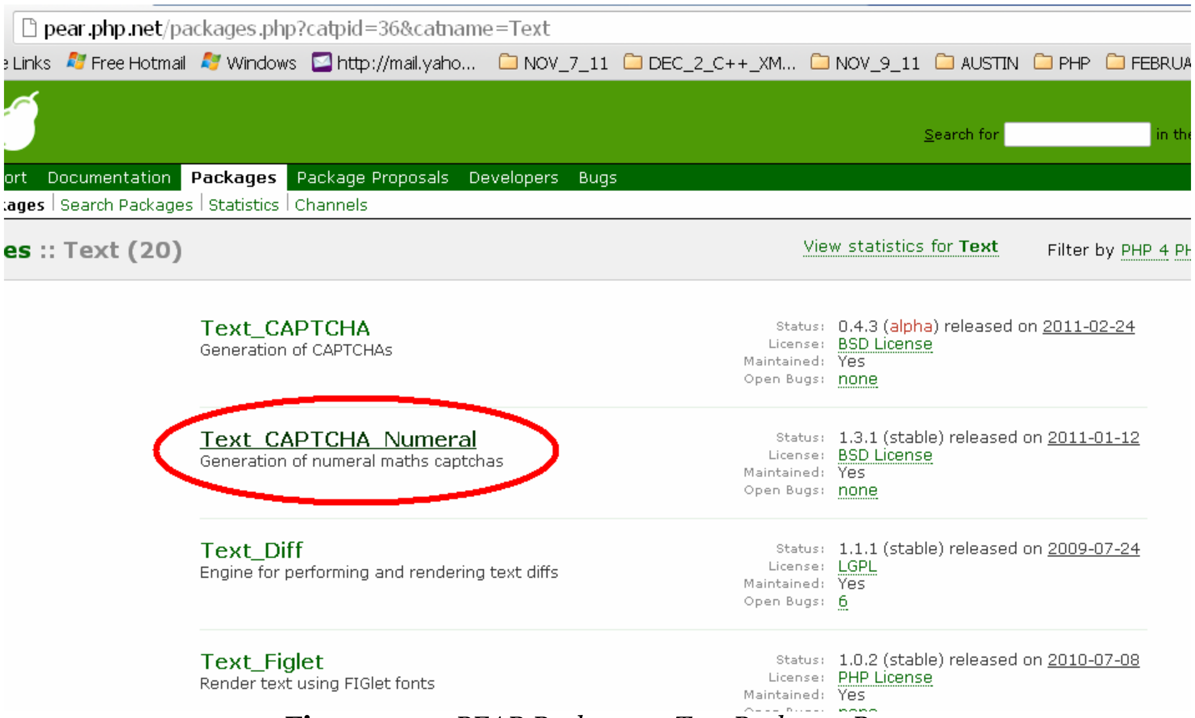
pear.php.net/packages.php

e Links | Free Hotmail | Windows | http://mail.yaho... | NOV_7_11 | DEC_2_C++_XM... | NOV_9_11 | AUSTIN | PHP | FEBRUARY

Mail 8 Mail, Mail_IMAPv2, Mail_Mbox, Mail_Mime, ...	Math 19 Math_Basex, Math_BigInteger, Math_BinaryUtils, Math_Combinatorics, ...	Networking 59 AsteriskManager, Net_CDDB, Net_CheckIP, Net_CheckIP2, ...
Numbers 2 Numbers_Roman, Numbers_Words	Payment 6 Payment_Clieop, Payment_DTA, Payment_PagamentoCerto, Payment_PayPal_SOAP, ...	PEAR 20 PEAR, PEAR_Command_Packaging, PEAR_Delegator, PEAR_Exception, ...
PEAR Website 9 QA Tools, pearweb, pearweb_channelxml, pearweb_election, ...	PHP 19 Inline_C, PHP_Archive, PHP_ArrayOf, PHP_Beautifier, ...	Processing 1 FSM
Science 1 Science_Chemistry	Semantic Web 5 RDF, RDF_N3, RDF_NTriple, RDF_RDQL, ...	Streams 2 Stream_SHM, Stream_Var
Structures 30 Games_Chess, OLE, Structures_BibTex, Structures_DataGrid, ...	System 9 System_Command, System_Daemon, System_Folders, System_Launcher, ...	Text 20 Text_CAPTCHA, Text_CAPTCHA_Numeral, Text_Diff, Text_Figlet, ...

Generation of numeral maths captchas

Figure 10-4. *PEAR Packages Page*



pear.php.net/packages.php?catpid=36&catname=Text

e Links | Free Hotmail | Windows | http://mail.yaho... | NOV_7_11 | DEC_2_C++_XM... | NOV_9_11 | AUSTIN | PHP | FEBRUA

Search for | in the

ort | Documentation | **Packages** | Package Proposals | Developers | Bugs

ages | Search Packages | Statistics | Channels

es :: Text (20) View statistics for **Text** Filter by PHP 4 PH

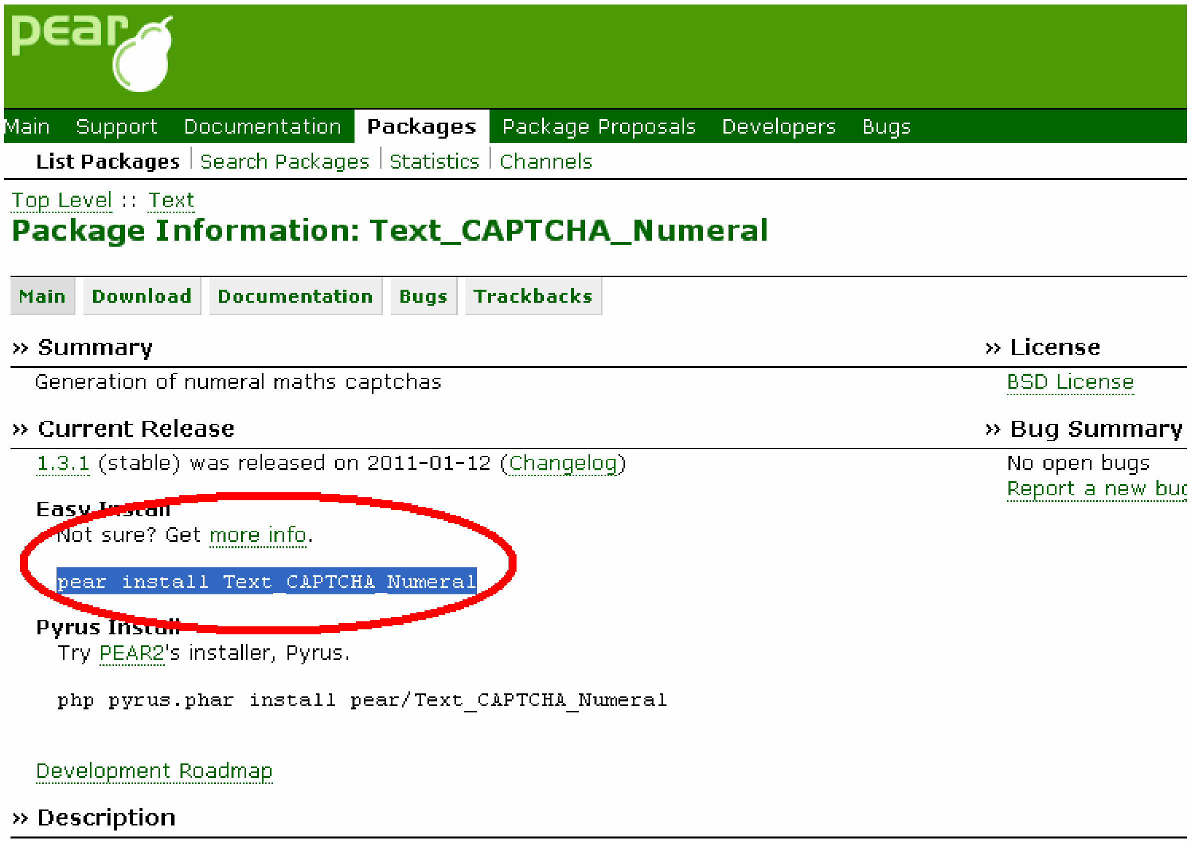
Text_CAPTCHA
Generation of CAPTCHAs
Status: 0.4.3 (alpha) released on 2011-02-24
License: BSD License
Maintained: Yes
Open Bugs: none

Text_CAPTCHA_Numeral
Generation of numeral maths captchas
Status: 1.3.1 (stable) released on 2011-01-12
License: BSD License
Maintained: Yes
Open Bugs: none

Text_Diff
Engine for performing and rendering text diffs
Status: 1.1.1 (stable) released on 2009-07-24
License: LGPL
Maintained: Yes
Open Bugs: 6

Text_Figlet
Render text using FIGlet fonts
Status: 1.0.2 (stable) released on 2010-07-08
License: PHP License
Maintained: Yes

Figure 10-5. *PEAR Packages – Text Packages Page*



Main Support Documentation **Packages** Package Proposals Developers Bugs

List Packages | Search Packages | Statistics | Channels

Top Level :: Text

Package Information: Text_CAPTCHA_Numeral

Main | Download | Documentation | Bugs | Trackbacks

» Summary

Generation of numeral maths captchas

» License

BSD License

» Current Release

1.3.1 (stable) was released on 2011-01-12 (Changelog)

Easy Install

Not sure? Get more info.

```
pear install Text_CAPTCHA_Numeral
```

Pyrus Install

Try PEAR2's installer, Pyrus.

```
php pyrus.phar install pear/Text_CAPTCHA_Numeral
```

Development Roadmap

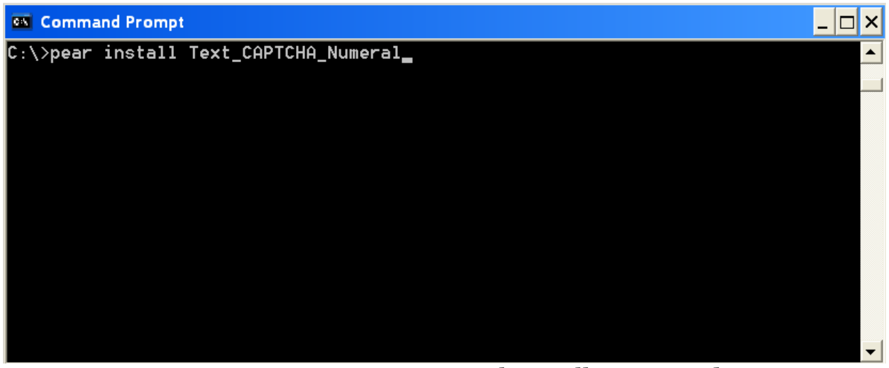
» Bug Summary

No open bugs

Report a new buc

» Description

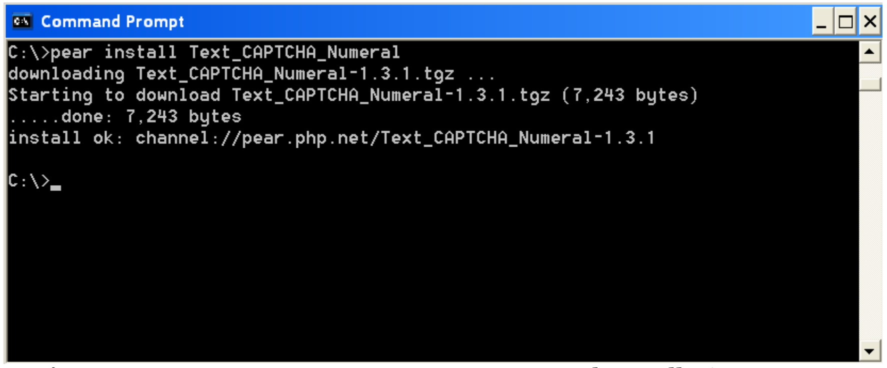
Figure 10-6. *PEAR Text_CAPTCHA_Numeral Install Page*



Command Prompt

```
C:\>pear install Text_CAPTCHA_Numeral_
```

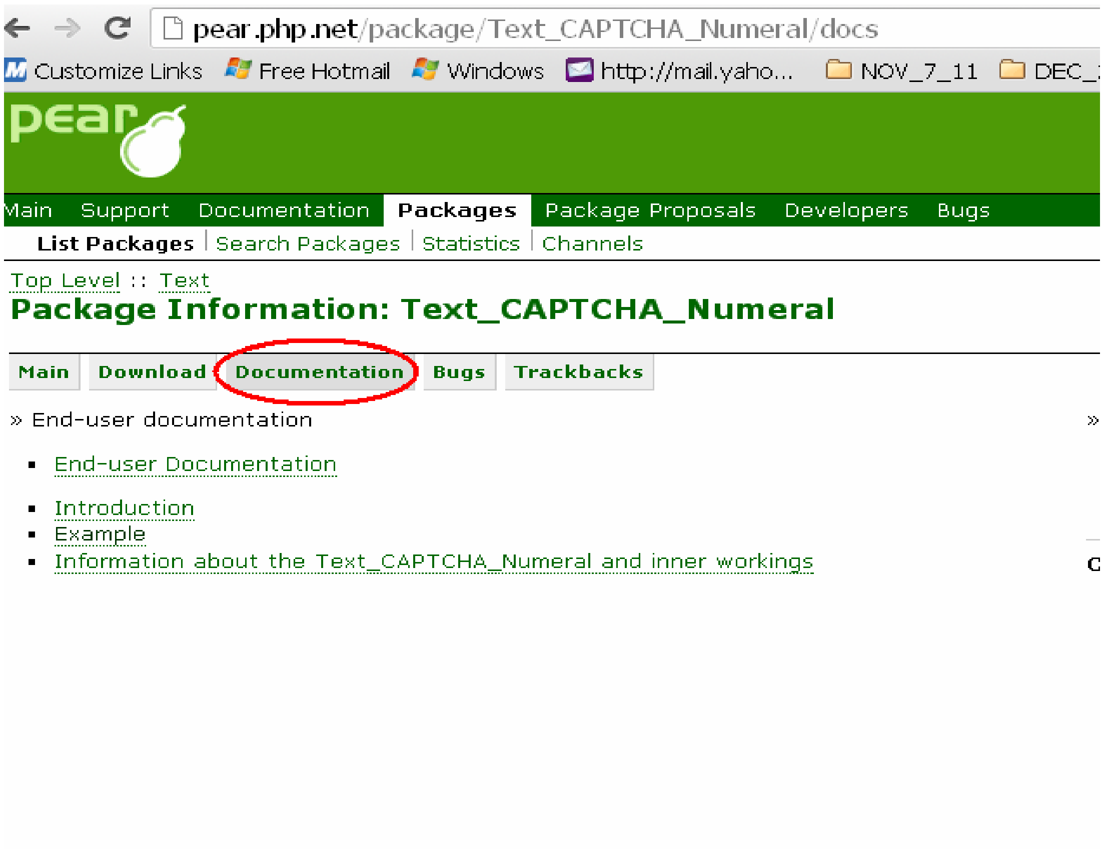
Figure 10-7. *PEAR Text_CAPTCHA_Numeral Install Command Prompt Command*



Command Prompt

```
C:\>pear install Text_CAPTCHA_Numeral
downloading Text_CAPTCHA_Numeral-1.3.1.tgz ...
Starting to download Text_CAPTCHA_Numeral-1.3.1.tgz (7,243 bytes)
.....done: 7,243 bytes
install ok: channel://pear.php.net/Text_CAPTCHA_Numeral-1.3.1

C:\>_
```

Figure 10-8. *PEAR Text_CAPTCHA_Numeral Installation Message*



Top Level :: Text

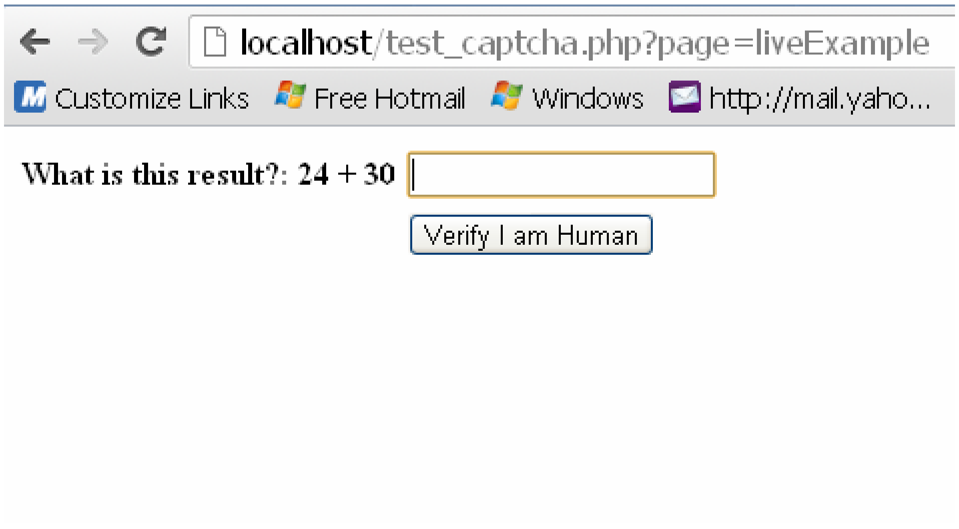
Package Information: Text_CAPTCHA_Numeral

» End-user documentation

- End-user Documentation
- Introduction
- Example
- Information about the Text_CAPTCHA_Numeral and inner workings

Figure 10-9. *PEAR Text_CAPTCHA_Numeral Documentation Page*

```
//test_captcha.php --
<?php
require_once 'Text/CAPTCHA/Numeral.php';
$numcap = new Text_CAPTCHA_Numeral;

if (isset($_POST['captcha']) && isset($_SESSION['answer'])) {
    if ($_POST['captcha'] == $_SESSION['answer']) {
        $errors[] = 'Ok... You might be human...';
    } else {
        $errors[] = 'You are dumb or not human';
    }
}
    if (!empty($errors)) {
        foreach ($errors as $error) {
            print "<h1><font color='red'>$error</font></h1><br />";
        }
    }


    print '
        <form name="capter" action="test_captcha.php?page=liveExample" method="post">
         <table>
          <tr>
           <th>What is this result?: '.$numcap->getOperation().'</th>
           <td><input type="text" value="" name="captcha" /></td>
          </tr>
          <tr>
           <th/>
           <td><input type="submit" value="Verify I am Human" /></td>
          </tr>
        </form>
    ';
    $_SESSION['answer'] = $numcap->getAnswer();
?>
```



What is this result?: 24 + 30

Figure 10-10. *PEAR Text_CAPTCHA_Numeral Browser Output*

Chapter 10
Unit Testing with PHPUnit

Unit Testing with PHPUnit

Testing your software is imperative however waiting until just before a release to do it can bring a plethora of headaches in and of itself. Yes, I said a plethora and I do know what a plethora is in case there are any Three Amigos fans reading this. Functional tests are a test-just-prior-to-release approach that many developers are familiar with, especially user acceptance testing which usually begins with a run through of the features of the user interface. Problems that are detected during user acceptance tests can be expensive to fix and can delay the release of your software, especially for poorly organized applications that are comprised of tightly coupled code. Change requests can be equally as difficult and costly to implement when the developer is relying on functional tests to verify that a change has been implemented properly.

The point is that all programmers make mistakes but the better programmers use tests to detect their mistakes as soon as possible. The sooner you test for a mistake, the greater your chance of finding it and the less it will cost to find and fix. This explains why leaving testing until just before releasing the software or limiting your test to simply checking to see that your program behaving as expected is inefficient and can be conducive to problems. Many errors slip past this kind of testing and emerge in production causing disruptions and utter embarrassment for the developer.

The most effective way to prevent errors from sneaking into production is to catch them early on in development by performing a battery of tests using runnable code fragments that automatically test the correctness of parts, or units, of the software. These runnable code fragments are called unit tests. Automating unit tests within the structure of a unit testing software framework can assure that your code modules will work as intended and without errors in production.

PHPUnit is a unit testing software framework for PHP and it supports the development of object-oriented PHP applications using the concepts and methods of Design-by-Contract development. It is one of the xUnit family of frameworks that which includes JUnit for Java and NUint for .NET.

PHPUnit was developed by Sebastian Bergmann and is not to be confused with PHP Unit Testing Framework which was developed by Ed Heal and released in April of 2011. Although there are other testing frameworks for PHP, PHPUnit is the de-facto standard for unit testing in PHP projects and will be the only one we will review.

Installing PHPUnit

PHPUnit should be installed using the PEAR (PHP Extension and Application Repository) Installer.

Install PHPUnit on Windows on Zend Framework Environment

Before you start delete everything from C:\Users\YOUR_USER_NAME\AppData\Local\Temp\

1. Open a cmd window. You might need to Run As Administrator

2. Run each of the following commands SEPARATELY (and each only once!):

```
pear channel-discover pear.phpunit.de
pear channel-discover components.ez.no
pear channel-discover pear.symfony-project.com
pear clear-cache
```

3. Now run:

```
pear install phpunit/PHPUnit
```

Install PHPUnit on Windows on XAMPP Environment

These procedures are assuming xampp is installed to C:\xampp
1. Open a command prompt and go to C:\xampp\php
2. Run each of the following commands SEPARATELY (and each only once!):

```
pear update-channels
pear upgrade
pear channel-discover components.ez.no
pear channel-discover pear.symfony-project.com
pear channel-discover pear.phpunit.de
pear clear-cache
pear install --alldeps phpunit/PHPUnit
```

Note:
You may have to edit "memory_limit" in your php.ini if you get some sort of memory error, just set it to something really large, then back when your done.

Install PHPUnit on Mac OS X

To install PHPUnit on a Mac OS X assuming that you followed the PEAR installation procedures for Mac OS in Chapter 9, simply run this command in the Terminal:

```
$ sudo pear channel-discover pear.phpunit.de
```

Next, run this command in the Terminal:

```
$ sudo pear install phpunit/PHPUnit
```

That's it. PHPUnit is installed and located at `/usr/local/PEAR/PHPUnit/`. You can now include PHPUnit in your PHP scripts with this line:

```
require_once 'PHPUnit/Framework.php';
```

Install PHPUnit on Linux

From the command line, enter the following commands:

```
$ ./bin/pear channel-discover pear.phpunit.de
$ ./bin/pear channel-discover components.ez.no
$ ./bin/pear channel-discover pear.symfony-project.com
$ ./bin/pear install phpunit/PHPUnit
```

Test fixtures

A test fixture refers to the fixed state or set of preconditions used as a baseline for running tests, it is essentially a dataset created exclusively for testing purposes. It is also known as a test context. The developer should set up a known good state before the tests, and return to the original state after the tests. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable. An example of a fixture in general is erasing a hard disk and installing a known clean operating system or loading a test database with a specific, known set of data. You would never want to use a production database for unit testing because the tests may alter the data. Fixtures, such as a test database, should be loaded before each test suite is executed.

Test Phases

Setup – Setting up the test fixture
Exercise – Interact with the system under the test
Verify – Determine whether or not the expected outcome has been obtained
Tear Down – Tear down the test fixture to return to the original state.

Writing Test Cases

Writing test cases with PHPUnit is very simple. You just need to include the PHPUnit/Framework.php file, declare a class that extends PHPUnit_Framework_TestCase and then have all your testing functions starting with the name "test" and use $this->assertTrue to check whether the condition is met. The following shows a complete but very simple example using PHPUnit.

```
<?php
//test_assertion.php

class SampleTest extends PHPUnit_Framework_TestCase {
  private $x;
  private $y;

  public function setup() {
    $this->x = 4;
    $this->y = '4';
  }

  public function testEqual() {
    if ( ($this->x == $y) && ($this->x!= NULL) && ($this->y!= NULL) ) {
        echo "$x is equal to $y";
        }
  }

  public function testIdentical() {
    if ( ($this->x === $y) && ($this->x!= NULL) && ($this->y!= NULL) ) {
        echo "$x is identical to $y";
        }
  }


  public function teardown() {
    $this->x = '';
    $this->y = '';
  }

}
?>
```

The setup() and teardown() methods serve to initialize and clean up test fixtures.

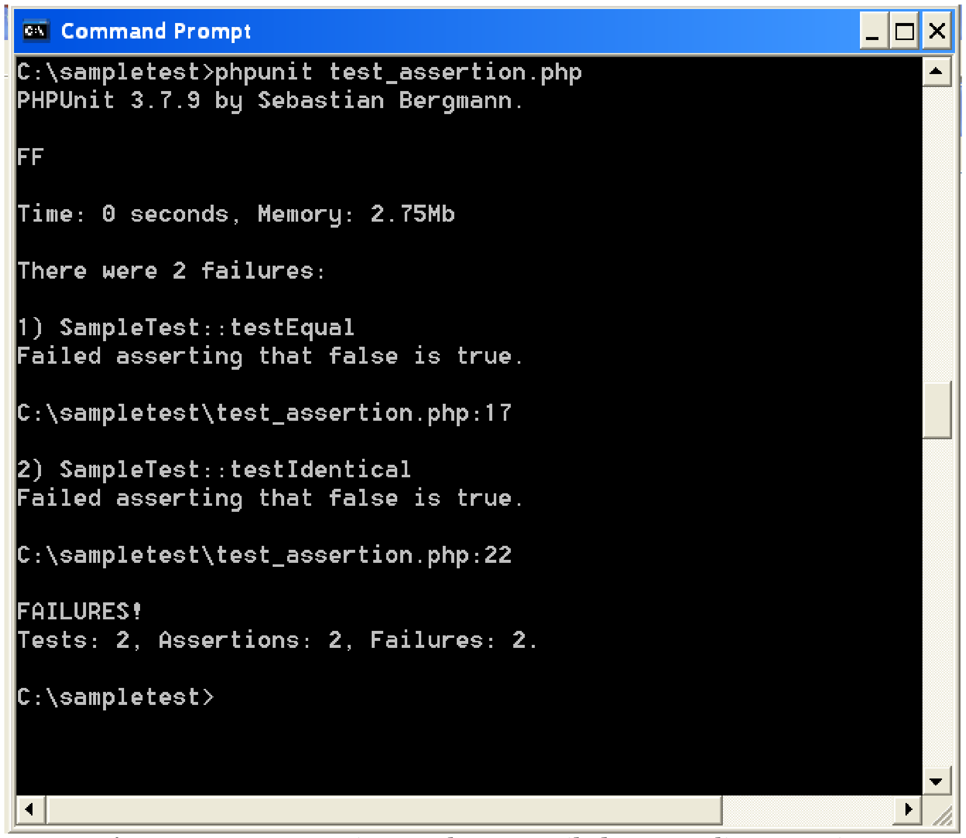
Assertion Methods

An assertion is a method that allows you to check your assumptions about an aspect of your system. A typical way to use an assertion is to define an expectation about the outcome of a code statement or segment that is under test. If the outcome of a test is not what was expected then the test fails and a warning message is generated. Calls to assertion methods also help document how the system under test is supposed to behave.

Let's modify the example test case above and use an assertion method instead:

```
<?php
//test_assertion.php

class SampleTest extends PHPUnit_Framework_TestCase {

  private $x;
  private $y;

  public function setup()
  {
    $this->x = 3;
    $this->y = '7';
  }

  public function testEqual() {
    $this->assertTrue($this->x == $this->y);
  }

  public function testIdentical() {
    $this->assertTrue($this->x === $this->y);
  }

  public function teardown() {
    $x = '';
    $y = '';
  }
}
?>
```

All assertion methods can take an optional description as a last parameter. This is to label the displayed result with. If omitted a default message is sent instead, which is usually sufficient. This default message can still be embedded in your own message if you include "%s" within the string. In this case all the assertions return true on a pass or false on failure.

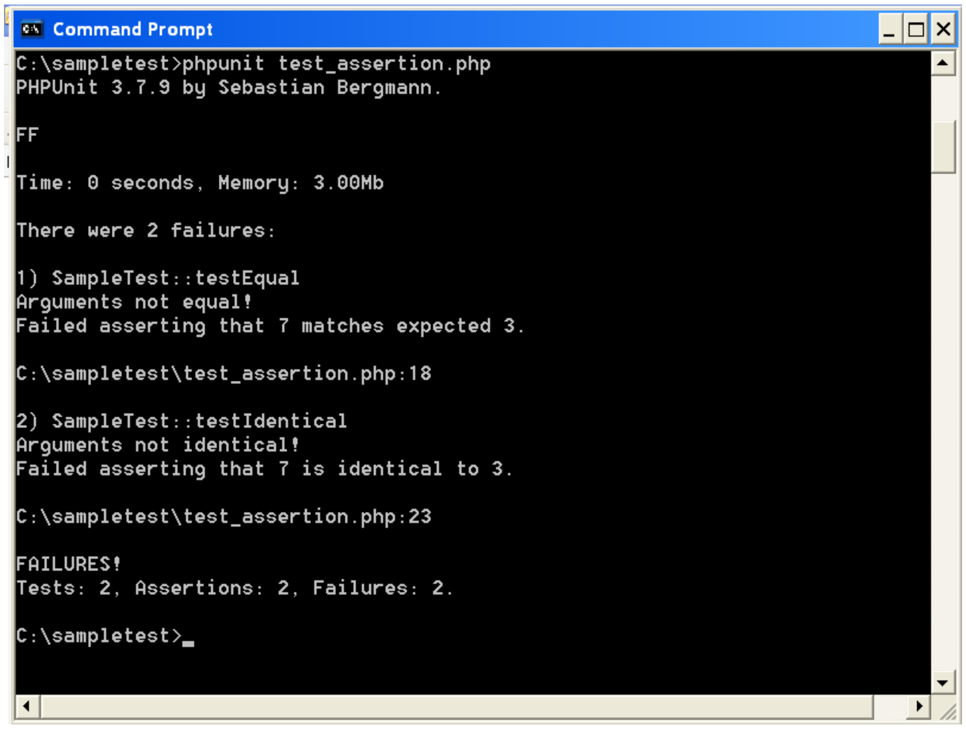


```
C:\sampletest>phpunit test_assertion.php
PHPUnit 3.7.9 by Sebastian Bergmann.

FF

Time: 0 seconds, Memory: 2.75Mb

There were 2 failures:

1) SampleTest::testEqual
Failed asserting that false is true.

C:\sampletest\test_assertion.php:17

2) SampleTest::testIdentical
Failed asserting that false is true.

C:\sampletest\test_assertion.php:22

FAILURES!
Tests: 2, Assertions: 2, Failures: 2.

C:\sampletest>
```

Figure 11-1. *PHPUnit SampleTest: Failed on Equality Assertion*

Let’s test these conditions again using the assertEqual() and assertSame() methods:

```
<?php
//test_assertion.php

class SampleTest extends PHPUnit_Framework_TestCase {

  private $x;
  private $y;

  public function setup()
  {
    $this->x = 3;
    $this->y = '7';
  }

  public function testEqual() {
    $this->assertEquals($this->x, $this->y, ‘Failed! Arguments not Equal!);
  }

  public function testIdentical() {
    $this->assertSame($this->x, $this->y, ‘Failed, Arguments not Identical!’);
  }

  public function teardown() {
    $x = '';
    $y = '';
  }
}

?>
```

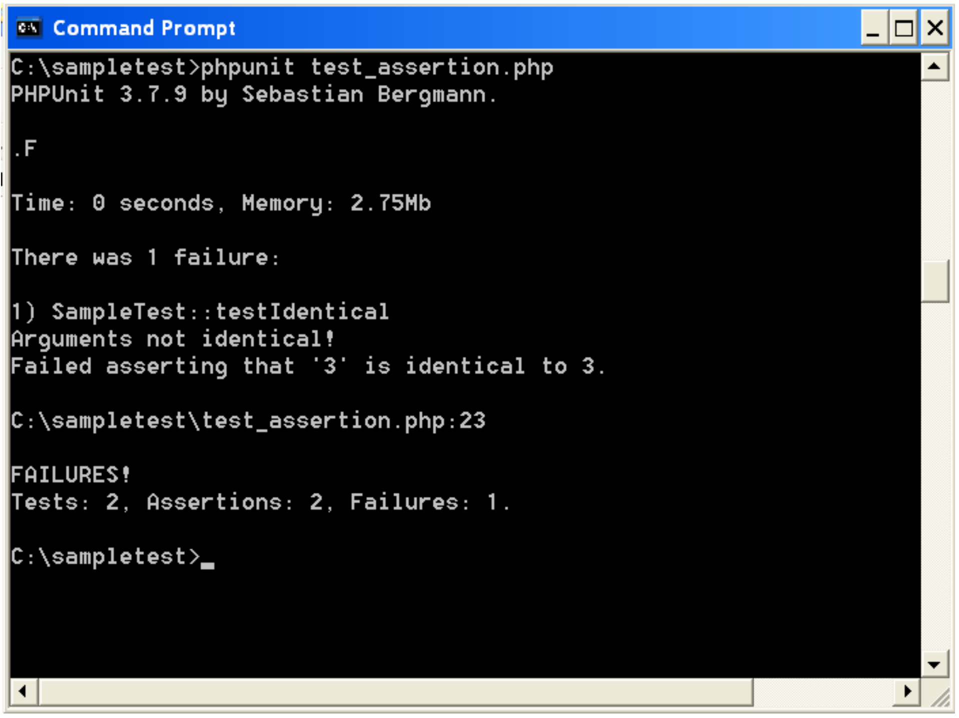


```
C:\sampletest>phpunit test_assertion.php
PHPUnit 3.7.9 by Sebastian Bergmann.

FF

Time: 0 seconds, Memory: 3.00Mb

There were 2 failures:

1) SampleTest::testEqual
Arguments not equal!
Failed asserting that 7 matches expected 3.

C:\sampletest\test_assertion.php:18

2) SampleTest::testIdentical
Arguments not identical!
Failed asserting that 7 is identical to 3.

C:\sampletest\test_assertion.php:23

FAILURES!
Tests: 2, Assertions: 2, Failures: 2.

C:\sampletest>_
```

Figure 11-2. *PHPUnit SampleTest: Failed on Identical Assertion*

Now let’s set $x and $y equal to each other but make them different types and test that they are not identical:

```
<?php
//test_assertion.php

class SampleTest extends PHPUnit_Framework_TestCase {

  private $x;
  private $y;

  public function setup()
  {
    $this->x = 3;
    $this->y = ‘3’;
  }

  public function testEqual() {
    $this->assertTrue($this->x == $this->y);
  }

  public function testIdentical() {
    $this->assertTrue($this->x === $this->y);
  }

  public function teardown() {
    $x = '';
    $y = '';
  }
}
```

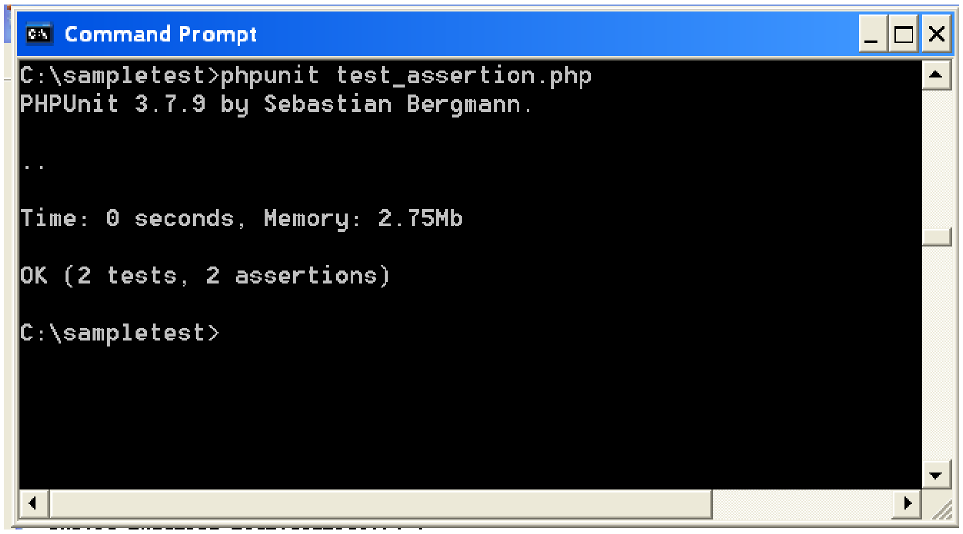


```
C:\sampletest>phpunit test_assertion.php
PHPUnit 3.7.9 by Sebastian Bergmann.

.F

Time: 0 seconds, Memory: 2.75Mb

There was 1 failure:

1) SampleTest::testIdentical
Arguments not identical!
Failed asserting that '3' is identical to 3.

C:\sampletest\test_assertion.php:23

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.

C:\sampletest>_
```

Figure 11-3. *PHPUnit SampleTest: Failed on Identical Assertion*

Now let's run this one last time setting $x = 3 and $y = 3 so that they are equal and of the same type:

```
<?php
//test_assertion.php

class SampleTest extends PHPUnit_Framework_TestCase {

  private $x;
  private $y;

  public function setup()
  {
    $this->x = 3;
    $this->y = 3;
  }

  public function testEqual() {
    $this->assertTrue($this->x == $this->y);
  }

  public function testIdentical() {
    $this->assertTrue($this->x === $this->y);
  }

  public function teardown() {
    $x = '';
    $y = '';
  }
}

?>
```



```
C:\sampletest>phpunit test_assertion.php
PHPUnit 3.7.9 by Sebastian Bergmann.

..

Time: 0 seconds, Memory: 2.75Mb

OK (2 tests, 2 assertions)

C:\sampletest>
```

Figure 11-4. *PHPUnit SampleTest: Passed.*

Below is a list of built in PHPUnit assertion methods:

`assertArrayHasKey()`
`assertClassHasAttribute()`
`assertClassHasStaticAttribute()`
`assertContains()`
`assertContainsOnly()`
`assertContainsOnlyInstancesOf()`
`assertCount()`
`assertEmpty()`
`assertEqualXMLStructure()`
`assertEquals()`
`assertFalse()`
`assertFileEquals()`
`assertFileExists()`
`assertGreaterThan()`
`assertGreaterThanOrEqual()`
`assertInstanceOf()`
`assertInternalType()`
`assertJsonFileEqualsJsonFile()`
`assertJsonStringEqualsJsonFile()`
`assertJsonStringEqualsJsonString()`
`assertLessThan()`
`assertLessThanOrEqual()`
`assertNull()`
`assertObjectHasAttribute()`
`assertRegExp()`
`assertStringMatchesFormat()`
`assertStringMatchesFormatFile()`
`assertSame()`
`assertSelectCount()`
`assertSelectEquals()`
`assertSelectRegExp()`
`assertStringEndsWith()`
`assertStringEqualsFile()`
`assertStringStartsWith()`
`assertTag()`
`assertThat()`
`assertTrue()`
`assertXmlFileEqualsXmlFile()`
`assertXmlStringEqualsXmlFile()`
`assertXmlStringEqualsXmlString`

Table 11-1. PHPUnit Assertion Methods

Testing Data from HTML Form Submissions

In the restaurant menu application that we developed, recall that we created a form to load menuitem data into the database using the predefined $_POST variable which is an array that collects the values from a form.

Inevitably you will have to work with forms when developing in PHP. It follows that you will have to test form submissions, or more specifically you will need to test GET and POST methods of transferring data in the arrays $_GET and $_POST. Let’s take a look at the idea of dependency injection. It is best practice for you to feed your code what it needs as opposed to it getting the data it needs. Here's an example of the difference:

Example without dependency injection:

```
function AddMenuItem1() {
  foreach($_POST as $form_element => $val) {
    // code to process $val
  }
}

AddMenuItem1();
```

Example with dependency injection:

```
function AddMenuItem2(array &$formData) {
  foreach($formData as $form_element => $val) {
    // code to process $val
  }
}

AddMenuItem2($_POST);
```

See the difference? In your PHPUnit test you can pass `AddMenuItem2()` an associative array of your choice; you've injected the dependency. Whereas `addMenuItem1()` is coupled with `$_POST. $_POST` and `$_GET are` assoc arrays anyways so in your production code you can pass `$_GET` or `$_POST` to your function but in your unit tests you will have to hard code some expected data.

Unit test example:

```
function testAddMenuItem() {
  $fakeFormData = array ('itemname'=>'Ice Tea', 'description'=>'tea with ice', 'servingsize'=>'16 oz',
'price'=>'$1.39');
  AddMenuItem($fakeFormData);
  // assert something...
}
```

Test suites

A test suite is a set of tests that all share the same fixture. The order of the tests shouldn't matter. In the test_assertion.php example above we actually performed two simple tests that shared the same setup() fixture that assigned a value to $x and $y. This is an example of a very simple test suite.

Mocks and Stubs

Stubbing is the practice of replacing an object with a *test double* that returns configured return values.

Mocking is the practice of replacing an object with a *test double* that verifies behavior. Mock objects are used in unit testing to imitate the behavior of real objects in test cases. By using them the functionality of the object you are implementing is easier to test. Mock objects are useful when the real implementation of one or more dependencies of an object has not been realized yet or one ore more of the dependencies of an object depends on factors that are difficult to simulate. An example of both of these cases is testing the interaction between your application and a database. Typically you would probably call on a method or some form of data access object, but what if the database has not been setup yet? Or what if the there was no data available or the code that queries the database hasn't been written yet? A mock data access object simulates the real data access object by returning some pre-defined values. This frees you from the trouble of having to setup the database, load and then look for test data or write the code that queries the database. In the case of an imaginary database connection mocks can test that the query, say SQL, was correctly formed by the object that is using the connection.

Create a mock class

Recall our abstract Menu class from previous chapters:

```
<?php
abstract class Menu
{
        private $menuid;
        private $parentid;
        private $menuitemid;
        private $menuname;
        private $description;

        public function setMenuID($menuid){$this->menuid = $menuid;}
        public function getMenuID(){return $this->menuid;}

        public function setMenuItemID($menuitemid){$this-> menuitemid = $menuitemid;}
        public function getMenuItemID(){return $this-> menuitemid;}

        public function setMenuName($menuname){$this->menuname = $menuname;}
        public function getMenuName(){return $this->menuname;}

        public function setDescription($description){$this->description= $description;}
        public function getDescription(){return $this->description;}

        function getAllMenuItems($menuid)
        {
                $connection = Database::Connect();
                $this->query = "select mitm.*, mi.itemname, mi.description,
                                mi.price, mi.servingsize
                                from `menuitemtomenu` as mitm
                                left join `menuitem` as mi
                                on mitm.menuitemid = mi.menuitemid
                                where mitm.menuid = $menuid";

                $menuitemList = Array();
                $cursor = Database::Reader($this->query, $connection);
                while ($row = Database::Read($cursor))
                {
                        $menuitem = new MenuItem;
                        $menuitem->menuitemid = $row['menuitemid'];
                        $menuitem->itemname = $row['itemname'];
                        $menuitem->price = $row['price'];
                        $menuitem->description = $row['description'];
                        $menuitemList[] = $menuitem;
                }
                return $menuitemList;
        }

}
?>
```

Let's use a stub to simulate our Menu class in a test class called MenuTest():

```
<?php

require_once('/path/to/MenuModel.php');

class MenuTest extends PHPUnit_Framework_TestCase
{
    public function testConcreteMethod()
    {
        $stub = $this->getMockForAbstractClass('Menu');

         $result = array("menuitemid"=>"34",
                "itemname"=>"Nachos",
                "price"=>"$5.99",
                "description"=>"Ground Steak with peppers and onions layered with 5 different cheeses")));

        $stub->expects($this->any())
             ->method('getAllMenuItems')
             ->will($this->returnValue($result));

        $this->assertNotNull($stub->getAllMenuItems('3'));
    }
}
?>
```

In the example above I called the `PHPUnit_Framework_Testcase` method `getMock()`, passing it "Menu" which is the name of the class I wish to mock. This method dynamically generates a class and instantiates an object from it. I then store this mock object in $stub. We then override the `getAllMenuItems` method. These few lines are the real key:

```
<?php
$stub->expects($this->any())
               ->method(' getAllMenuItems ')
               ->will($this->returnValue($results));
?>
```

These three lines, referred to as the *fluent interface*, tell the stub object that anytime the `getAllMenuItems` method is called on it, it should go ahead and return the value in `$result` instead. Mock (and Stub) objects automatically generated by PHPUnit have an `expect()` method as you can see above. This method requires a matcher object which defines the number of times a method should be called, or the cardinality of the expectation. In the example above the `any()` convenience method was used which tells the `expect()` method that the match fails unless zero or more calls are made to the corresponding method. This is useful for stub objects that return values but don't test invocations.

Here are some TestCase Matcher methods:

TestCase Method	**Description**
`any()`	Zero or more calls are made to the corresponding method
`never()`	No calls are made to the corresponding method
`atLeastOnce()`	One or more calls are made to the corresponding method
`Once()`	Only one call is made to the corresponding method
`exactly($num)`	$num calls are made to the corresponding method
`at($num)`	A call made to the corresponding at $num index

Table 11-2. *TestCase Matcher Methods*

Let's revisit our refactored MenuItem class from Chapter 6:

```
class MenuItem
{
        // constructor (not implemented)
        public function _construct(){}

        // set undeclared property
        function __set($property, $value)
        {
        $this->$property = $value;
        }

        // get defined property
        function __get($property)
        {
                if (isset($this->$property))
                {
                        return $this->$property;
                }
        }
}
```

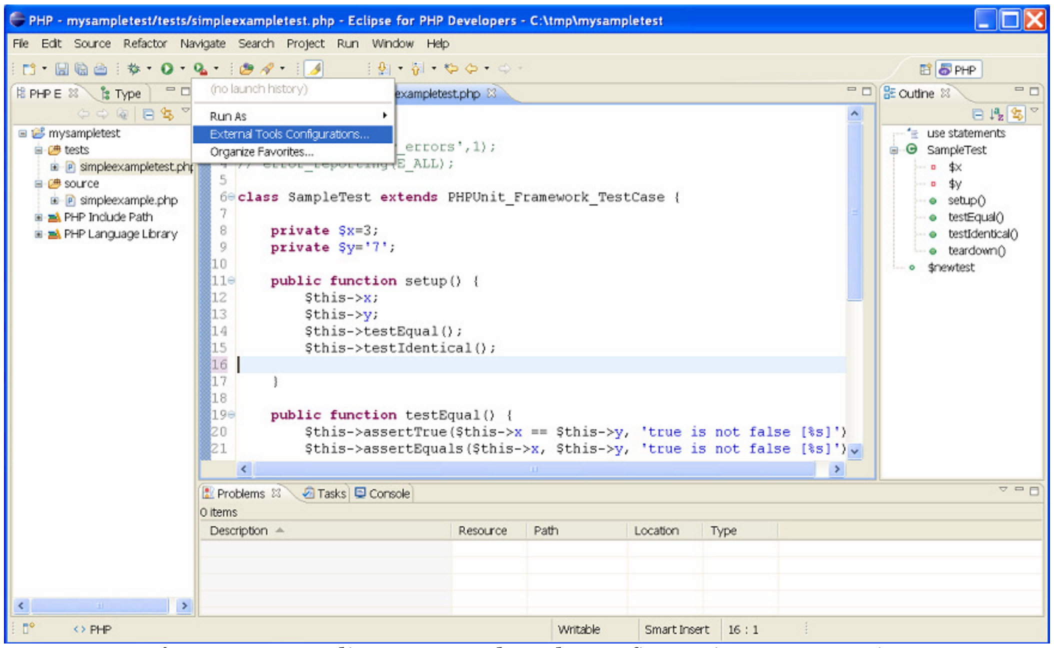
Now let's set up a test class for it to mock it's behavior:

```
<?php
class MenuItemTest extends PHPUnit_Framework_TestCase
{
  public function testPHPUnitMock()
  {
    $mock = $this->getMock('MenuItem');

    //NOTE: this uses at() instead of once()
    $mock->expects($this->at(0))
         ->method('__set')
          ->with('itemname', 'Cajun Curly Fries');

    //NOTE: this uses at() instead of once()
    $mock->expects($this->at(1))
         ->method('__set')
          ->with('price', '$2.99');

    $mock->__set('itemname', 'Cajun Curly Fries');
    $mock->__set('price', '$2.99');
    }
}
?>
```

Running PHPUnit from Eclipse Without Plug-ins

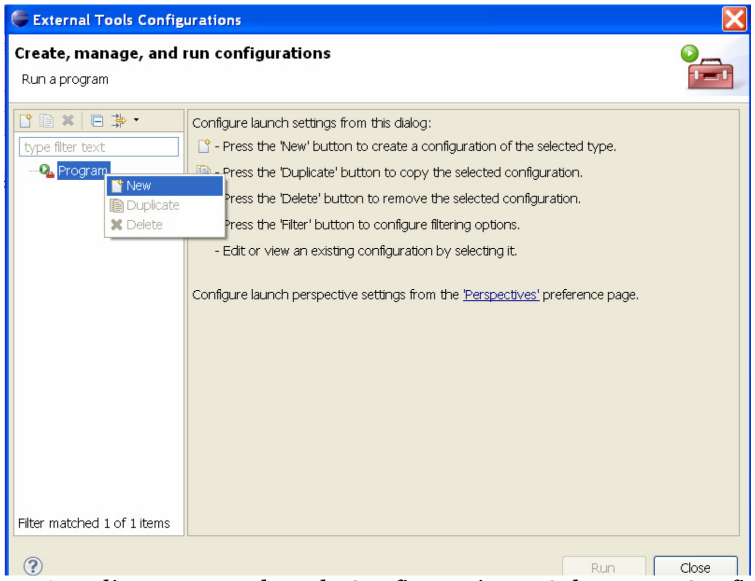
PHPUnit can be run from Eclipse as an alternative to running it from the command-line. Many developers have difficulty in setting up PHPUnit on Eclipse using plug-ins. This section will provide you with some quick and simple instructions on how to run PHPUnit from Eclipse without the hassle of using plug-ins.

First of all, open Eclipse. We will set up a unit test for our restaurant menu project. All of our code for the restaurant project is in a folder called “source” in our project workspace. Similarly, we need to create a folder for our unit tests so let’s create a folder called “test” in our project workspace.



```
_errors',1);
E_ALL);

class SampleTest extends PHPUnit_Framework_TestCase {

    private $x=3;
    private $y='7';

    public function setup() {
        $this->x;
        $this->y;
        $this->testEqual();
        $this->testIdentical();

    }

    public function testEqual() {
        $this->assertTrue($this->x == $this->y, 'true is not false [%s]')
        $this->assertEquals($this->x, $this->y, 'true is not false [%s]')
```

Figure 11-5. *Eclipse External Tools Configuration Menu Option*



Create, manage, and run configurations

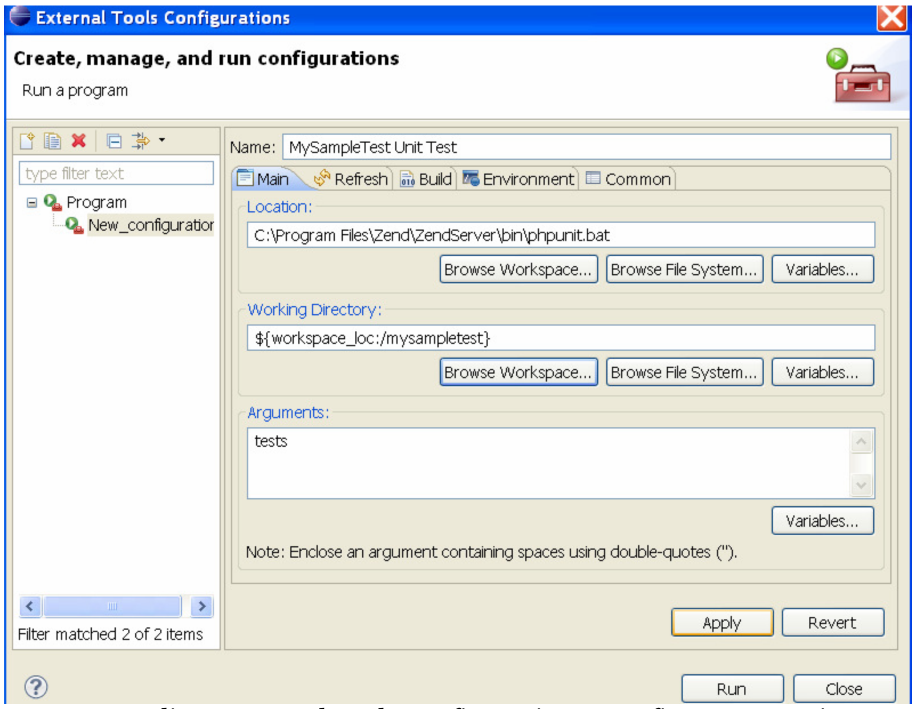
Run a program

Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

Figure 11-6. *Eclipse External Tools Configuration – Select New Configuration*


External Tools Configurations

Create, manage, and run configurations

Run a program

Name: MySampleTest Unit Test

Location: C:\Program Files\Zend\ZendServer\bin\phpunit.bat

Working Directory: ${workspace_loc:/mysampletest}

Arguments: tests

Note: Enclose an argument containing spaces using double-quotes (").

Figure 11-7. *Eclipse External Tools Configuration – Configure PHPUnit Test Tool*

On Macs the path to the phpunit executable is usually /usr/local/bin/phpunit.exe.

The goal of this chapter is to get you acquainted with PHPUnit and some of it's important features. For more detailed information please refer to the official PHPUnit documentation at: **http://www.phpunit.de/manual/current/en/phpunit-book.pdf** . French and Japanese versions of this book are also available at **www.phpunit.de**.

Chapter 11
Source Code Management with Subversion

An important element of the modern software development process is source control management (or version control). Cooperating developers commit their changes incrementally to a common source repository, which allows them to collaborate on code without resorting to crude file-sharing techniques (shared drives, email, ftp, etc). Source control tools track all prior versions of all files, allowing developers to look backward and forward in their software to determine when and where bugs are introduced. These tools also identify conflicting simultaneous modifications made by two poorly-communicating team members, forcing them to work out the correct solution rather than blindly overwriting one or the other original submission.

Every developer who has been under the pressure of meeting a deadline and been in a rush to make changes to their code has likely experienced the confusion of breaking their code and scrambling to get it back to it's last known working state. Without version control this kind of problem can haunt a developer more and more as the application gets bigger and bigger especially when all the "little" details start getting added in and piling up which take up so much time and rapidly grow the code base. This problem only gets worse when multiple developers are working on the same project that has no version control or source control and begin stepping on each other's toes. Nothing is worse then opening a source file and feeling dumbfounded and confused as to why all the hard work you did the day before is not there anymore and the code is much different than it was the last time you touched it and you are in the dark as to what has happened.

The purpose of Subversion (often abbreviated SVN) is to address this problem by providing a relatively simple yet practical and free service that allows multiple developers to check out their own copies a common code base from a central repository. The Subversion repository is a central database which contains all of a project's version controlled files along with their complete history. The repository resides on the machine that hosts the Subversion server. The Subversion server supplies content to the Subversion clients (i.e.: TortoiseSVN) upon request.

This chapter will introduce you to Subversion and show you the basic command-line tools (for Unix, Mac, and Windows), and some excellent graphical tools (e.g. the Subclipse plugin for the cross-platform Eclipse IDE; the TortoiseSVN extension for the Windows graphical shell) needed to perform the essential tasks in Subversion. A general synopsis of these essential tasks and SVN usage include:

- **Acquiring and setting up SVN**
- **Creating or Importing a Project**
- **Managing the fundamental Subversion lifecycle** (which is the following):
  1. **Check out** a project (a directory path) from a repository.
  2. In that project directory, **create or edit** files and subdirectories.
  3. **Update** your local copy from the repository, picking up changes your team members may have made since your last update.
  4. *Go to step 2*. If you're ready to commit your changes, *go to step 5*.
  5. **Commit** your changes to the repository. *Go to step 2*.
- **Adding and Removing Files and Directories**
- **Exporting a Release**
- **Branching a Project**

This chapter will elaborate on how to apply these six aspects to Windows, Mac OS, and Unix/Linux systems.

Setting Up a Subversion Server and Client on Windows

The first thing we'll do is download the latest Subversion Windows binary installer from: **http://subversion.apache.org/packages.html**

Or directly from the mirror site at: **http://sourceforge.net/projects/win32svn/**

In the Installer dialogue box, set the installation path to:

C:\SVN\

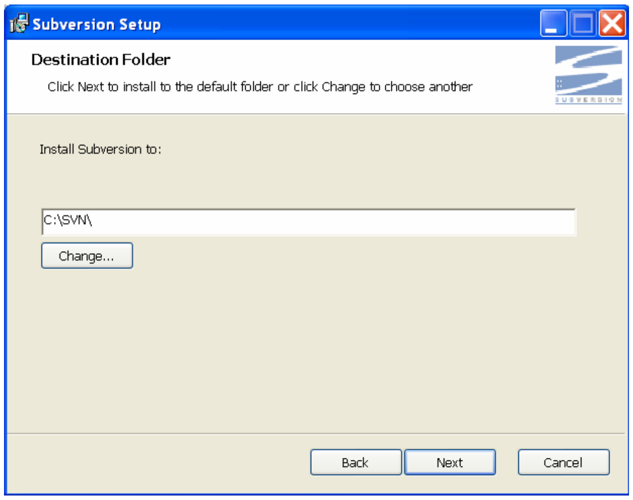


Figure 12-2. *Subversion Setup on Windows – Destination Folder Dialog*

Note that the installer adds C:\SVN\bin to your path, so you can launch a command prompt and start working with it immediately.

Let's create our first source repository, which is effectively a system path.

```
svnadmin create "C:\SVN\repository"
```

Within that newly created folder, uncomment the following lines in the **conf/svnserve.conf** file by removing the pound character from the start of each line:

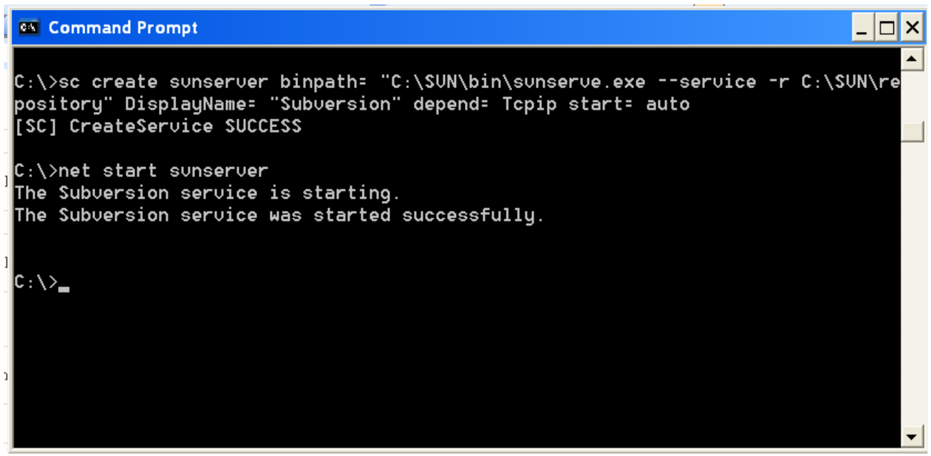
anon-access = none
auth-access = write
password-db = passwd

Next, add some users to the **conf/passwd** file. You can uncomment the default harry and sally users to play with, or add your own:

harry = harryssecret
sally = sallyssecret

As of Subversion 1.4, you can easily **install Subversion as a Windows service**, so it's always available. Just issue the following command:

```
sc create svnserver binpath= "C:\SVN\bin\svnserve.exe --service -r c:\SVN\repository"
displayname= "Subversion" depend= Tcpip start= auto
```



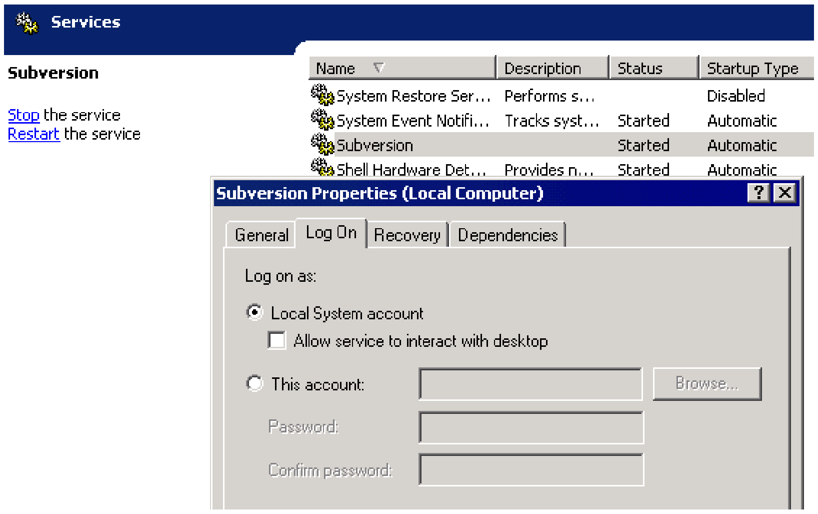
```
C:\>sc create svnserver binpath= "C:\SVN\bin\svnserve.exe --service -r C:\SVN\re
pository" DisplayName= "Subversion" depend= Tcpip start= auto
[SC] CreateService SUCCESS

C:\>net start svnserver
The Subversion service is starting.
The Subversion service was started successfully.


C:\>_
```

Figure 12-3. *Create SVN bin path and start SVN service on Windows*

It's set to auto-start so it will start up automatically when the server is rebooted. Use the following command to start the service without re-booting:

```
net start svnserver
```

Note that the service is running under the **Local System account**. Normally, this is OK, but if you plan to implement any Subversion hook scripts later, you may want to switch the service identity to an Administrator account with more permissions. This is easy enough to do through the traditional Windows services GUI.



Name	Description	Status	Startup Type
System Restore Ser...	Performs s...		Disabled
System Event Notifi...	Tracks syst...	Started	Automatic
Subversion		Started	Automatic
Shell Hardware Det...	Provides n...	Started	Automatic

Log on as:

[✓ Local System account]

☐ Allow service to interact with desktop

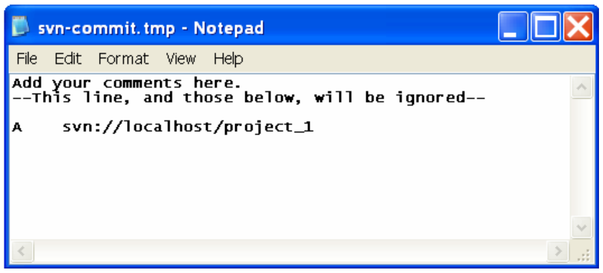
○ This account:

Figure 12-4. *Windows Services Control Panel*

Now let's verify that things are working locally by adding a root-level folder in source control for our new project, named **project_1**.

```
set SVN_EDITOR=C:\windows\system32\notepad.exe
svn mkdir svn://localhost/project_1
```

Now, Subversion will pop up a copy of Notepad with a place for us to enter commit comments.



```
Add your comments here.
--This line, and those below, will be ignored--

A    svn://localhost/project_1
```

Figure 12-5. *svn-commit.tmp File – Add Comments*

Enter whatever comment you like, then save and close Notepad. You'll be prompted for credentials at this point; ignore the prompt for Administrator credentials and press enter. Use the credentials you set up earlier in the conf/passwd file.

svn mkdir svn://localhost/myproject
Authentication realm: <svn://localhost:3690>
Password for 'Administrator': [enter]
Authentication realm: <svn://localhost:3690>
Username: harry
Password for 'harry': ************

Committed revision 1.

We specified **svn://** as the prefix to our source control path, which means we're using the native Subversion protocol. The **Subversion protocol operates on TCP port 3690**, so be sure to open an appropriate hole in your server's firewall, otherwise clients won't be able to connect.

You now have a working Subversion server running on your local Windows machine. Now let's set up the client interface.

Adding Your Source Code the First Time

The next step after installing subversion on your system is to add your source code to the repository. Use the following command to import an existing project:

```
svn import C://projects/trunk/proj_restaurant file:///usr/local/svnrepos/trunk -m
"initial import"
```

The "`-m`" flag is used to post a comment along with the svn command.

Creating a Copy to Work On

Now that your code is checked into the repository, the next step is to create a copy of the code for you to make changes to, this is called the working copy. This is done using the `svn checkout` command.

```
svn checkout file:///usr/local/svnrepos/trunk C://workingcopyof_myproject
```

Updating and Committing Your Working Copy

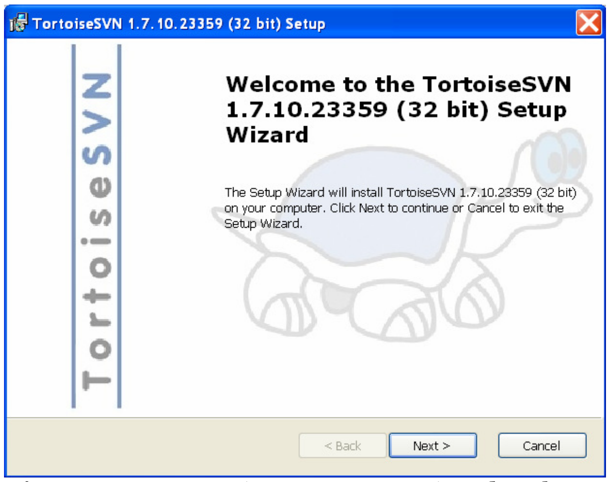
Now use the working copy you just created to make your changes, build, and test. You do not have to notify Subversion about the changes that you make to the files. It will detect this automatically. However, if you `add`, `copy`, `move`, or `remove` any versioned files then you will need to use the corresponding `svn add`, `copy`, `move`, or `remove` command.

Once you have made changes to your working copy and verified that everything is working correctly, you will need to write your changes to the repository. You use the `svn commit` command to do this. This command writes the changes in your working copy back to the repository.

```
commit -m 'published changes made to my working copy'
```

Setting up TortoiseSVN

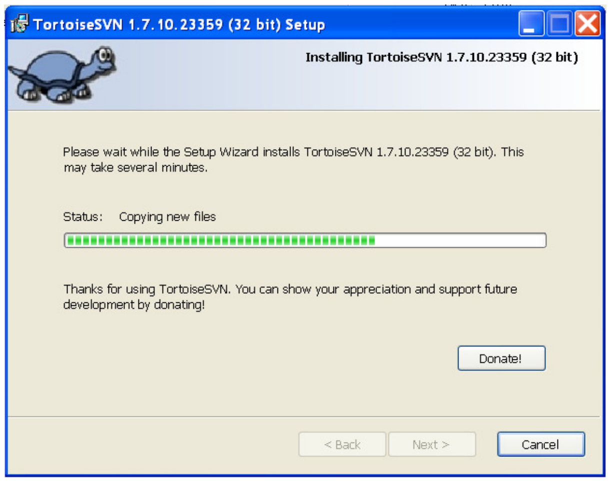
Many developers use a shell extension called **TortoiseSVN** to interact with Subversion instead of using the Unix style commands. Download the latest 32-bit or 64-bit Windows client from **http://tortoisesvn.net/downloads.html** and install it. The installer will tell you to reboot, but you don't have to.


TortoiseSVN 1.7.10.23359 (32 bit) Setup

Welcome to the TortoiseSVN 1.7.10.23359 (32 bit) Setup Wizard

The Setup Wizard will install TortoiseSVN 1.7.10.23359 (32 bit) on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Figure 12-6. *TortoiseSVN Setup Wizard Welcome Page*


TortoiseSVN 1.7.10.23359 (32 bit) Setup

Installing TortoiseSVN 1.7.10.23359 (32 bit)

Please wait while the Setup Wizard installs TortoiseSVN 1.7.10.23359 (32 bit). This may take several minutes.

Status: Copying new files

Thanks for using TortoiseSVN. You can show your appreciation and support future development by donating!

Figure 12-7. *TortoiseSVN Setup Wizard Status Page*

Now create a project folder somewhere on your drive. I used **C:\project_1**. Since TortoiseSVN is a shell extension, to interact with it, you right click in Explorer. Once you've created the project folder, right click in it and select "SVN Checkout..."

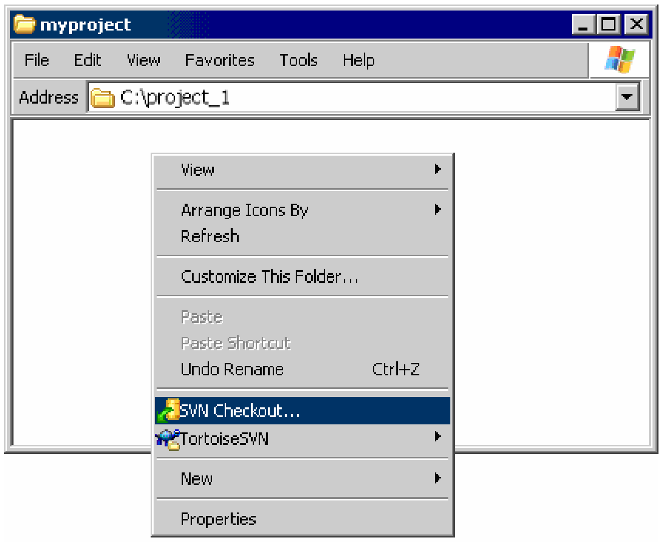
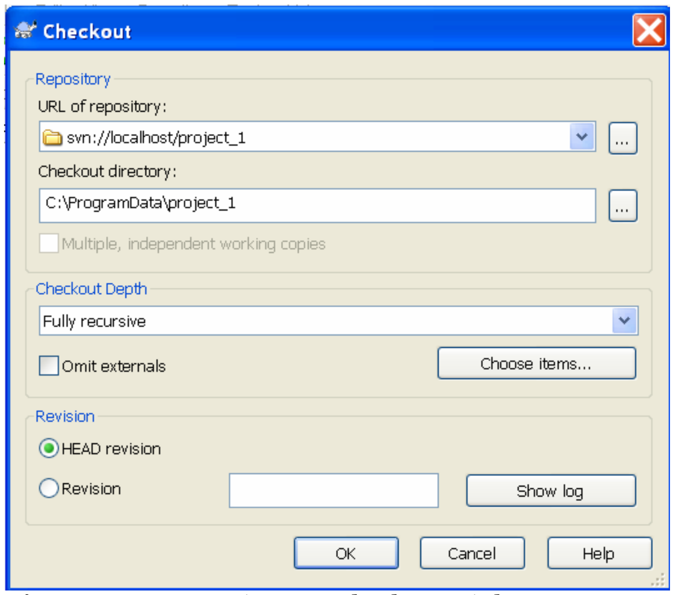


Figure 12-8. *Windows Explorer – Right-click Menu Options –> SVN Checkout*

Type **svn://localhost/project_1/** for the repository URL and click OK.



Repository

URL of repository: svn://localhost/project_1

Checkout directory: C:\ProgramData\project_1

Multiple, independent working copies

Checkout Depth

Fully recursive

Omit externals

Revision

HEAD revision [✓]

Revision

Figure 12-9. *TortoiseSVN Checkout Dialog*

Tortoise now associates the c:\project_1 folder with the svn://localhost/project_1 path in source control. Just about anything you do on your local file system path can be checked back in to source control.

There's a standard convention in Subversion to start with the "TTB folders" at the root of any project, **Trunk, Tags & Branches** folders.

Because Subversion uses regular directory copies for branching and tagging, the Subversion community recommends that you choose a repository location for each project root -- the "top-most" directory which contains data related to that project -- and then create three subdirectories beneath that root: **trunk**, meaning the directory under which the main project development occurs; **branches,** which is a directory in which to create various named branches of the main development line; **tags**, which is a collection of tree snapshots that are created, and perhaps destroyed, but never changed.

Note that we can batch up as many changes as we want and check them all in atomically as one unit. Once we're done, right click the folder and select "SVN Commit..."

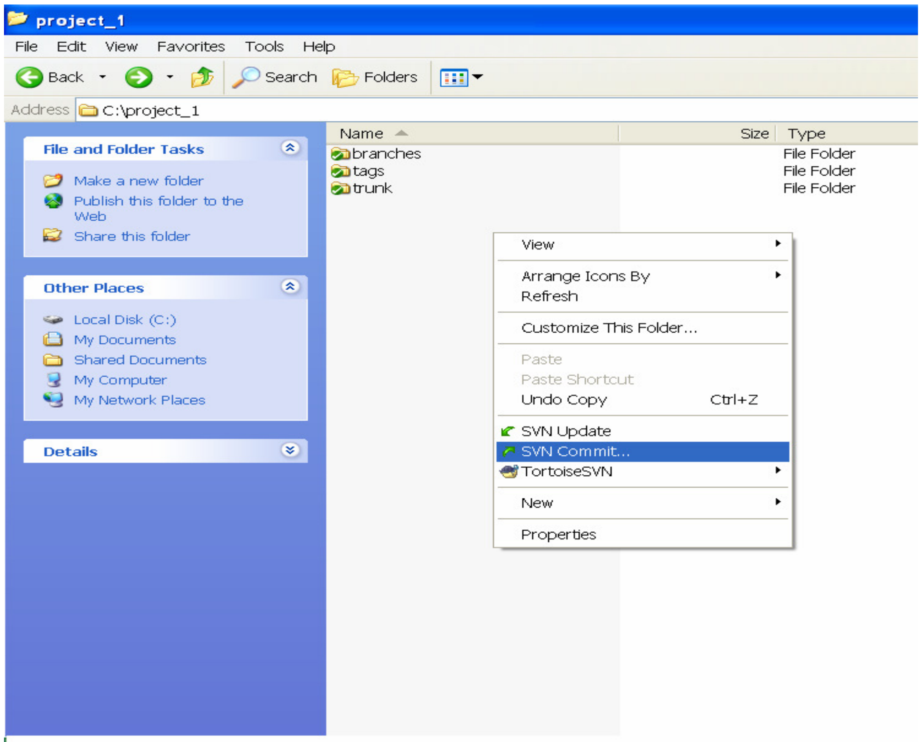
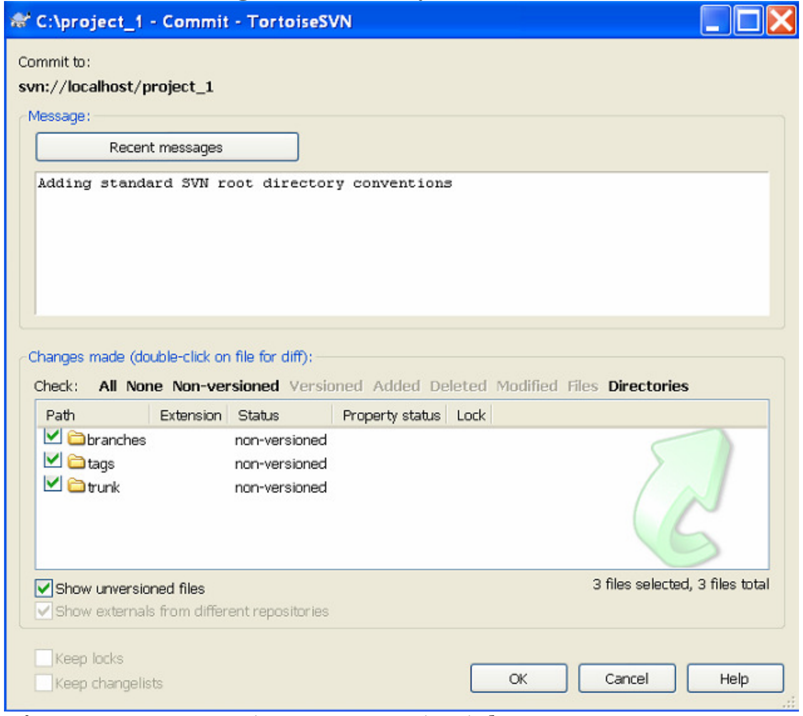


Figure 12-10. *Windows Explorer Right-click Menu Options -> SVN Commit*

In the commit dialog, indicate that yes, we do want to check in these files, and add a comment.


C:\project_1 - Commit - TortoiseSVN

Commit to:
svn://localhost/project_1

Message:

Recent messages

```
Adding standard SVN root directory conventions
```

Changes made (double-click on file for diff):

Check: **All** **None** **Non-versioned** Versioned Added Deleted Modified Files **Directories**

Path	Extension	Status	Property status	Lock
✓ branches		non-versioned		
✓ tags		non-versioned		
✓ trunk		non-versioned		

✓ Show unversioned files — 3 files selected, 3 files total

✓ Show externals from different repositories

Keep locks

Keep changelists

Figure 12-11. *TortoiseSVN Commit Dialog*

You'll have to enter your server credentials here, but Tortoise will offer to conveniently cache them for you. Once the commit completes, note that the files show up in the shell with source control icon overlays:

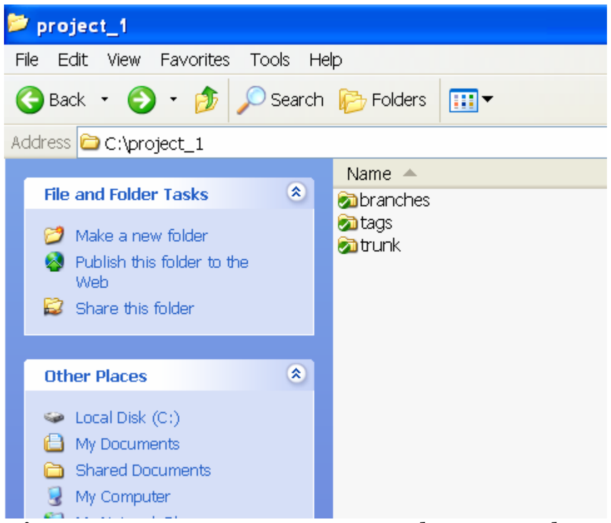
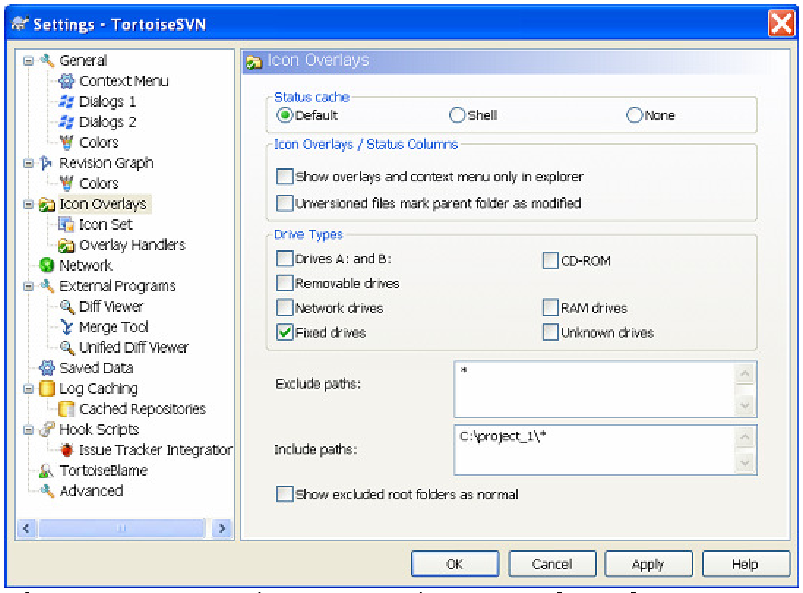


Figure 12-12. *SVN Source Control Icon Overlays*

Now, right click and select "TortoiseSVN, Settings".

<u>NOTE</u>: Tortoise will attempt to apply source control overlays across every single folder and drive on your system. This can lead to some frustrating file locking problems. To instruct Tortoise to apply it's shell commands on specific folders, set this restriction via "Icon Overlays"; look for the exclude and include paths. In this example I set the exclude path to everything, and the include path to only my project folder(s).



Icon Overlays

Status cache: [✓ Default] Shell None

Icon Overlays / Status Columns
- [ ] Show overlays and context menu only in explorer
- [ ] Unversioned files mark parent folder as modified

Drive Types
- [ ] Drives A: and B:
- [ ] CD-ROM
- [ ] Removable drives
- [ ] Network drives
- [ ] RAM drives
- [✓] Fixed drives
- [ ] Unknown drives

Exclude paths: *

Include paths: C:\project_1\*

- [ ] Show excluded root folders as normal

Figure 12-13. *TortoiseSVN – Settings Control Panel*

Be aware, since Tortoise is a shell extension, setting changes may mean you need to reboot.

Adding a new file is simple with TortoiseSVN. For example, open Notepad (or your preferred text editor) and create a file called Employee.php. Type in the following in the new file:

```
<?php

Class Employee { }

?>
```

Now save Employee.php to C://project_1 and close Notepad. Using Windows Explorer, open the C://projects_1 folder and right click on the Employee.php file and select "Add" in the TortoiseSVN menu.

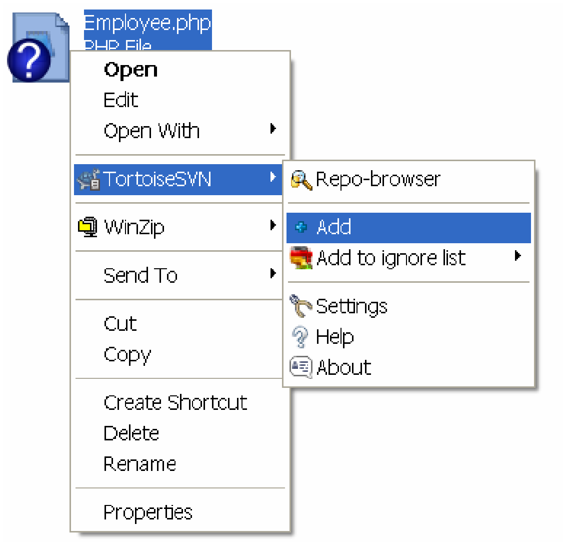
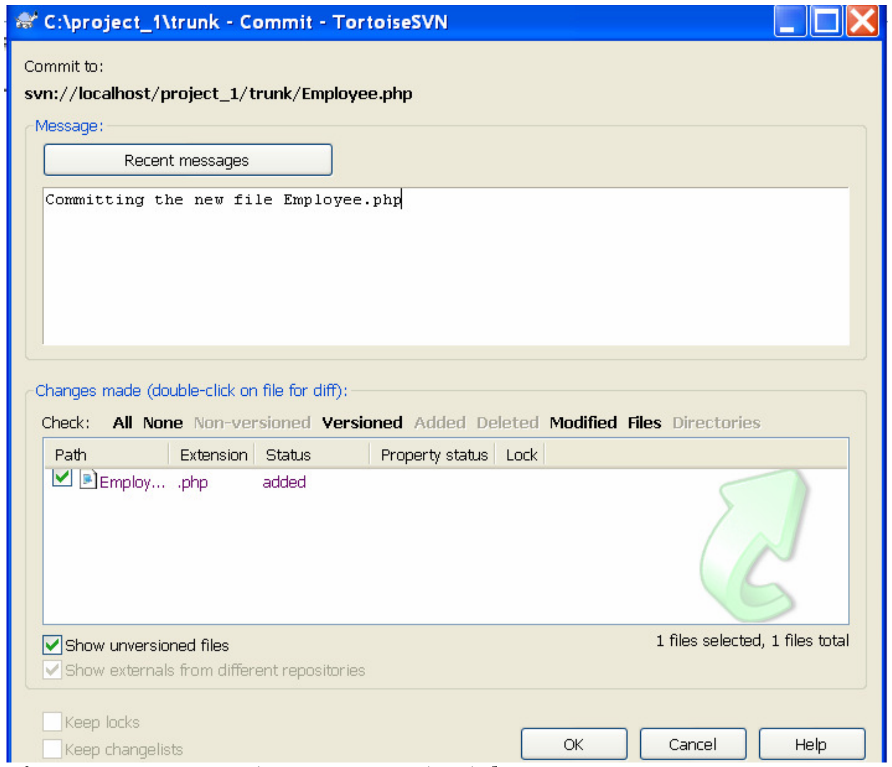


Figure 12-14. *Windows Explorer Right-click Menu Options->TortoiseSVN->Add*

Type some comments in the comment field of the dialog that pops up next and click "Ok"



Commit to:
svn://localhost/project_1/trunk/Employee.php

Message:

```
Committing the new file Employee.php
```

Changes made (double-click on file for diff):

Check: All None Non-versioned Versioned Added Deleted Modified Files Directories

Path	Extension	Status	Property status	Lock
✓ Employ...	.php	added		

[✓] Show unversioned files
[✓] Show externals from different repositories

1 files selected, 1 files total

[ ] Keep locks
[ ] Keep changelists

Figure 12-15. *TortoiseSVN Commit Dialog*

And that's it, you have just added the new file Employee.php to the SVN repository.

Exporting a Release

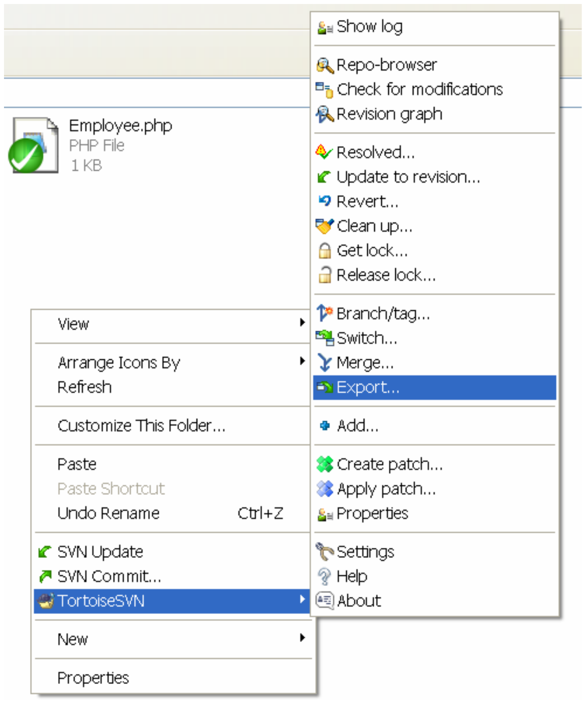


Figure 12-16. *Windows Explorer Right-click Menu Options - TortoiseSVN->Export Option*

Tagging & Branching a File

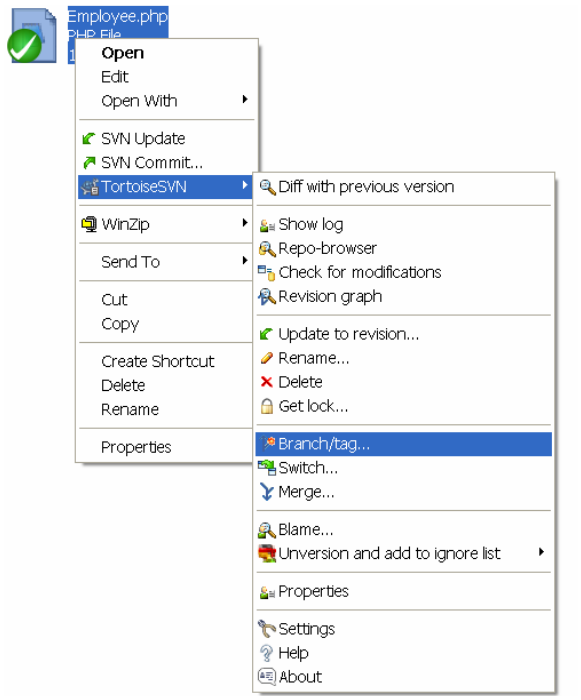


Figure 12-17. *Windows Explorer Right-click Menu Options - TortoiseSVN->Branching Option*

Setting Up a Subversion Server and Client on Mac OS X

Download Subversion

Download and install the Subversion client from **http://subversion.apache.org/packages.html** **.** Simply scroll down to the **Mac OS X** Section and select your package management system (**Fink, MacPorts, openCollabNet, WANdisco**).

Unpack and Install Subversion

After the download, unpack the .dmg file. just click the .pkg-installer and install Subversion. Subversion itself doesn't feature a graphical user interface, so you won't find any new files in your application directory after installation. Instead it installs some command line commands into the directory **:/usr/local/bin** on your hard drive.

Add Subversion to Your Path

Open the Terminal, Located in /Application/Utilities and type: svn [enter] If you have an output like: Type 'svn help' for usage after the dollar sign (**$**). Then svn is working correctly. To be able to call the Subversion commands from every directory, you must add it to your path. If you don't know what that means, don't worry. Just follow the instructions.

Start by creating a new text file called '.profile', i.e. with the command line text editor pico:

```
$ pico .profile
```

Add the following line to the text file:

```
$ export PATH=$PATH:/usr/local/bin
```

Now hit Control-X, then confirm saving the file with 'y', followed by return. You have just added Subversions's location to your path. Let Terminal read this file to know the path has changed (there's an empty space between the dots):

```
$ . .profile
```

Then open another Terminal window and try again with: svn [enter]

Setting up the Subversion Repository

First we will need to set up a Subversion repository on our local computer. That is the place to store all versions of our project. Now create your repository like this:

```
$ svnadmin create svnrepos
```

This will create a repository named ' svnrepos ' in your home directory, although svnadmin won't give you any feedback. Check the existence of this folder by typing 'ls' in the terminal or looking for it in the Finder.

Creating a Subversion Project

Next we create our sample project. Create a new folder and an empty file named 'Employee.php' inside this folder:

```
$ mkdir proj_restaurant
$ touch proj_restaurant/Employee.php
```

Let's import this project into the repository. Type in your terminal, by replacing ' your_user_name ' with your own user name:

```
$ svn import proj_restaurant file:///Users/your_user_name/svnrepos/proj_restaurant -m
"Initial import"
```

output:

```
Adding Employee.php
Committed revision 1.
```

We have just imported the folder 'proj_restaurant' into the repository 'svnrepos'. Each version in the repository is called a "revision" and is identified by a unique number. Always provide a short message ('-m') of the changes you made to the repository.

Checking Out Files from Subversion

Now let’s retrieve, or “check out” a copy of our file from the repository to work on:

```
$ svn checkout file:///Users/your_user_name/svnrepos/proj_restaurant proj_restaurant-copy
```

The output will show all files added ('A') to our working copy:

```
A     proj_restaurant-copy/Employee.php
Checked out revision 1.
```

Change into the working copy directory by typing:

```
$ cd proj_restaurant-copy
```

Next check the content of our working copy in the terminal:

```
$ ls -a1
```

This will output the directory including hidden files:

```
.
..
.svn
Employee.php
```

Note the hidden directory '.svn'. This holds some subversion metadata like the name of the repository, so you don't have to type that in the future.

Updating and Committing Changes

It is time to make some changes to our file and save it back to the repository. Open 'Employee.php' from the working copy with your favorite text editor and add the following to the file:

```
<?php

Class Employee{}

?>
```

You can then query Subversion to find out the differences between the repository and your copy:

```
$ svn status
```

This will state that our file has been modified ('M'):
```
M Employee.php
```

Now we want to update the repository with the changes we made. This process is called "committing":
```
$ svn commit -m "Added some comments"
```

Output:
```
Sending Employee.php
Transmitting file data .
Committed revision 2.
```

Adding and Removing Files and Directories

Adding a File
Adding a new a new document to Subversion is easy, all you have to do is use the svn add command and then commit it. In this example I create a new file called newfile1.php with the touch command and then simply use the svn add subcommand to add it to the repository.

```
% touch /projects/proj_restaurant/newfile1.php
% svn add /projects/proj_restaurant/newfile1.php
% svn commit -m 'added new file'

Adding            newfile1.php
Transmitting file date ...
Committed revision 3.
```

Removing a File
Removing a file from Subversion is as simple as adding one. To remove a file you use the remove subcommand.

```
% svn remove /projects/proj_restaurant/newfile1.php
% svn commit -m 'removed newfile1.php'

Deleting    newfile1.php
Committed revision 4.
```

Adding a Directory
To be consistent with the standard Subversion directory structure, let's go ahead and set up our SVN directory tree in our repository:

```
$ svn add trunk/
$ svn add tags/
$ svn add branches/
```

Let's also add a dummy directory for the sake of discussion:
```
$ svn add dummy/
```

Removing a Directory
Likewise, you use the `remove` subcommand to remove a directory from Subversion:

```
% svn remove dummy

D     dummy

% svn commit -m 'commit to removing the images directory'
```

Tagging & Exporting a Release

You'll recall that you created three directories for your project : **trunk, branches, and tags**. This is the recommended directory layout for individual projects in Subversion.

The **trunk** directory is where you typically do your work. Create files here, edit them, check them in, and compile them. Think of the trunk as your home directory for the purposes of any given project. In fact, for beginners this is probably the only directory you'll use.

The **branches** and **tags** directories exist when you want to move on to more sophisticated parallel development. For example, branches are copies of the code that evolve independently. On large projects this can be important when multiple parties are making substantially different changes to the code; you probably won't need to use them in this course.

Tags, on the other hand, are more likely to be of use to you. A tag represents a snapshot of your code at a single point in time, it’s simple a copy of your code that allows for a more human friendly naming convention. You might, for example, set a tag on your code at the prototype stage, so you can always conveniently refer back to this version of things.

```
$ cd proj_restaurant/trunk
$ touch Timecard.php
$ pico Timecard.php
$ svn add Timecard.php
A       Timecard.php
$ svn commit
[enter a message in your editor, save and close]
Sending Timecard.php
Transmitting file data ...
Committed revision 3.
[Maybe this is your finished initial release, so let's take a snapshot.]
$ cd ..
$ ls
branches/      tags/          trunk/
[Now we'll use "svn cp" to copy the trunk, as it exists right now,
 to a new spot underneath tags.]
$ svn cp trunk tags/release_1.0.0
A tags/release_1.0.0
$ svn commit trunk
[enter a message along the lines of "Taking a snapshot of the release
 we're turning in."]
Sending tags/release_1.0.0...
Committed revision 4.
$ ls trunk/
Timecard.php     # <-- your working code; continue to develop here
$ ls tags/
release_1.0.0/
$ ls tags/release_1.0.0/
Timecard.php     # <-- this version is frozen in time
```

```
$ svn export file:///Users/your_user_name/svnrepos/proj_restaurant proj_restaurant-copy
```

This copy is now safe to deploy on the web. It is not a working copy though, so you can't commit changes back to the repository from this folder.

Branching & Merging a Project

Creating a Branching

First of all, you are probably wondering what a branch is. Well suppose you build an small point-of-sale application for a store owner. You deliver it to them and they love it, so much that they show it to another business owner who decides that they also want it. So you deliver an exact copy of the initial point-of-sale system that you built for the first store owner but the second business owner asks you to make a few custom modifications and tweaks. The first guy, the store owner occasionally asks for subtle enhancements as well. As time goes on you find yourself maintaining two different applications that at one time been identical.

This is the basic concept of a branch, in a nut shell it is a parallel line of development that exists independently of another line, yet still shares a common history if you look far enough back in time. A branch always begins life as a copy of the trunk within the repository, and moves on from there, generating its own history.

There are a couple of common types of branching which are scenario based. The first can be described as the **developer branch.** In this case it is beneficial to create a branch and develop the new feature within it, later merging it all back to the trunk. A common scenario for this is when you have a large feature to deliver, which may take some time to finish. You can't halt all other development on the project, or commit partial changes, but on the other hand, as you develop the new feature you do want to continue to enjoy the benefits version controlled files and directories, and making regular commits.

The other common type of branch is known as a **release branch**. This is typically done when you are in the last few days before a major release of your project. A release branch is created, and this is where all the last minute tweaks and bug fixes are made. By working in a branch in this way, the rest of the team can continue their day-to-day development work on the trunk, independently of what's happening in the release branch.

Creating a branch is very simple. A branch is simply a copy of the trunk, and can easily be made using the `svn cp` or `svn copy` command, supplying it with two Subversion repository URLs. The first of these will be the address of the resource to be branched, typically the trunk, and the second will be the desired address of the new branch.

```
% svn cp -m 'Making test branch' svn://localhost/usr/local/svnrepos/projects/trunk
svn://localhost/usr/local/svnrepos/projects/branches/branch_1.0.0

Committed revision 12.
```

There it is, your branch created and automatically committed, ready for you to check out and work with. Working with a branch is no different from working with any versioned resource in Subversion.

Merging

Once you've created a branch, you will inevitably come to a point where you will want to integrate the changes in the branch back into the trunk . This is known as merging, and is achieved by use of the `svn merge` command. This is a little more complex than branching, but is still easy enough.

Assume that your endeavoring co-developer Dan has made some extensive changes to Employee.php. To merge all of those changes at once, you'll need a note of those two Subversion URLs we used earlier (the URLs of both the trunk and the branch), and you will also need a working copy of the trunk. Use the `cd` command to navigate into the directory which holds that working copy, and initiate the merge using `svn merge` and the following syntax:

```
% svn merge svn://localhost/usr/local/svnrepos/projects/branches/branch_1.0.0
svn://localhost/usr/local/svnrepos/projects/trunk
U    Employee.php
% svn status
M    Employee.php
```

The above command will take all changes made in the branch_1.0.0 branch and merge them into the trunk. This time, nothing is committed automatically. The resulting changes are actually made to the working copy, so that we can review them first, and make sure that we're happy with them. If so, we can commit the changes with `svn commit` as with any other changes to a working copy.

Another approach to merging is to specify revision numbers. Let's say Dan made some useful changes and committed them as revisions 114 and 115, and that we want those changes to be replicated in the trunk (or in another branch), we can pass those revision numbers to `svn merge` using the -r flag:

```
% svn merge  -r 113:115 svn://localhost/usr/local/svnrepos/projects/branches/branch_1.0.0
U    Employee.php
```

```
% svn status
M    Employee.php
```

Subversion is instructed to take all changes made to `/branches/branch_1.0.0` between revisions 113 and 115, and copy them all into my current working copy. Nothing is committed automatically, and you are free to review the resulting code before committing it yourself.

Graphical User Interfaces

Many people don't like working with the terminal. They find it complicated to remember the text commands, as opposed to clicking on buttons in applications. There is a couple of free or commercial apps available on the internet, that provide a graphical user interface for Subversion commands.

A nice and free GUI for Mac OS X is **svnX**. To manage your working copies from the same application that you write your code with, the text editor **TextMate** is a good choice. **TextMate** includes a Subversion bundle that allows you to easily invoke most Subversion commands from the menu. Only once for setting up the repository and checking out the first working copy, you will have to use the terminal. After that, just press Shift-Control-A to open the Subversion bundle menu.

Setting Up a Subversion Server on a Linux Server

Unix-like operating systems most likely already have a Subversion client installed and ready to use. Verify this by typing the following on the command line (I will be using the csh shell for this discussion):

```
% svn help
```

If you see some usage information come up then you can proceed with setting up your repository. If you do not then you will need to download and install Subversion. Depending on your flavor of Linux or Unix, you should have access to a simple installation program like Yum or Apt. Regardless, it is best practice to refer to your distributions documentation before you proceed.

You can download the Subversion source code and binaries specific to your Linux or Unix system from **http://subversion.apache.org/packages.html** .

Step 1:
In this example I will set up a repository in the directory /usr/local/svnrepos. You may need root privileges to write to this directory.

```
% svadmin create -fs-type fsfs /usr/local/svnrepos
```

This command will not provide any confirmation to the command line but you can easily check to see that it created a directory call svnrepos in the /usr/local directory.

Step 2:
Create your SVN user: Now that your repository is successfully set up, you'll need to create an svn user. Simply open the svnserve.conf file in the editor of your choice:

```
% vi /usr/local/svnrepos/conf/svnserve.conf
```

In that file add these three lines:

```
anon-access = none
auth-access = write
password-db = passwd
```

Step 3:
Now you'll need to create a password file:

```
% vi /usr/local/svnrepos/conf/passwd
```

Add a line in that file for your user in the format =

exampleuser = examplepassword

Step 4:
When you invoke svn commit, Subversion will launch your "default editor" so it can ask you to add some comments to a text file. Let's go ahead and set up a default editor that's easier to use with Subervion than the vi editor such as pico or emacs. I will use emacs in this example:

```
% setenv EDITOR emacs
```

Step 5:
For the sake of simplicity, let's create a bare bones file named Employee.php by opening emacs (or your preferred text editor) and adding the following text:

```
<?php

Class Employee
{
}

?>
```

Save the file as Employee.php in /projects/proj_restaurant/ or whatever directory you want to use as your project directory.

Let's also set up our **trunk, tags, branches** directory structure. This is an important convention used by Subversion and I will explain it's use as we go along. Fundamentally, you will do all of your work in the trunk directory. All three need to be just beneath your project folder so if your project path looked like this:

```
/projects
      /proj_restaurant
```

It needs to be like this:

```
/projects
      /branches
      /tags
      /trunk
            /proj_restaurant
```

So `cd` to `/projects` and create the "TTB" directories:

```
mkdir trunk
mkdir branches
mkdir tags
```

Set up the same directory structure in your repository (we'll discuss adding directories and files a little later):

```
% svn add trunk/
% svn add tags/
% svn add branches/
```

The resulting directory tree will look like this:

```
/svnrepos
      /branches
      /tags
      /trunk
```

Now import your project:
This example assumes you've put your project file in /projects/proj_restaurant/. You can start a project anywhere you like, just make sure the import path in the following command matches the directory path of your project files.

```
% svn import /projects/trunk/proj_restaurant file:///usr/local/svnrepos/trunk
```

Step 6:
Run the svn server as daemon:

```
% svnserve –d
```

Done! You should now have a svn server running with one project named proj_restaurant, it will contain one file called Employee.php.

Step 7:
Try checking it out of the repository:

```
% svn co svn://localhost/usr/local/svnrepos/trunk/proj_restaurant
```

Since we set anon-access to none you should be prompted for username and password which you created in the file /usr/local/svnrepos/conf/passwd.

Updating and Committing Changes

Let's update our Employee.php file by adding some document level comments to the file:

```
<?php
/**
* This file contains the Employee class
**/

class Employee
{
}

?>
```

In order to check to see which files have changed before you incorporate differences locally use the `status` command:

```
% svn status –show-updates
```

The `status` command will display a list of files that an update would affect locally, now let's go `commit` the changes:

```
% svn commit Employee.php
```

Your default editor (pico in this example) will be invoked on a temporary file; into this file you should type some brief comments explaining the purpose of your change.
You'll note that Subversion responds to your check in with a message:

```
Sending       Employee.php
```

```
Transmitting file data ...
Committed revision 2.
```

Each committed change is assigned a monotonically-increasing revision number. (In this example, that number was 2.) You can go back and examine any past check in with svn log by using this command:

```
% svn log -v -r2
```

The -r flag lets you pick the revision to look at; the -v flag tells you what files were touched.

Note: If you happen to invoke the `svn commit` but realize that you didn't really mean to commit all the files you checked out or all the changes you made, exit your editor without adding a commit message. You will be given a chance to back out. Once you write and save a commit message, you've lost that opportunity.

Adding and Removing Files and Directories

Adding a File

Adding a new a new document to Subversion is easy, all you have to do is use the **`svn add`** command and then `commit` it. In this example I create a new file called newfile1.php with the `touch` command and then simply use the **`svn add`** subcommand to add it to the repository.

```
% touch /projects/proj_restaurant/newfile1.php

% svn add /projects/proj_restaurant/newfile1.php

% svn commit -m 'added new file'

Adding        newfile1.php
Transmitting file date ...
Committed revision 3.
```

Removing a File

Removing a file from Subversion is as simple as adding one. To remove a file you use the `remove` subcommand.

```
% svn remove /projects/proj_restaurant/newfile1.php

% svn commit -m 'removed newfile1.php'

Deleting      newfile1.php
Committed revision 4.
```

Adding a Directory

You can also use the add subcommand to add a directory to Subversion:

```
% mkdir images
% touch images/picture1.jpg
% svn add images

A       images
A       images/picture1.jpg
```

As you can see by this simple example, the contents of the images directory are also added to Subversion.

Removing a Directory

Likewise, you use the `remove` subcommand to remove a directory from Subversion:

```
% svn remove images

D      images/picture1.jpg
D      images

% svn commit -m 'commit to removing the images directory'
```

Tagging & Exporting

You'll recall that you created three directories for your project : **trunk, branches, and tags**. This is the recommended directory layout for individual projects in Subversion.

The **trunk** directory is where you typically do your work. Create files here, edit them, check them in, and compile them. Think of the trunk as your home directory for the purposes of any given project. In fact, for beginners this is probably the only directory you'll use.

The **branches** and **tags** directories exist when you want to move on to more sophisticated parallel development. For example, branches are copies of the code that evolve independently. On large projects this can be important when multiple parties are making substantially different changes to the code; you probably won't need to use them in this course.

Tags, on the other hand, are more likely to be of use to you. A tag represents a snapshot of your code at a single point in time, it's simple a copy of your code that allows for a more human friendly naming convention. You might, for example, set a tag on your code at the prototype stage, so you can always conveniently refer back to this version of things.

Allow me to illustrate the use of a tag with an example:

```
% cd /projects/proj_restaurant/trunk
% emacs MyPHPCode.php
% svn add MyPHPCode.php
A      MyPHPCode.php
% svn commit
[enter a message in your editor, save and close]
Sending MyPHPCode.php
Transmitting file data ...
Committed revision 7.
[Maybe this is your finished initial release, so let's take a snapshot.]
% cd ..
% ls
branches/      tags/          trunk/
[Now we'll use "svn cp" to copy the trunk, as it exists right now,
 to a new spot underneath tags.]
% svn cp trunk tags/release_1.0.0
A tags/release_1.0.0
% svn commit trunk
[enter a message along the lines of "Taking a snapshot of the release
 we're turning in."]
Sending tags/release_1.0.0...
Committed revision 8.
% ls trunk/
MyPHPCode.php     # <-- your working code; continue to develop here
% ls tags/
release_1.0.0/
% ls tags/release_1.0.0/
MyPHPCode.php     # <-- this version is frozen in time
```

Now to export a clean release version of your codebase use the `export` subcommand:

```
% svn export svn://localhost/projects/proj_restaurant/tags/release_1.0.0 \ release_1.0.0
```

Branching & Merging a Project

Creating a Branching

First of all, you are probably wondering what a branch is. Well suppose you build an small point-of-sale application for a store owner. You deliver it to them and they love it, so much that they show it to another business owner who decides that they also want it. So you deliver an exact copy of the initial point-of-sale system that you built for the first store owner but the second business owner asks you to make a few custom modifications and tweaks. The first guy, the store owner occasionally asks for subtle enhancements as well. As time goes on you find yourself maintaining two different applications that at one time been identical.

This is the basic concept of a branch, in a nut shell it is a parallel line of development that exists independently of another line, yet still shares a common history if you look far enough back in time. A branch always begins life as a copy of the trunk within the repository, and moves on from there, generating its own history.

There are a couple of common types of branching which are scenario based. The first can be described as the **developer branch.** In this case it is beneficial to create a branch and develop the new feature within it, later merging it all back to the trunk. A common scenario for this is when you have a large feature to deliver, which may take some time to finish. You can't halt all other development on the project, or commit partial changes, but on the other hand, as you develop the new feature you do want to continue to enjoy the benefits version controlled files and directories, and making regular commits.

The other common type of branch is known as a **release branch**. This is typically done when you are in the last few days before a major release of your project. A release branch is created, and this is where all the last minute tweaks and bug fixes are made. By working in a branch in this way, the rest of the team can continue their day-to-day development work on the trunk, independently of what's happening in the release branch.

Creating a branch is very simple. A branch is simply a copy of the trunk, and can easily be made using the `svn cp` or `svn copy` command, supplying it with two Subversion repository URLs. The first of these will be the address of the resource to be branched, typically the trunk, and the second will be the desired address of the new branch.

```
% svn cp -m 'Making test branch' svn://localhost/usr/local/svnrepos/projects/trunk
svn://localhost/usr/local/svnrepos/projects/branches/branch_1.0.0

Committed revision 12.
```

There it is, your branch created and automatically committed, ready for you to check out and work with. Working with a branch is no different from working with any versioned resource in Subversion.

Merging

Once you've created a branch, you will inevitably come to a point where you will want to integrate the changes in the branch back into the trunk . This is known as merging, and is achieved by use of the `svn merge` command. This is a little more complex than branching, but is still easy enough.

Assume that your endeavoring co-developer Dan has made some extensive changes to Employee.php. To merge all of those changes at once, you'll need a note of those two Subversion URLs we used earlier (the URLs of both the trunk and the branch), and you will also need a working copy of the trunk. Use the `cd` command to navigate into the directory which holds that working copy, and initiate the merge using `svn merge` and the following syntax:

```
% svn merge svn://localhost/usr/local/svnrepos/projects/branches/branch_1.0.0
svn://localhost/usr/local/svnrepos/projects/trunk
U    Employee.php
% svn status
M    Employee.php
```

The above command will take all changes made in the branch_1.0.0 branch and merge them into the trunk. This time, nothing is committed automatically. The resulting changes are actually made to the working copy, so that we can review them first, and make sure that we're happy with them. If so, we can commit the changes with `svn commit` as with any other changes to a working copy.

Another approach to merging is to specify revision numbers. Let's say Dan made some useful changes and committed them as revisions 114 and 115, and that we want those changes to be replicated in the trunk (or in another branch), we can pass those revision numbers to `svn merge` using the -r flag:

```
% svn merge  -r 113:115 svn://localhost/usr/local/svnrepos/projects/branches/branch_1.0.0
U    Employee.php
% svn status
M    Employee.php
```

Subversion is instructed to take all changes made to **/branches/branch_1.0.0** between revisions 113 and 115, and copy them all into my current working copy. Nothing is committed automatically, and you are free to review the resulting code before committing it yourself.

Summary

With the introduction of PHP 5, object oriented programming became the center of the PHP universe. In Chapters 2 through 7 we covered the key aspects of PHP's object oriented support. We also discussed some important principles of object oriented PHP such as coding to an interface and not to an implementation as well as how important it is to avoid tight coupling of your code. Although we did not dig deep into design patterns we did cover how to organize a custom application according to the Model-View-Controller design pattern. This is the *de facto* way to build PHP web applications these days. In this edition of this book you were introduced to unit testing with PHPUnit and version control with SVN. In the next addition I intend to expand on the use of PHPUnit with the Zend Framework and also introduce Git as a distributed revision control and source code management (SCM) system. Also in this next edition I will demonstrate step-by-step how to take the custom MVC application we developed in Chapter 8 and migrate it into the Zend Framework. We didn’t discuss Object-Relational Mappers in this edition but I intend to cover that topic in the next edition with the introduction to Doctrine 2. A couple of other topics that I missed in this edition but will also include in the next are PHP namespaces and type hinting.

Be aware that PHP is a language that is still evolving and that future enhancements to the PHP language will be beset with new object oriented features. There is no doubt that PHP has come a long way since PHP 3. In fact, of all the web sites that use PHP, over 96% of them use PHP 5 while only 0.1% use PHP3. PHP 5 has made PHP increasingly popular which is why it is important to learn it and stay current with it. According to W3 Tech Web Technology Surveys, PHP is used as the server side language for 78.7% of web applications compared to other languages.

It is reasonable to expect PHP to continue to gain market share. PHP5+ is used as the development language for many popular content management systems such as Drupal, Joomla and WordPress as well as many popular eCommerce platforms such Magento. In the next edition of this book I will also discuss these open source systems with ample examples and discussion on how to use and customize them. In add to that we will build another custom application, a custom image gallery that will show you how to develop controls for pagination using the iterative design pattern.

Appendix A
Bibliography

Web Resources

Eclipse: http://www.eclipse.org

JQuery: http://jquery.com

MySQL: http://www.mysql.com

MySQL Workbench 5.2 CE: http://dev.mysql.com/downloads/workbench/5.2.html

Design Patterns: http:// www.ibm.com/developerworks/library/os-php-designptrns/

PEAR: http://pear.php.net

PHP: http://www.php.net

PHPUnit: http://www.phpunit.de

Sourceforge: http://sourceforge.net/projects/win32svn/

Subversion: http://subversion.apache.org

Templates: http://www.smarty.net

TortoiseSVN: http://tortoisesvn.net/downloads.html

W3 Tech Web Technology Surveys : http://w3techs.com/technologies/details/pl-php/all/all

Zend: http://www.zend.com

Books

Dagfinn Reiersøl with Marcus Baker and Chris Shiflett. ***PHP in Action.*** Greenwich, CT: Manning Publications Co., 2007

Powers, David. PHP ***Object-Oriented Solutions.*** N.Y, N.Y**.** Apress, 2008

Welling**,** Luke and Thompson, Laura. ***PHP and MySQL Web Development, 4th Edition.*** Upper Saddle River, New Jersey: Pearson Education, Inc, 2009

Appendix B

Setting Up Your Development Stack

If you don't have a solution stack already, this section will guide you through the steps needed to set up an AMP stack on your own computer. If you don't know already, the AMP solution is a software bundle that is used as a platform to run PHP/MySQL applications.

AMP is an acronym for:

A: Apache 2+ HTTP Web Server
M: MySQL 5.x Database Engine
P: PHP 5.x

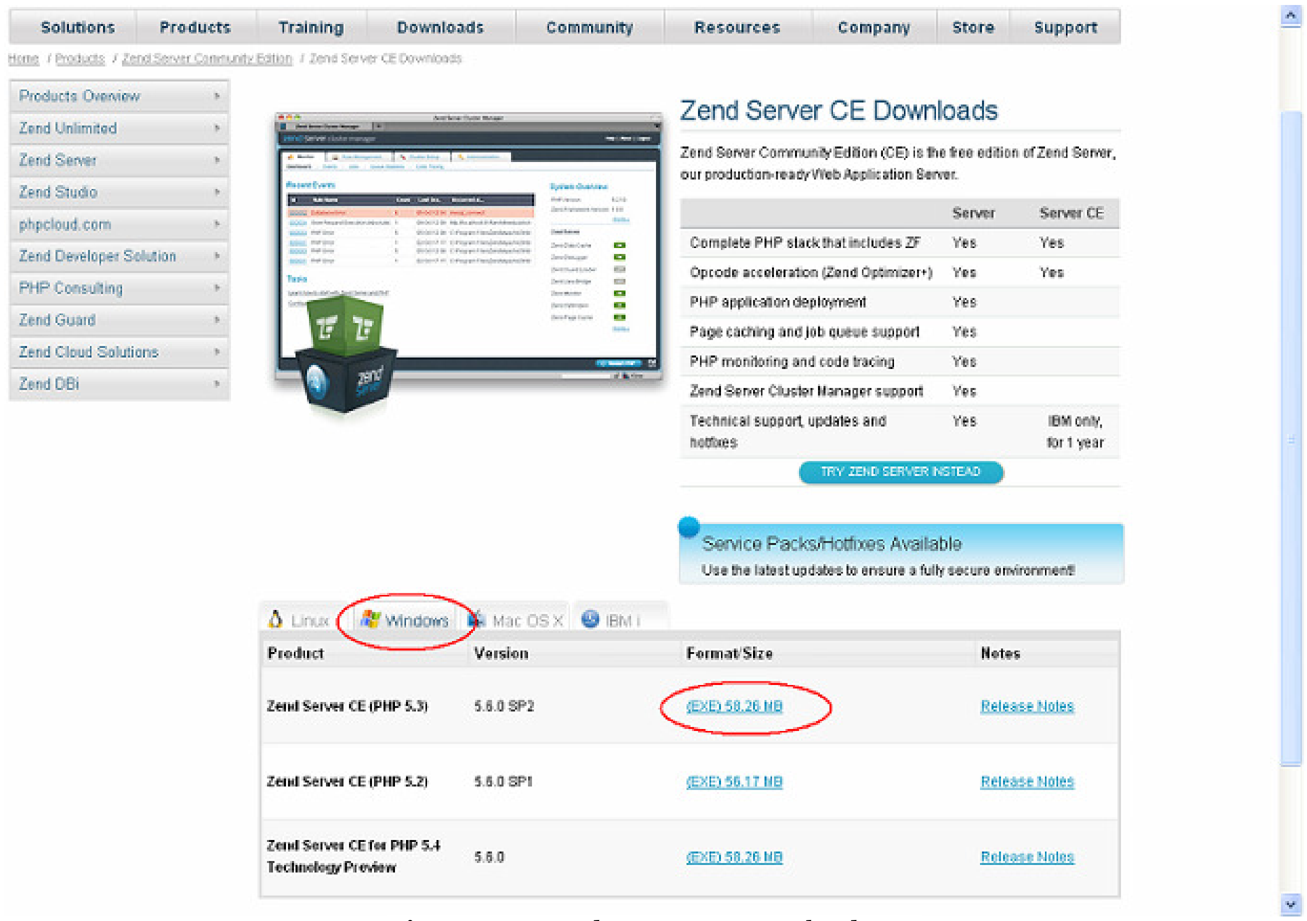
If you load this stack on a Linux platform it is know as a **LAMP** stack, **WAMP** for Windows and **MAMP** for Macs and **XAMP** for cross-platform.

XAMP is an acronym for:

X: cross-platform
A: Apache 2+ HTTP Web Server
M: MySQL 5.x Database Engine
P: PHP 5.x

I am going to go through the steps to load a XAMP stack onto a Windows platform using the free and open source Zend Server Community Edition. ***If you are a Linux (LAMP) or Mac (MAMP) user please select the appropriate tab and proceed with the Zend Server CE (PHP 5.3) download.*** This download will include the Apache web server, MySQL database engine and PHP5.3 library and is available for download at:

http://www.zend.com/products/server-ce/downloads



Zend Server CE Downloads

Zend Server Community Edition (CE) is the free edition of Zend Server, our production-ready Web Application Server.

	Server	Server CE
Complete PHP stack that includes ZF	Yes	Yes
Opcode acceleration (Zend Optimizer+)	Yes	Yes
PHP application deployment	Yes	
Page caching and job queue support	Yes	
PHP monitoring and code tracing	Yes	
Zend Server Cluster Manager support	Yes	
Technical support, updates and hotfixes	Yes	IBM only, for 1 year

Service Packs/Hotfixes Available
Use the latest updates to ensure a fully secure environment!

Product	Version	Format/Size	Notes
Zend Server CE (PHP 5.3)	5.6.0 SP2	(EXE) 58.26 MB	Release Notes
Zend Server CE (PHP 5.2)	5.6.0 SP1	(EXE) 56.17 MB	Release Notes
Zend Server CE for PHP 5.4 Technology Preview	5.6.0	(EXE) 58.26 MB	Release Notes

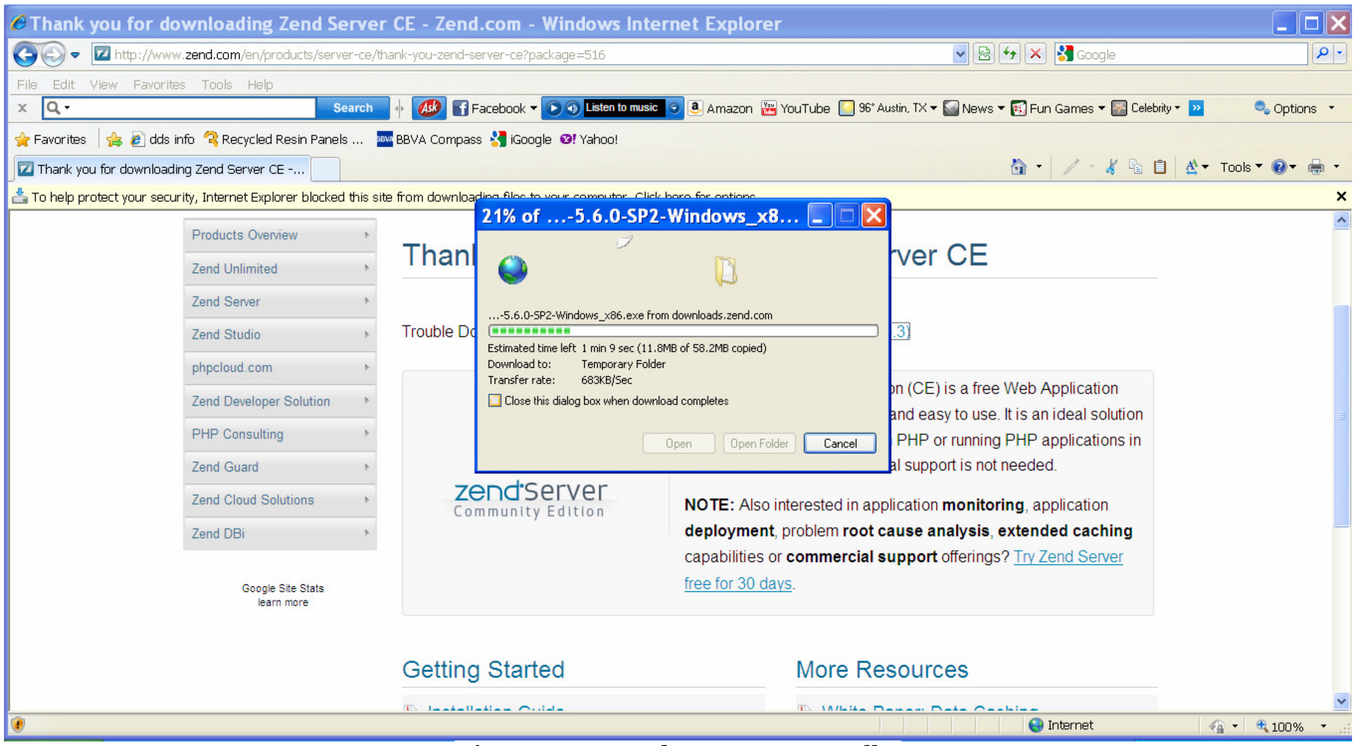
Figure 1-2. *Zend Server CE Download Page*

The full instruction guide is available at http://www.zend.com/en/products/server/resources which includes the following system requirements in section 3:

```
=============================
3. System Requirements
=============================
* Supported Operating Systems, Platforms and OS versions:
  - Linux x86 and x86-64:
    - RHEL 6.x, 5.x, CentOS 5.x,6.x, OEL 5.x and Fedora 13 and above
      through RPM packages
    - SLES 10.x, 11.x and OpenSUSE 11.x, 12.x through RPM packages
    - Debian GNU/Linux 6.0/5.x, Ubuntu Linux 10.04 and above
      through DEB packages
  - Windows x86 and x86-64:
    - Windows XP SP 2 and above
    - Windows Vista (except for "Starter Edition")
    - Windows Server 2003
    - Windows Server 2008
    - Windows 7
  - Mac
    - Apple Mac OS X versions 10.6 and 10.7
```

Install Zend Server CE (Community Edition)

After you have downloaded the Zend Server, run the installer.



To help protect your security, Internet Explorer blocked this site from downloading files to your computer. Click here for options...

21% of ...-5.6.0-SP2-Windows_x8...

...-5.6.0-SP2-Windows_x86.exe from downloads.zend.com

Estimated time left 1 min 9 sec (11.8MB of 58.2MB copied)
Download to: Temporary Folder
Transfer rate: 683KB/Sec

Close this dialog box when download completes

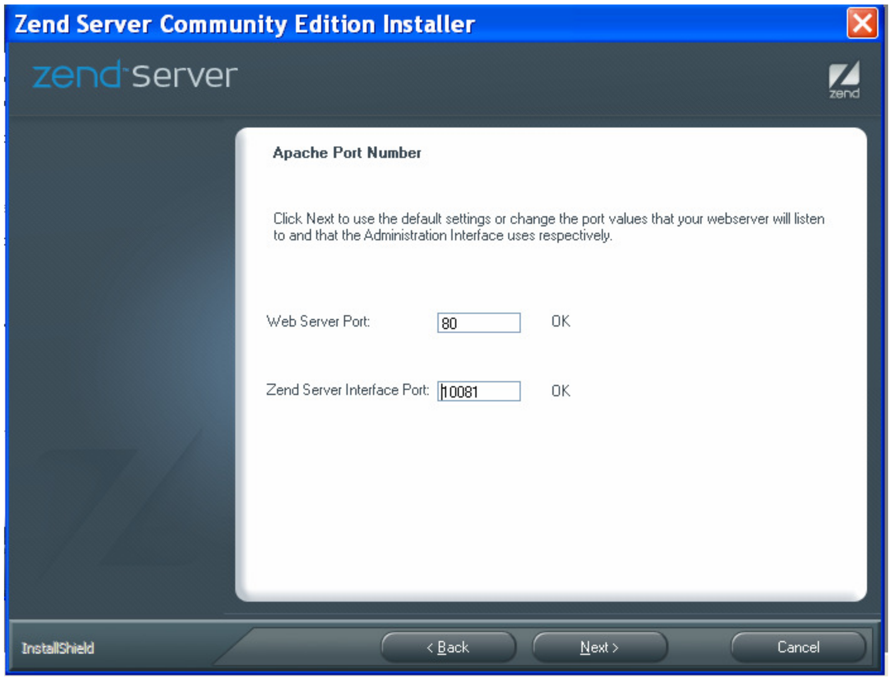
...n (CE) is a free Web Application ... and easy to use. It is an ideal solution ... PHP or running PHP applications in ... al support is not needed.

NOTE: Also interested in application **monitoring**, application **deployment**, problem **root cause analysis**, **extended caching** capabilities or **commercial support** offerings? Try Zend Server free for 30 days.

Figure 1-3. *Zend Server CE Installer*

When prompted for the Apache port number, enter the following:

Web Server Port: **80**
Zend Server Interface Port: **10081**



Apache Port Number

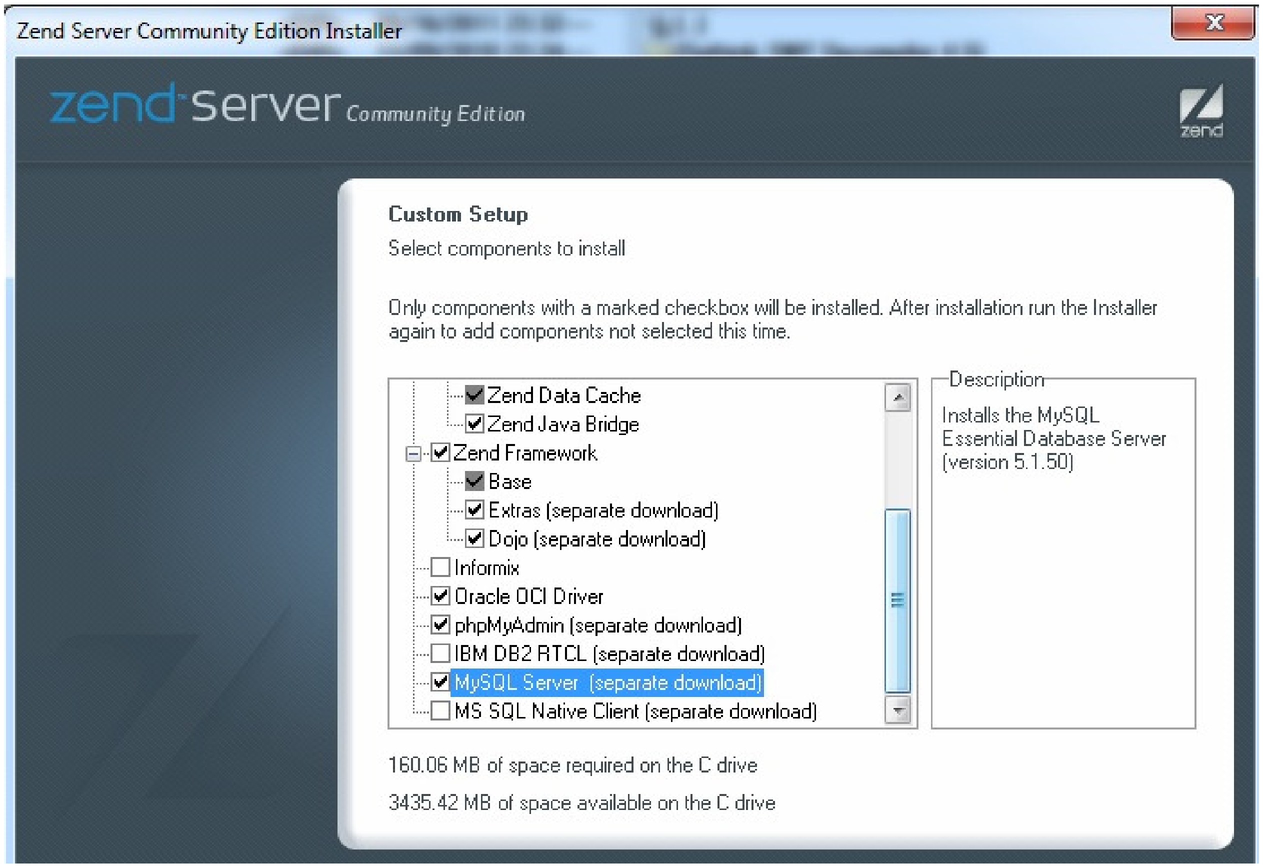
Click Next to use the default settings or change the port values that your webserver will listen to and that the Administration Interface uses respectively.

Web Server Port: 80 OK

Zend Server Interface Port: 10081 OK

Figure 1-4. *Apache Port Number*

Click “Next.” Select “Custom Install” and check all options except the ones that are unchecked in following screen shot:


Zend Server Community Edition Installer
zend server Community Edition
zend

Custom Setup

Select components to install

Only components with a marked checkbox will be installed. After installation run the Installer again to add components not selected this time.

- [✓] Zend Data Cache
- [✓] Zend Java Bridge
- [✓] Zend Framework
  - [✓] Base
  - [✓] Extras (separate download)
  - [✓] Dojo (separate download)
- [ ] Informix
- [✓] Oracle OCI Driver
- [✓] phpMyAdmin (separate download)
- [ ] IBM DB2 RTCL (separate download)
- [✓] MySQL Server (separate download)
- [ ] MS SQL Native Client (separate download)

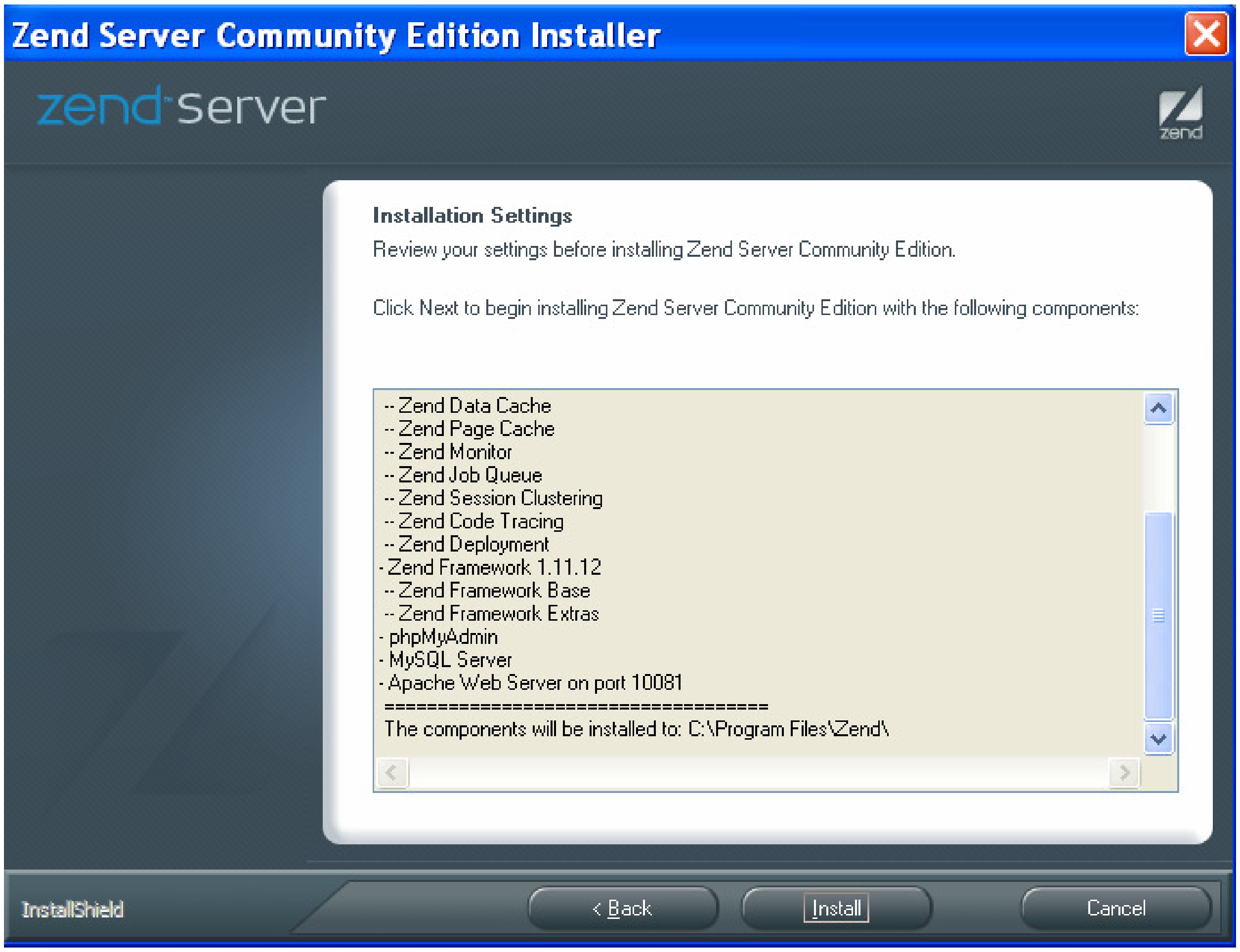
Description

Installs the MySQL Essential Database Server (version 5.1.50)

160.06 MB of space required on the C drive

3435.42 MB of space available on the C drive

Figure 1-5. *Zend Server CE Custom Setup*



Installation Settings

Review your settings before installing Zend Server Community Edition.

Click Next to begin installing Zend Server Community Edition with the following components:

```
-- Zend Data Cache
-- Zend Page Cache
-- Zend Monitor
-- Zend Job Queue
-- Zend Session Clustering
-- Zend Code Tracing
-- Zend Deployment
- Zend Framework 1.11.12
-- Zend Framework Base
-- Zend Framework Extras
- phpMyAdmin
- MySQL Server
- Apache Web Server on port 10081
=====================================
The components will be installed to: C:\Program Files\Zend\
```

Figure 1-6. *Zend Server CE Installation Settings*

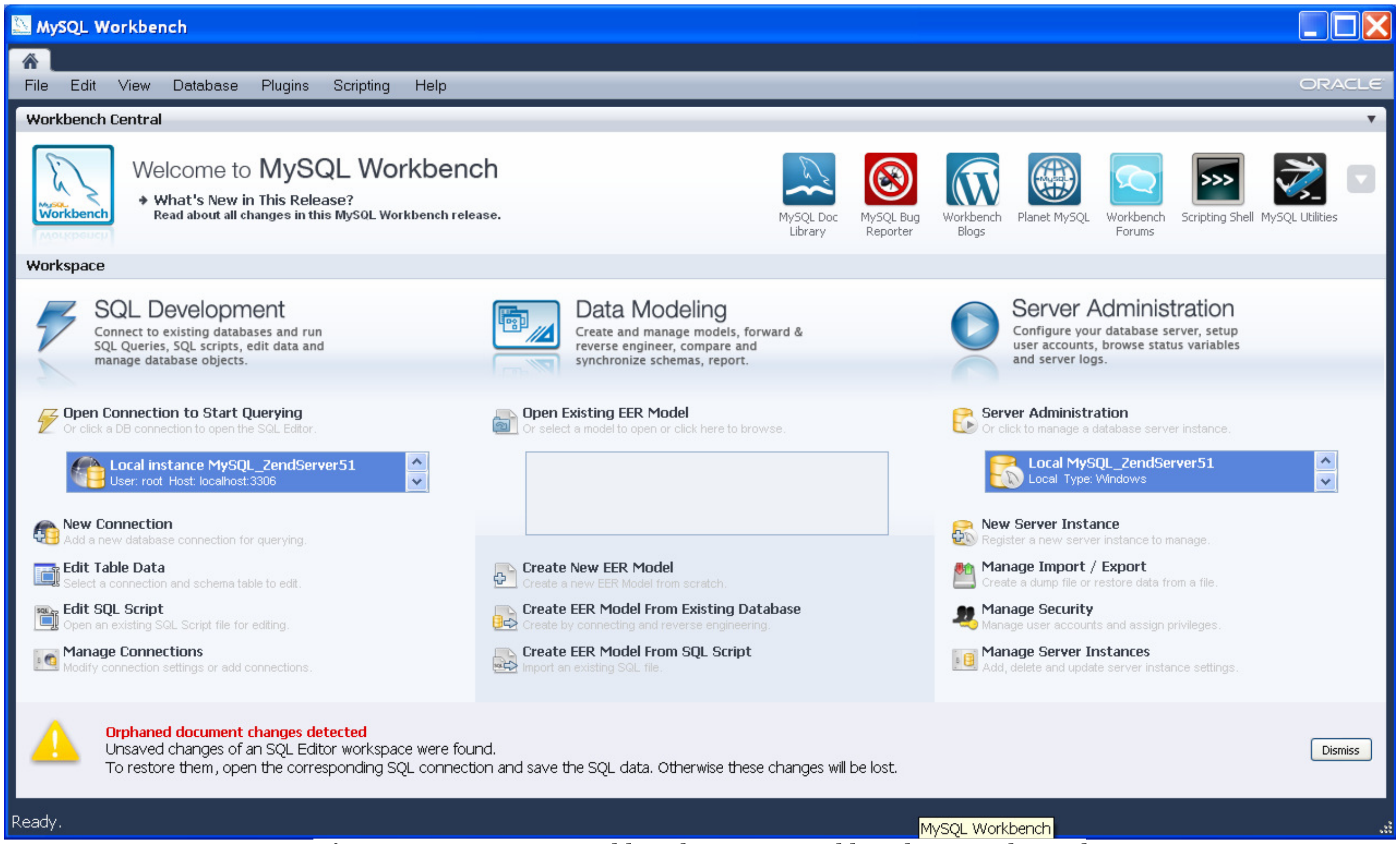
Complete the steps available at **http://localhost:10081/ZendServer/**

After the installation is complete, go to C:\Program Files\Zend\ZendServer\etc\php.ini and set:

```
display_errors = On
display_startup_errors = On
```

MySQL Workbench 5.2 CE

In the installation of the Zend Server Community Edition we included phpMyAdmin. I am going to use **MySQL Workbench CE Version 5.2** as the MySQL database interface for all pertinent discussions in this book, feel free to use which ever you prefer. phpMyAdmin is a very versatile and user friendly tool however I have discovered that occasionally people have problems logging into it when it is installed with Zend Server CE. You can edit the username and password in the C:\Program Files\Zend\phpMyAdmin\config.inc file however typically if you just use the username “admin” and leave the password field blank you should be able to access it.


MySQL Workbench
File
Edit
View
Database
Plugins
Scripting
Help
ORACLE
Workbench Central
Welcome to MySQL Workbench
What's New in This Release?
Read about all changes in this MySQL Workbench release.
MySQL Doc Library
MySQL Bug Reporter
Workbench Blogs
Planet MySQL
Workbench Forums
Scripting Shell
MySQL Utilities
Workspace
SQL Development
Connect to existing databases and run SQL Queries, SQL scripts, edit data and manage database objects.
Data Modeling
Create and manage models, forward & reverse engineer, compare and synchronize schemas, report.
Server Administration
Configure your database server, setup user accounts, browse status variables and server logs.
Open Connection to Start Querying
Or click a DB connection to open the SQL Editor.
Local instance MySQL_ZendServer51
User: root Host: localhost:3306
Open Existing EER Model
Or select a model to open or click here to browse.
Server Administration
Or click to manage a database server instance.
Local MySQL_ZendServer51
Local Type: Windows
New Connection
Add a new database connection for querying.
Edit Table Data
Select a connection and schema table to edit.
Edit SQL Script
Open an existing SQL Script file for editing.
Manage Connections
Modify connection settings or add connections.
Create New EER Model
Create a new EER Model from scratch.
Create EER Model From Existing Database
Create by connecting and reverse engineering.
Create EER Model From SQL Script
Import an existing SQL file.
New Server Instance
Register a new server instance to manage.
Manage Import / Export
Create a dump file or restore data from a file.
Manage Security
Manage user accounts and assign privileges.
Manage Server Instances
Add, delete and update server instance settings.
Dismiss
Ready.
MySQL Workbench

Orphaned document changes detected
Unsaved changes of an SQL Editor workspace were found.
To restore them, open the corresponding SQL connection and save the SQL data. Otherwise these changes will be lost.

Figure 1-7. *MySQL Workbench 5.2 CE Workbench Control Panel*

MySQL Workbench 5.2 CE is a comprehensive open source database design, development and administrative tool and can be downloaded from:

http://dev.mysql.com/downloads/workbench/5.2.html

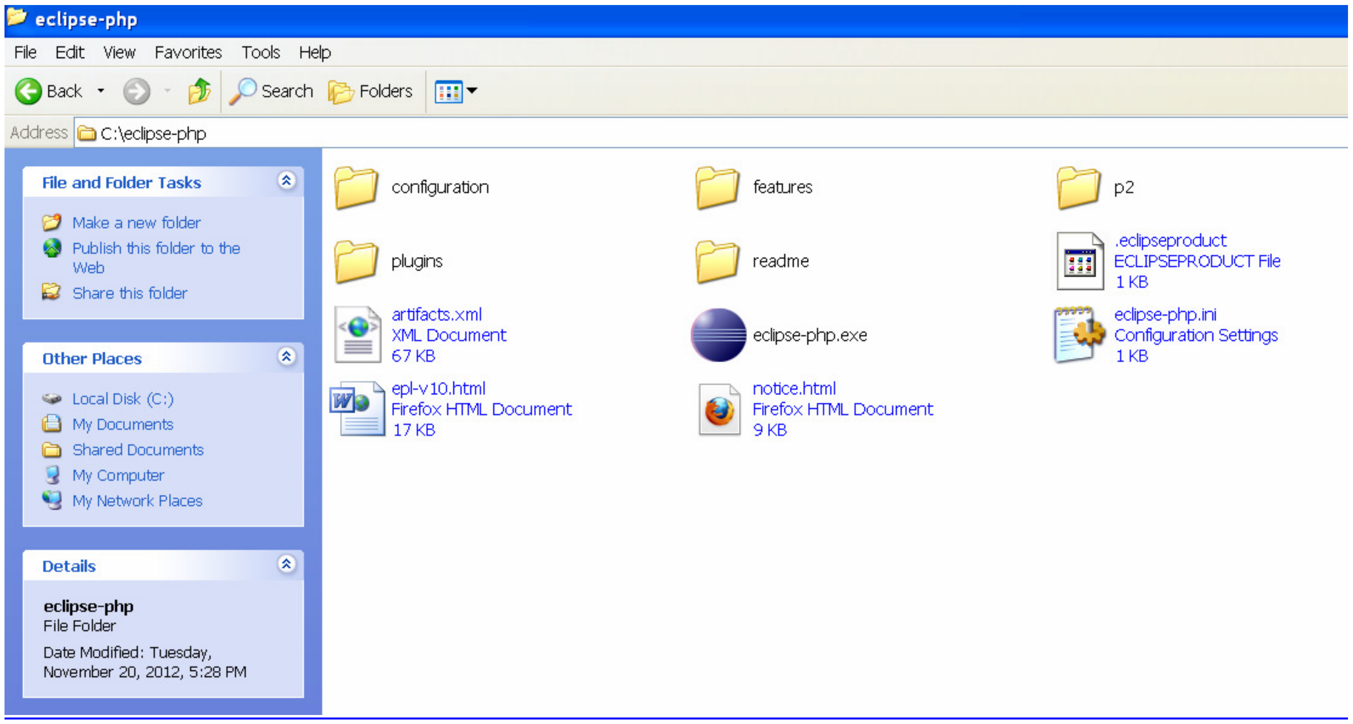
Setting Up Your Integrated Development Environment

For the purpose of this book I will be using **Eclipse for PHP Developers Version: Helios Release** to run code snippets from the examples in the following chapters. Eclipse is optional because you can execute your test PHP scripts via your preferred browser by using the Apache path: http://localhost/mytestfile.php. Your source files will need to be copied into the Apache htdocs directory. According to the set up that I explain in this book that path is: C:\Program Files\Zend\Apache2\htdocs .

I you would like to go ahead and use Eclipse, you can download it from:

http://www.eclipse.org/downloads/packages/eclipse-php-developers/heliosr

Eclipse is available for Windows, Linux and Macs. On the right side of the download page you just need to click the link that corresponds to your platform, download the zip file and extract the eclipse files to a convenient directory.

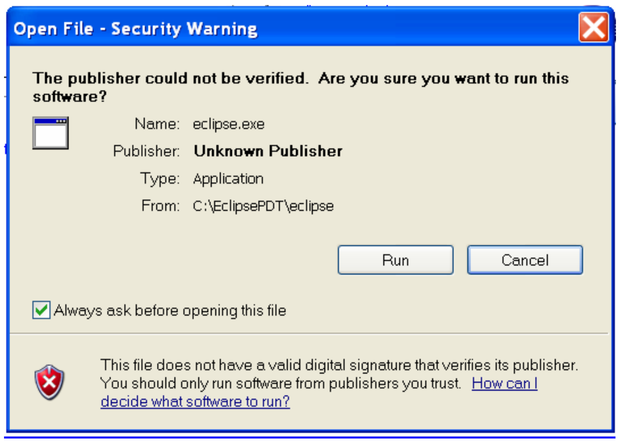


eclipse-php
File Folder
Date Modified: Tuesday, November 20, 2012, 5:28 PM

configuration
features
p2
plugins
readme
.eclipseproduct ECLIPSEPRODUCT File 1 KB
artifacts.xml XML Document 67 KB
eclipse-php.exe
eclipse-php.ini Configuration Settings 1 KB
epl-v10.html Firefox HTML Document 17 KB
notice.html Firefox HTML Document 9 KB

Figure 1-8. *Eclipse Helio PDT Start-up Icon*

Double click the eclipse-php.exe icon then click "run" in the dialog that pops up:



The publisher could not be verified. Are you sure you want to run this software?

Name: eclipse.exe
Publisher: Unknown Publisher
Type: Application
From: C:\EclipsePDT\eclipse

☑ Always ask before opening this file

This file does not have a valid digital signature that verifies its publisher. You should only run software from publishers you trust. How can I decide what software to run?

Figure 1-9. *Eclipse Helio PDT Run Dialog*

Verify that the Eclipse splash screen indicates that it is the Helios version:

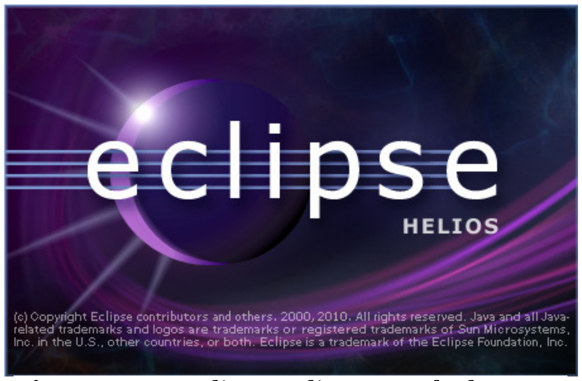
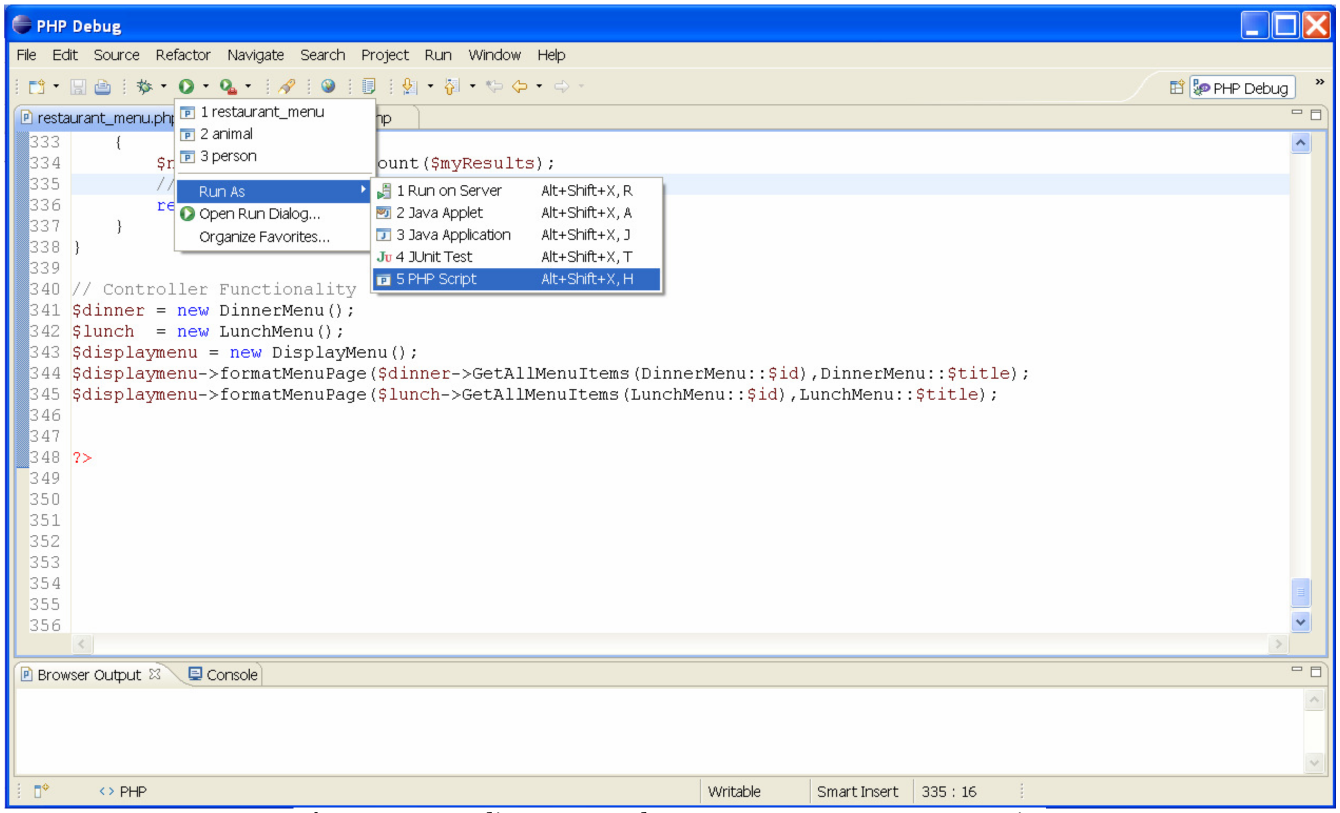

eclipse
HELIOS
(c) Copyright Eclipse contributors and others, 2000, 2010. All rights reserved. Java and all Java-related trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S., other countries, or both. Eclipse is a trademark of the Eclipse Foundation, Inc.

Figure 1-10. *Eclipse Helio PDT Splash Screen*

Now you are ready to start coding in Eclipse. To execute a PHP script click the green "run" arrow on the tool bar then select:

Run As -> PHP Script


PHP Debug
File Edit Source Refactor Navigate Search Project Run Window Help
PHP Debug
restaurant_menu.php
1 restaurant_menu
2 animal
3 person
Run As
Open Run Dialog...
Organize Favorites...
1 Run on Server Alt+Shift+X, R
2 Java Applet Alt+Shift+X, A
3 Java Application Alt+Shift+X, J
4 JUnit Test Alt+Shift+X, T
5 PHP Script Alt+Shift+X, H
Browser Output
Console
PHP
Writable
Smart Insert
335 : 16

```
        {
            $r                              ount($myResults);
            //
            re
        }
}

// Controller Functionality
$dinner = new DinnerMenu();
$lunch  = new LunchMenu();
$displaymenu = new DisplayMenu();
$displaymenu->formatMenuPage($dinner->GetAllMenuItems(DinnerMenu::$id),DinnerMenu::$title);
$displaymenu->formatMenuPage($lunch->GetAllMenuItems(LunchMenu::$id),LunchMenu::$title);


?>
```

Figure 1-11. *Eclipse PHP Debug Screen: Run As -> PHP Script*

Index

$

$_GET, 78, 86, 87, 93, 98, 99, 102, 128, 129
$_POST, 90, 91, 96, 98, 100, 118, 128, 129
$this variable, 21, 53

&

& CSS, Events, 8

.

.htaccess, 4, 77, 79

_

__call, 3, 33, 34
__clone, 36
__construct, 3, 19, 20, 22, 23, 27, 28, 30, 31, 32, 41, 48, 50, 60, 64, 79, 80, 81, 106
__destruct, 3, 31, 60
__get(), 3, 31, 32, 33, 59
__isset, 3, 31, 59
__set(), 3, 31, 32, 33, 59
__sleep, 34
__wakeup, 35

A

abstract, 3, 39, 40, 41, 42, 45, 53, 59, 61, 62, 70, 71, 81, 86, 96, 129
Access Modifiers, 3, 21
Agile development, 8
AJAX, 8
AMP, 9
anon-access, 137
any(), 130, 131
assertArrayHasKey(), 128
assertClassHasAttribute(, 128
assertClassHasStaticAttribute, 128
assertContains, 128
assertContainsOnly, 128
assertContainsOnlyInstancesOf, 128
assertCount, 128
assertEmpty, 128
assertEquals, 128
assertEqualXMLStructure, 128
assertFalse, 128
assertFileEquals, 128
assertFileExists, 128
assertGreaterThan, 128
assertGreaterThanOrEqual, 128
assertInstanceOf, 128
assertInternalType, 128
Assertion Methods, 5, 123, 128
assertJsonFileEqualsJsonFile, 128

assertJsonStringEqualsJsonFile, 128
assertJsonStringEqualsJsonString, 128
assertLessThan, 128
assertLessThanOrEqual, 128
assertNull, 128
assertObjectHasAttribute, 128
assertRegExp, 128
assertSame, 128
assertSelectCount, 128
assertSelectEquals, 128
assertSelectRegExp, 128
assertStringEndsWith, 128
assertStringEqualsFile, 128
assertStringMatchesFormat, 128
assertStringMatchesFormatFile, 128
assertStringStartsWith, 128
assertTag, 128
assertThat, 128
assertTrue, 128
assertXmlFileEqualsXmlFile, 128
assertXmlStringEqualsXmlFile, 128
assertXmlStringEqualsXmlString, 128
`at($num)`, 131
`atLeastOnce()`, 131
auth-access, 137

C

CAPTCHA, 4, 115, 117, 118, 119
`class Database`, 58, 65
Coding-to-the-Interface, 8
Controller, 77

D

Dependency Injection, 129
Design-by-Contract, 8
Displaying Startup Errors, 68
DOM Manipulation, 8
Dynamic Binding, 3, 28

E

Eclipse, 3, 5, 14, 15, 16, 33, 55, 132, 133, 135, 162
Eclipse for PHP Developers Version: Helios Release, 14
Effects & Animations, 8
encapsulation, 18, 23, 26
Entity Relationship Diagram, 90
Error Reporting Sensitivity, 67
Error Sensitivity, 67
error_reporting, 75
Error-Reporting Sensitivity Levels, 67
`exactly($num)`, 131
Exception Class, 4, 64
Exception Handling, 4, 64
extends, 22, 23, 24, 27, 28, 30, 31, 40, 41, 42, 44, 46, 47, 48, 49, 53, 54, 61, 62, 64, 71, 72, 73, 77, 85, 91, 96, 98, 105, 123, 124, 125, 126, 127, 130, 131
Extreme Programming, 8

F

final, 3, 41, 42, 46, 47, 54, 64, 72, 105, 106
FollowSymLinks, 77

G

getCode, 64
getFile, 64
getLine, 64
getPrevious, 64
getTrace, 64
getTraceAsString, 64
go-pear, 111, 114, 115

H

Horizontal Inheritance, 3, 25

I

Identifying the Log File, 68
Ignoring Errors Originating from the Same Location, 68
Ignoring Repeated Errors, 68
implements, 3, 33, 44, 45, 47, 48, 49, 50, 54, 72, 105, 106
index.php, 78
inheritance, 18, 22, 23, 25, 26, 30, 39, 42, 44, 48, 49, 61
ini_set, 67, 75, 78, 86, 93, 98, 101
Instantiation, 3, 20
interface, 3, 13, 26, 27, 39, 44, 45, 46, 47, 48, 49, 50, 54, 55, 72, 73, 94, 96, 109, 121, 131, 138, 146, 151, 160
Invoking parent methods, 24

J

JQuery
  Core Functionality, 8
  DOM Selection, 8
  Extensibility through Plug-ins, 8
  User Interface, 8

L

LAMP, 10
Late Binding, 3, 28
Linux, 4, 5, 10, 11, 14, 68, 115, 122, 135, 152
LOG_ALERT, 70
LOG_CRIT, 70
LOG_DEBUG, 70
LOG_EMERG, 70
LOG_ERR, 70
LOG_INFO, 70
LOG_NOTICE, 70
LOG_WARNING, 70
Logging Errors, 68
Logging Options, 69

M

Mac OS, 4, 5, 11, 114, 122, 135, 146, 151
MAMP, 10
Method overloading, 26
Method Overriding, 3, 23, 26, 41
Mocks, 5, 129
Model, 4, 70, 77, 86, 91, 99, 160
MVC Folder Structure, 78
MVC URL Structure, 77

MySQL Workbench 5.2 CE, 3, 13, 14, 57, 58, 162

N

`never()`, 131
`new` keyword, 20

O

`Once()`, 131

P

password-db, 137
PEAR, 4, 9, 111, 112, 113, 114, 115, 116, 117, 118, 119, 121, 122, 162
PHP Extension and Application Repository, 111
php.ini, 13, 67, 112, 113, 114, 115, 122
PHPUnit, 4, 5, 9, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 160, 162
polymorphism and then elaborate on them., 18
`preg_replace`, 82, 83
Private, 21
Protected, 21
Public, 21, 61, 62

R

`ReflectionClass`, 104, 105, 106
repository, 111, 135, 136, 137, 139, 141, 144, 146, 147, 148, 149, 150, 151, 152, 154, 155, 157
`RewriteEngine`, 77

S

Scope Resolution Operator, 4, 52
Setting the Maximum Log Line Length, 68
Singleton, 60
`static`, 3, 4, 52, 53, 54, 55, 56, 58, 59, 60, 61, 62, 65, 71, 72, 79, 84, 94, 105, 106
Storing Most Recent Error in a Variable, 69
Stubs, 5, 129
Subversion, 5, 9, 135, 136, 137, 138, 139, 141, 146, 147, 148, 149, 150, 151, 152, 153, 155, 156, 157, 158, 162
SVN, 5, 9, 135, 136, 137, 138, 140, 141, 142, 143, 144, 148, 152, 160
- Adding Files, 4, 5, 135, 139, 143, 147, 148, 155
- branches, 141
- Branching, 5, 135, 145, 150, 157
- Committing, 5, 139, 148, 154
- Exporting, 5, 135, 145, 149, 156
- Graphical User Interfaces, 5, 151
- Removing Files, 5, 135, 148, 149, 155, 156
- Tagging, 5, 145, 149, 156
- tags, 141, 148, 149, 153, 154, 156, 157
- trunk, 139, 141, 148, 149, 150, 151, 153, 154, 156, 157, 158
- Updating, 5, 139, 148, 154

`svnserve.exe`, 137

T

Templates, 92, 94, 100, 162
Test Driven Development, 8
Test fixtures, 4, 122
Test Phases, 5, 123
the Logging Connection, 69
TortoiseSVN, 5, 135, 139, 140, 141, 142, 143, 144, 145, 162
Traits, 3, 25

U

URL Mapping, 77

W

WAMP, 10
Web Server Port, 11
Writing Test Cases, 5, 123

X

XAMP, 10

Z

Zend Framework, 4, 111, 121, 160
Zend Server
  System Requirements, 10
Zend Server Interface Port, 11
Zend\phpMyAdmin\config.inc, 13