

---

# PHP 5 Advanced OOP and Design Patterns

“I made up the term ‘object-oriented,’ and I can tell you I didn’t have C++ in mind.”—Alan Kay, OOPSLA ’97

## 4.1 INTRODUCTION

In this chapter, you learn how to use PHP’s more advanced object-oriented capabilities. When you finish reading this chapter, you will have learned

- ▣ Overloading capabilities that can be controlled from PHP code
- ▣ Using design patterns with PHP 5
- ▣ The new reflection API

## 4.2 OVERLOADING CAPABILITIES

In PHP 5, extensions written in C can overload almost every aspect of the object syntax. It also allows PHP code to overload a limited subset that is most often needed. This section covers the overloading abilities that you can control from your PHP code.

### 4.2.1 Property and Method Overloading

PHP allows overloading of property access and method calls by implementing special proxy methods that are invoked if the relevant property or method doesn’t exist. This gives you a lot of flexibility in intercepting these actions and defining your own functionality.

You may implement the following method prototypes:

```
function __get($property)
function __set($property, $value)
function __call($method, $args)
```

`__get` is passed the property's name, and you should return a value.

`__set` is passed the property's name and its new value.

`__call` is passed the method's name and a numerically indexed array of the passed arguments starting from 0 for the first argument.

The following example shows how to use the `__set` and `__get` functions (`array_key_exists()` is covered later in this book; it checks whether a key exists in the specified array):

```
class StrictCoordinateClass {
    private $arr = array('x' => NULL, 'y' => NULL);

    function __get($property)
    {
        if (array_key_exists($property, $this->arr)) {
            return $this->arr[$property];
        } else {
            print "Error: Can't read a property other than x & y\n";
        }
    }

    function __set($property, $value)
    {
        if (array_key_exists($property, $this->arr)) {
            $this->arr[$property] = $value;
        } else {
            print "Error: Can't write a property other than x & y\n";
        }
    }
}

$obj = new StrictCoordinateClass();

$obj->x = 1;
print $obj->x;

print "\n";

$obj->n = 2;
print $obj->n;
```

The output is

```
1
Error: Can't write a property other than x & y
Error: Can't read a property other than x & y
```

As `x` exists in the object's array, the setter and getter method handlers agrees to read/write the values. However, when accessing the property `n`, both for reading and writing, `array_key_exists()` returns `false` and, therefore, the error messages are reached.

`__call()` can be used for a variety of purposes. The following example shows how to create a delegation model, in which an instance of the class `HelloWorldDelegator` delegates all method calls to an instance of the `HelloWorld` class:

```
class HelloWorld {
    function display($count)
    {
        for ($i = 0; $i < $count; $i++) {
            print "Hello, World\n";
        }
        return $count;
    }
}

class HelloWorldDelegator {
    function __construct()
    {
        $this->obj = new HelloWorld();
    }

    function __call($method, $args)
    {
        return call_user_func_array(array($this->obj , $method),
            ↪$args);
    }

    private $obj;
}

$obj = new HelloWorldDelegator();
print $obj->display(3);
```

This script's output is

```
Hello, World
Hello, World
Hello, World
3
```

The `call_user_func_array()` function allows `__call()` to relay the function call with its arguments to `HelloWorld::display()` which prints out "Hello, World\n" three times. It then returns `$count` (in this case, 3) which is then printed out. Not only can you relay the method call to a different object (or handle it in whatever way you want), but you can also return a value from `__call()`, just like a regular method.

## 4.2.2 Overloading the Array Access Syntax

It is common to have key/value mappings or, in other words, lookup dictionaries in your application framework. For this purpose, PHP supports **associative arrays** that map either integer or string values to any other PHP value. This feature was covered in Chapter 2, “PHP 5 Basic Language,” and in case you forgot about it, here’s an example that looks up the user John’s social-security number using an associative array which holds this information:

```
print "John's ID number is " . $userMap["John"];
```

Associative arrays are extremely convenient when you have all the information at hand. But consider a government office that has millions of people in its database; it just wouldn’t make sense to load the entire database into the `$userMap` associative array just to look up one user. A possible alternative is to write a method that will look up the user’s id number via a database call. The previous code would look something like the following:

```
print "John's ID number is " . $db->FindIDNumber("John");
```

This example would work well, but many developers prefer the associative array syntax to access key/value-like dictionaries. For this purpose, PHP 5 enables you to overload an object so that it can behave like an array. Basically, it would enable you to use the array syntax, but behind the scenes, a method written by you would be called, which would execute the relevant database call, returning the wanted value.

It is really a matter of personal preference as to what method to use. Sometimes, it is nicer to use this overloading ability than the verbosity of calling a method, and it’s up to you to decide which method suits you best.

To allow your class to overload the array syntax, it needs to implement the `ArrayAccess` interface (see Figure 4.1).

<b>interface ArrayAccess</b>
<code>bool offsetExists(\$index)</code>
<code>mixed offsetGet(\$index)</code>
<code>void offsetSet(\$index,\$new_value)</code>
<code>void offsetUnset(\$index)</code>

**Fig. 4.1** ArrayAccess interface.

The following example shows how to use it. It is incomplete because the database methods themselves aren't implemented:

```
class UserToSocialSecurity implements ArrayAccess {
    private $db; // An object which includes database access methods

    function offsetExists($name) {
        return $this->db->userExists($name);
    }

    function offsetGet($name) {
        return $this->db->getUserId($name);
    }

    function offsetSet($name, $id) {
        $this->db->setUserId($name, $id);
    }

    function offsetUnset($name) {
        $this->db->removeUser($name);
    }
}

$userMap = new UserToSocialSecurity();

print "John's ID number is " . $userMap["John"];
```

You can see that the object `$userMap` is used just like an array, but behind the scenes, when the `$userMap["John"]` lookup is performed, the `offsetGet()` method is invoked, which in turn calls the database `getUserId()` method.

## 4.3 ITERATORS

The properties of an object can be iterated using the `foreach()` loop:

```
class MyClass {
    public $name = "John";
    public $sex = "male";
}

$obj = new MyClass();

foreach ($obj as $key => $value) {
```

```

    print "obj[$key] = $value\n";
}

```

Running this script results in

```

obj[name] = John
obj[sex] = male

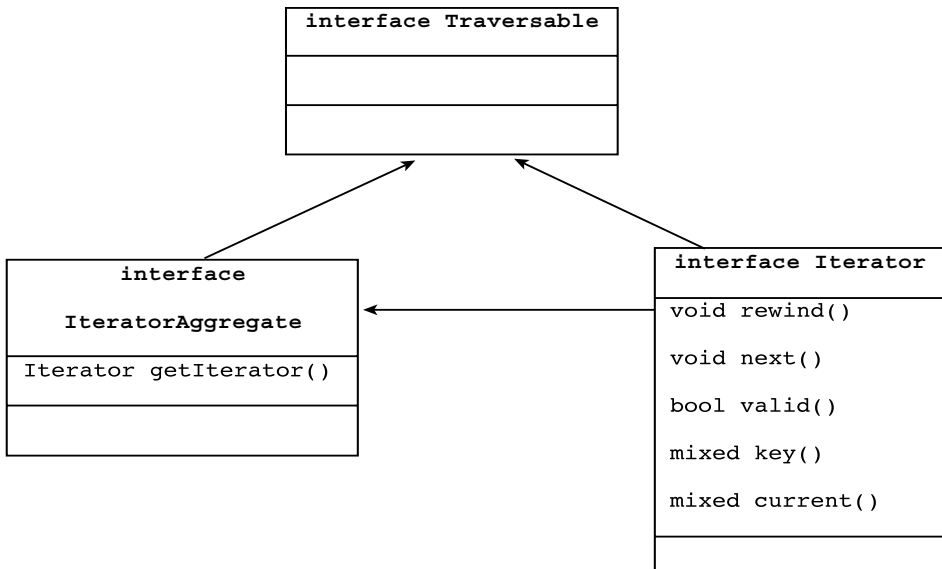
```

However, often when you write object-oriented code, your classes don't necessarily represent a simple key/value array as in the previous example, but represent more complex data, such as a database query or a configuration file.

PHP 5 allows you to overload the behavior of the `foreach()` iteration from within your code so you can have it do what makes sense in respect to your class's design.

**Note:** Not only does PHP 5 enable you to overload this behavior, but it also allows extension authors to override such behavior, which has brought iterator support to various PHP extensions such as SimpleXML and SQLite.

To overload iteration for your class kind, you need to adhere to certain interfaces that are pre-defined by the language (see Figure 4.2).



**Fig. 4.2** Class diagram of Iterator hierarchy.

Any class that implements the `Traversable` interface is a class that can be traversed using the `foreach()` construct. However, `Traversable` is an empty interface that shouldn't be implemented directly; instead, you should either implement `Iterator` or `IteratorAggregate` that inherit from `Traversable`.

The main interface is `Iterator`. It defines the methods you need to implement to give your classes the `foreach()` iteration capabilities. These methods should be public and are listed in the following table.

<b>Interface Iterator</b>	
<code>void rewind()</code>	Rewinds the iterator to the beginning of the list (this might not always be possible to implement).
<code>mixed current()</code>	Returns the value of the current position.
<code>mixed key()</code>	Returns the key of the current position.
<code>void next()</code>	Moves the iterator to the next key/value pair.
<code>bool valid()</code>	Returns <code>true/false</code> if there are more values (used before the call to <code>current()</code> or <code>key()</code> ).

If your class implements the `Iterator` interface, it will be traversable with `foreach()`. Here's a simple example:

```
class NumberSquared implements Iterator {
    public function __construct($start, $end)
    {
        $this->start = $start;
        $this->end = $end;
    }

    public function rewind()
    {
        $this->cur = $this->start;
    }

    public function key()
    {
        return $this->cur;
    }

    public function current()
    {
        return pow($this->cur, 2);
    }

    public function next()
    {
        $this->cur++;
    }
}
```

```
    }

    public function valid()
    {
        return $this->cur <= $this->end;
    }

    private $start, $end;
    private $cur;
}

$obj = new NumberSquared(3, 7);

foreach ($obj as $key => $value) {
    print "The square of $key is $value\n";
}
```

The output is

```
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
```

This example demonstrates how you can implement your own behavior for iterating a class. In this case, the class represents the square of integers, and after given a minimum and maximum value, iterating over those values will give you the number itself and its square.

Now in many cases, your class itself will represent data and have methods to interact with this data. The fact that it also requires an iterator might not be its main functionality. Also, when iterating an object, the state of the iteration (current position) is usually stored in the object itself, thus not allowing for nested iterations. For these two reasons, you may separate the implementation of your class and its iterator by making your class implement the `IteratorAggregate` interface. Instead of having to define all the previous methods, you need to define a method that returns an object of a different class, which implements the iteration scheme for your class.

The public method you need to implement is `Iterator getIterator()` because it returns an iterator object that handles the iteration for this class.

By using this method of separating between the class and its iterator, we can rewrite the previous example the following way:

```
class NumberSquared implements IteratorAggregate {
    public function __construct($start, $end)
    {
        $this->start = $start;
        $this->end = $end;
    }
}
```

```
public function getIterator()
{
    return new NumberSquaredIterator($this);
}

public function getStart()
{
    return $this->start;
}

public function getEnd()
{
    return $this->end;
}

private $start, $end;
}

class NumberSquaredIterator implements Iterator {
    function __construct($obj)
    {
        $this->obj = $obj;
    }

    public function rewind()
    {
        $this->cur = $this->obj->getStart();
    }

    public function key()
    {
        return $this->cur;
    }

    public function current()
    {
        return pow($this->cur, 2);
    }

    public function next()
    {
        $this->cur++;
    }

    public function valid()
    {
        return $this->cur <= $this->obj->getEnd();
    }

    private $cur;
    private $obj;
}
```

```
$obj = new NumberSquared(3, 7);

foreach ($obj as $key => $value) {
    print "The square of $key is $value\n";
}
```

The output is the same as the previous example. You can clearly see that the `IteratorAggregate` interface enables you to separate your classes' main functionality and the methods needed for iterating it into two independent entities.

Choose whatever method suits the problem at hand. It really depends on the class and its functionality as to whether the iterator should be in a separate class.

## 4.4 DESIGN PATTERNS

So, what exactly qualifies a language as being **object-oriented** (OO)? Some people believe that any language that has objects that encapsulate data and methods can be considered OO. Others would also include polymorphism via inheritance and access modifiers into the definition. The purists would probably list dozens of pages of things they think an OO language must support, such as exceptions, method overloading, reflection, strict typing, and more. You can bet that none of these people would ever agree with each other because of the diversity of OOP languages, each of them good for certain tasks and not quite as good for others.

However, what most people would agree with is that developing OO software is not only about the syntax and the language features but it is a state of mind. Although there are some professionally written programs in functional languages such as C (for example, PHP), people developing in OO languages tend to give the software design more of an emphasis. One reason might be the fact that OO languages tend to contain features that help in the design phase, but the main reason is probably cultural because the OO community has always put a lot of emphasis on good design.

This chapter covers some of the more advanced OO techniques that are possible with PHP, including the implementation of some common design patterns that are easily adapted to PHP.

When designing software, certain programming patterns repeat themselves. Some of these have been addressed by the software design community and have been given accepted general solutions. These repeating problems are called **design patterns**. The advantage of knowing and using these patterns is not only to save time instead of reinventing the wheel, but also to give developers a common language in software design. You'll often hear software developers say, "Let's use a singleton pattern for this," or "Let's use a factory pattern for that." Due to the importance of these patterns in today's software development, this section covers some of these patterns.

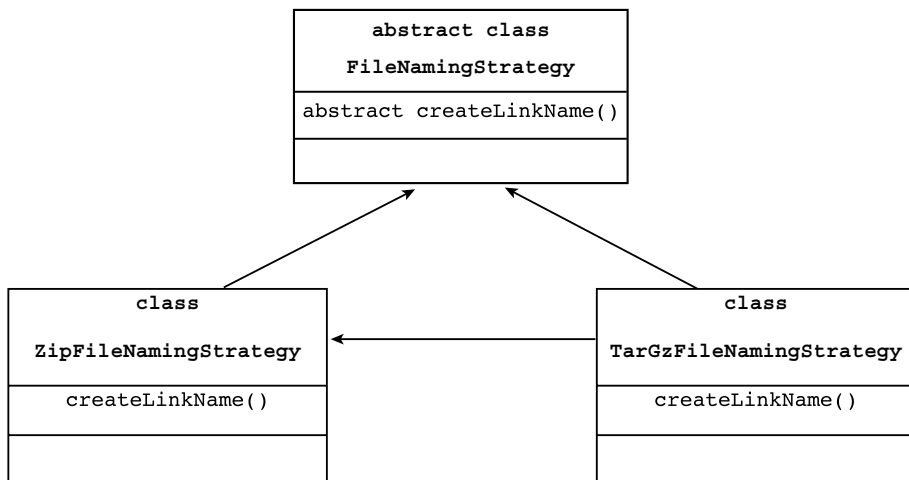
### 4.4.1 Strategy Pattern

The **strategy pattern** is typically used when your programmer's algorithm should be interchangeable with different variations of the algorithm. For example, if you have code that creates an image, under certain circumstances, you might want to create JPEGs and under other circumstances, you might want to create GIF files.

The strategy pattern is usually implemented by declaring an abstract base class with an algorithm method, which is then implemented by inheriting concrete classes. At some point in the code, it is decided what concrete strategy is relevant; it would then be instantiated and used wherever relevant.

Our example shows how a download server can use a different file selection strategy according to the web client accessing it. When creating the HTML with the download links, it will create download links to either .tar.gz files or .zip files according to the browser's OS identification. Of course, this means that files need to be available in both formats on the server. For simplicity's sake, assume that if the word "Win" exists in `$_SERVER["HTTP_USER_AGENT"]`, we are dealing with a Windows system and want to create .zip links; otherwise, we are dealing with systems that prefer .tar.gz.

In this example, we would have two strategies: the .tar.gz strategy and the .zip strategy, which is reflected as the following strategy hierarchy (see Figure 4.3).



**Fig. 4.3** Strategy hierarchy.

The following code snippet should give you an idea of how to use such a strategy pattern:

```

abstract class FileNamingStrategy {
    abstract function createLinkName($filename);
}

class ZipFileNamingStrategy extends FileNamingStrategy {
    function createLinkName($filename)
    {
        return "http://downloads.foo.bar/$filename.zip";
    }
}

class TarGzFileNamingStrategy extends FileNamingStrategy {
    function createLinkName($filename)
    {
        return "http://downloads.foo.bar/$filename.tar.gz";
    }
}

if (strstr($_SERVER["HTTP_USER_AGENT"], "Win")) {
    $fileNamingObj = new ZipFileNamingStrategy();
} else {
    $fileNamingObj = new TarGzFileNamingStrategy();
}

$calc_filename = $fileNamingObj->createLinkName("Calc101");
$stat_filename = $fileNamingObj->createLinkName("Stat2000");

print <<<EOF
<h1>The following is a list of great downloads<</h1>
<br>
<a href="$calc_filename">A great calculator</a><br>
<a href="$stat_filename">The best statistics application</a><br>
<br>
EOF;

```

Accessing this script from a Windows system gives you the following HTML output:

```

<h1>The following is a list of great downloads<</h1>
<br>
<a href="http://downloads.foo.bar/Calc101.zip">A great calculator<
➤a><br>
<a href="http://downloads.foo.bar/Stat2000.zip">The best statistics
➤application</a><br>
<br>

```

**Tip:** The strategy pattern is often used with the factory pattern, which is described later in this section. The factory pattern selects the correct strategy.

### 4.4.2 Singleton Pattern

The **singleton pattern** is probably one of the best-known design patterns. You have probably encountered many situations where you have an object that handles some centralized operation in your application, such as a logger object. In such cases, it is usually preferred for only one such application-wide instance to exist and for all application code to have the ability to access it. Specifically, in a logger object, you would want every place in the application that wants to print something to the log to have access to it, and let the centralized logging mechanism handle the filtering of log messages according to log level settings. For this kind of situation, the singleton pattern exists.

Making your class a singleton class is usually done by implementing a static class method `getInstance()`, which returns the only single instance of the class. The first time you call this method, it creates an instance, saves it in a private static variable, and returns you the instance. The subsequent times, it just returns you a handle to the already created instance.

Here's an example:

```
class Logger {
    static function getInstance()
    {
        if (self::$instance == NULL) {
            self::$instance = new Logger();
        }
        return self::$instance;
    }

    private function __construct()
    {
    }

    private function __clone()
    {
    }

    function Log($str)
    {
        // Take care of logging
    }

    static private $instance = NULL;
}

Logger::getInstance()->Log("Checkpoint");
```

The essence of this pattern is `Logger::getInstance()`, which gives you access to the logging object from anywhere in your application, whether it is from a function, a method, or the global scope.

In this example, the constructor and clone methods are defined as `private`. This is done so that a developer can't mistakenly create a second instance of the `Logger` class using the `new` or `clone` operators; therefore, `getInstance()` is the only way to access the singleton class instance.

### 4.4.3 Factory Pattern

Polymorphism and the use of base class is really the center of OOP. However, at some stage, a concrete instance of the base class's subclasses must be created. This is usually done using the **factory pattern**. A `Factory` class has a `static` method that receives some input and, according to that input, it decides what class instance to create (usually a subclass).

Say that on your web site, different kinds of users can log in. Some are guests, some are regular customers, and others are administrators. In a common scenario, you would have a base class `User` and have three subclasses: `GuestUser`, `CustomerUser`, and `AdminUser`. Likely `User` and its subclasses would contain methods to retrieve information about the user (for example, permissions on what they can access on the web site and their personal preferences).

The best way for you to write your web application is to use the base class `User` as much as possible, so that the code would be generic and that it would be easy to add additional kinds of users when the need arises.

The following example shows a possible implementation for the four `User` classes, and the `UserFactory` class that is used to create the correct user object according to the username:

```
abstract class User {
    function __construct($name)
    {
        $this->name = $name;
    }

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

    // Permission methods
    function hasReadPermission()
    {
        return true;
    }

    function hasModifyPermission()
    {
        return false;
    }
}
```

```
    }

    function hasDeletePermission()
    {
        return false;
    }

    // Customization methods
    function wantsFlashInterface()
    {
        return true;
    }

    protected $name = NULL;
}

class GuestUser extends User {
}

class CustomerUser extends User {
    function hasModifyPermission()
    {
        return true;
    }
}

class AdminUser extends User {
    function hasModifyPermission()
    {
        return true;
    }

    function hasDeletePermission()
    {
        return true;
    }

    function wantsFlashInterface()
    {
        return false;
    }
}

class UserFactory {
    private static $users = array("Andi"=>"admin", "Stig"=>"guest",
                                  "Derick"=>"customer");

    static function Create($name)
    {
        if (!isset(self::$users[$name])) {
            // Error out because the user doesn't exist
        }
        switch (self::$users[$name]) {
            case "guest": return new GuestUser($name);
        }
    }
}
```

```
        case "customer": return new CustomerUser($name);
        case "admin": return new AdminUser($name);
        default: // Error out because the user kind doesn't exist
    }
}

function boolToStr($b)
{
    if ($b == true) {
        return "Yes\n";
    } else {
        return "No\n";
    }
}

function displayPermissions(User $obj)
{
    print $obj->getName() . "'s permissions:\n";
    print "Read: " . boolToStr($obj->hasReadPermission());
    print "Modify: " . boolToStr($obj->hasModifyPermission());
    print "Delete: " . boolToStr($obj->hasDeletePermission());
}

function displayRequirements(User $obj)
{
    if ($obj->wantsFlashInterface()) {
        print $obj->getName() . " requires Flash\n";
    }
}

$logins = array("Andi", "Stig", "Derick");

foreach($logins as $login) {
    displayPermissions(UserFactory::Create($login));
    displayRequirements(UserFactory::Create($login));
}
```

## Running this code outputs

```
Andi's permissions:
Read: Yes
Modify: Yes
Delete: Yes
Stig's permissions:
Read: Yes
Modify: No
Delete: No
Stig requires Flash
Derick's permissions:
Read: Yes
```

```
Modify: Yes
Delete: No
Derick requires Flash
```

This code snippet is a classic example of a factory pattern. You have a class hierarchy (in this case, the `User` hierarchy), which your code such as `displayPermissions()` treats identically. The only place where treatment of the classes differ is in the factory itself, which constructs these instances. In this example, the factory checks what kind of user the username belongs to and creates its class accordingly. In real life, instead of saving the user to user-kind mapping in a static array, you would probably save it in a database or a configuration file.

**Tip:** Besides `Create()`, you will often find other names used for the factory method, such as `factory()`, `factoryMethod()`, or `createInstance()`.

#### 4.4.4 Observer Pattern

PHP applications, usually manipulate data. In many cases, changes to one piece of data can affect many different parts of your application's code. For example, the price of each product item displayed on an e-commerce site in the customer's local currency is affected by the current exchange rate. Now, assume that each product item is represented by a PHP object that most likely originates from a database; the exchange rate itself is most probably being taken from a different source and is not part of the item's database entry. Let's also assume that each such object has a `display()` method that outputs the HTML relevant to this product.

The **observer pattern** allows for objects to register on certain events and/or data, and when such an event or change in data occurs, it is automatically notified. In this way, you could develop the product item to be an observer on the currency exchange rate, and before printing out the list of items, you could trigger an event that updates all the registered objects with the correct rate. Doing so gives the objects a chance to update themselves and take the new data into account in their `display()` method.

Usually, the observer pattern is implemented using an interface called `Observer`, which the class that is interested in acting as an observer must implement.

For example:

```
interface Observer {
    function notify($obj);
}
```

An object that wants to be "observable" usually has a `register` method that allows the `Observer` object to register itself. For example, the following might be our exchange rate class:

```
class ExchangeRate {
    static private $instance = NULL;
    private $observers = array();
    private $exchange_rate;

    private function ExchangeRate() {
    }

    static public function getInstance() {
        if (self::$instance == NULL) {
            self::$instance = new ExchangeRate();
        }
        return self::$instance;
    }

    public function getExchangeRate() {
        return $this->$exchange_rate;
    }

    public function setExchangeRate($new_rate) {
        $this->$exchange_rate = $new_rate;
        $this->notifyObservers();
    }

    public function registerObserver($obj) {
        $this->observers[] = $obj;
    }

    function notifyObservers() {
        foreach($this->observers as $obj) {
            $obj->notify($this);
        }
    }
}

class ProductItem implements Observer {
    public function __construct() {
        ExchangeRate::getInstance()->registerObserver($this);
    }

    public function notify($obj) {
        if ($obj instanceof ExchangeRate) {
            // Update exchange rate data
            print "Received update!\n";
        }
    }
}

$product1 = new ProductItem();
$product2 = new ProductItem();

ExchangeRate::getInstance()->setExchangeRate(4.5);
```

This code prints

```
Received update!  
Received update!
```

Although the example isn't complete (the `ProductItem` class doesn't do anything useful), when the last line executes (the `setExchangeRate()` method), both `$product1` and `$product2` are notified via their `notify()` methods with the new exchange rate value, allowing them to recalculate their cost.

This pattern can be used in many cases; specifically in web development, it can be used to create an infrastructure of objects representing data that might be affected by cookies, `GET`, `POST`, and other input variables.

## 4.5 REFLECTION

### 4.5.1 Introduction

New to PHP 5 are its **reflection** capabilities (also referred to as **introspection**). These features enable you to gather information about your script at runtime; specifically, you can examine your functions, classes, and more. It also enables you to access such language objects by using the available metadata. In many cases, the fact that PHP enables you to call functions indirectly (using `$func(...)`) or instantiate classes directly (`new $classname(...)`) is sufficient. However, in this section, you see that the provided reflection API is more powerful and gives you a rich set of tools to work directly with your application.

### 4.5.2 Reflection API

The reflection API consists of numerous classes that you can use to introspect your application. The following is a list of these items. The next section gives examples of how to use them.

```
interface Reflector  
static export(...)  
  
class ReflectionFunction implements Reflector  
__construct(string $name)  
string __toString()  
static mixed export(string $name [,bool $return = false])  
bool isInternal()  
bool isUserDefined()  
string getName()  
string getFileName()  
int getStartLine()
```

```
int getEndLine()
string getDocComment()
mixed[] getStaticVariables()
mixed invoke(mixed arg0, mixed arg1, ...)
bool returnsReference()
ReflectionParameter[] getParameters()

class ReflectionMethod extends ReflectionFunction implements
↳Reflector
bool isPublic()
bool isPrivate()
bool isProtected()
bool isAbstract()
bool isFinal()
bool isStatic()
bool isConstructor()
bool isDestructor()
int getModifiers()
ReflectionClass getDeclaringClass()

class ReflectionClass implements Reflector
string __toString()
static mixed export(string $name [,bool $return = false])
string getName()
bool isInternal()
bool isUserDefined()
bool isInstantiable()
string getFileName()
int getStartLine()
int getEndLine()
string getDocComment()
ReflectionMethod getConstructor()
ReflectionMethod getMethod(string $name)
ReflectionMethod[] getMethods(int $filter)
ReflectionProperty getProperty(string $name)
ReflectionProperty[] getProperties(int $filter)
mixed[] getConstants()
mixed getConstant(string $name)
ReflectionClass[] getInterfaces()
bool isInterface()
bool isAbstract()
bool isFinal()
int getModifiers()
bool isInstance($obj)
object newInstance(mixed arg0, arg1, ...)
ReflectionClass getParentClass()
bool isSubclassOf(string $class)
bool isSubclassOf(ReflectionClass $class)
mixed[] getStaticProperties()
mixed[] getDefaultProperties()
bool isIterable()
bool implementsInterface(string $ifc)
bool implementsInterface(ReflectionClass $ifc)
```

```
ReflectionExtension getExtension()
string getExtensionName()

class ReflectionParameter implements Reflector
static mixed export(mixed func, int/string $param [,bool $return =
↳false])
__construct(mixed func, int/string $param [,bool $return = false])
string __toString()
string getName()
bool isPassedByReference()
ReflectionClass getClass()
bool allowsNull()

class ReflectionExtension implements Reflector
static export(string $ext [,bool $return = false])
__construct(string $name)
string __toString()
string getName()
string getVersion()
ReflectionFunction[] getFunctions()
mixed[] getConstants()
mixed[] getINIEntries()
ReflectionClass[] getClasses()
String[] getClassNames()

class ReflectionProperty implements Reflector
static export(string/object $class, string $name, [,bool $return =
↳false])
__construct(string/object $class, string $name)
string getName()
mixed getValue($object)
setValue($object, mixed $value)
bool isPublic()
bool isPrivate()
bool isProtected()
bool isStatic()
bool isDefault()
int getModifiers()
ReflectionClass getDeclaringClass()

class Reflection
static mixed export(Reflector $r [, bool $return = 0])
static array getModifierNames(int $modifier_value)

class ReflectionException extends Exception
```

### 4.5.3 Reflection Examples

As you may have noticed, the reflection API is extremely rich and allows you to retrieve a large amount of information from your scripts. There are many situations where reflection could come in handy, and realizing this potential requires you to play around with the API on your own and use your imagination. In the meanwhile, we demonstrate two different ways you can use the reflection API. One is to give you runtime information of a PHP class (in this case an internal class), and the second is to implement a delegation model using the reflection API.

**4.5.3.1 Simple Example** The following code shows a simple example of using the `ReflectionClass::export()` static method to extract information about the class `ReflectionParameter`. It can be used to extract information of any PHP class:

```
ReflectionClass::export("ReflectionParameter");
```

The result is

```
Class [ <internal> class ReflectionProperty implements Reflector ] {  
  
    - Constants [0] {  
    }  
  
    - Static properties [0] {  
    }  
  
    - Static methods [1] {  
        Method [ <internal> static public method export ] {  
        }  
    }  
  
    - Properties [0] {  
    }  
  
    - Methods [13] {  
        Method [ <internal> final private method __clone ] {  
        }  
  
        Method [ <internal> <ctor> public method __construct ] {  
        }  
  
        Method [ <internal> public method __toString ] {  
        }  
  
        Method [ <internal> public method getName ] {  
        }  
    }  
}
```

```
Method [ <internal> public method getValue ] {
}

Method [ <internal> public method setValue ] {
}

Method [ <internal> public method isPublic ] {
}

Method [ <internal> public method isPrivate ] {
}

Method [ <internal> public method isProtected ] {
}

Method [ <internal> public method isStatic ] {
}

Method [ <internal> public method isDefault ] {
}

Method [ <internal> public method getModifiers ] {
}

Method [ <internal> public method getDeclaringClass ] {
}
}
}
```

As you can see, this function lists all necessary information about the class, such as methods and their signatures, properties, and constants.

#### 4.5.4 Implementing the Delegation Pattern Using Reflection

Times arise where a class (*One*) is supposed to do everything another class (*Two*) does and more. The preliminary temptation would be for class *One* to extend class *Two*, and thereby inheriting all of its functionality. However, there are times when this is the wrong thing to do, either because there isn't a clear semantic is-a relationship between classes *One* and *Two*, or class *One* is already extending another class, and inheritance cannot be used. Under such circumstances, it is useful to use a delegation model (via the **delegation design pattern**), where method calls that class *One* can't handle are redirected to class *Two*. In some cases, you may even want to chain a larger number of objects where the first one in the list has highest priority.

The following example creates such a delegator called `ClassOneDelegator` that first checks if the method exists and is accessible in `ClassOne`; if not, it tries all other objects that are registered with it. The application can register

additional objects that should be delegated to by using the `addObject($obj)` method. The order of adding the objects is the order of precedence when `ClassOneDelegator` searches for an object that can satisfy the request:

```
class ClassOne {
    function callClassOne() {
        print "In Class One\n";
    }
}

class ClassTwo {
    function callClassTwo() {
        print "In Class Two\n";
    }
}

class ClassOneDelegator {
    private $targets;

    function __construct() {
        $this->target[] = new ClassOne();
    }

    function addObject($obj) {
        $this->target[] = $obj;
    }

    function __call($name, $args) {
        foreach ($this->target as $obj) {
            $r = new ReflectionClass($obj);

            if ($method = $r->getMethod($name)) {
                if ($method->isPublic() && !$method->isAbstract()) {
                    return $method->invoke($obj, $args);
                }
            }
        }
    }
}

$obj = new ClassOneDelegator();
$obj->addObject(new ClassTwo());
$obj->callClassOne();
$obj->callClassTwo();
```

Running this code results in the following output:

```
In Class One
In Class Two
```

You can see that this example uses the previously described feature of overloading method calls using the special `__call()` method. After the call is intercepted, `__call()` uses the reflection API to search for an object that can satisfy the request. Such an object is defined as an object that has a method with the same name, which is publicly accessible and is not an abstract method.

Currently, the code does nothing if no satisfying function is found. You may want to call `classOne` by default, so that you make PHP error out with a nice error message, and in case `classOne` has itself defined a `__call()` method, it would be called. It is up to you to implement the default case in a way that suits your needs.

## 4.6 SUMMARY

This chapter covered the more advanced object-oriented features of PHP, many of which are critical when implementing large-scale OO applications. Thanks to the advances of PHP 5, using common OO methodologies, such as design patterns, has now become more of a reality than with past PHP versions. For further reading, we recommend additional material on design patterns and OO methodology. A good starting point is [www.cetus-links.org](http://www.cetus-links.org), which keeps an up-to-date list of good starting points. Also, we highly recommend reading the classic book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides.