OVERVIEW OF CLASSES, OBJECTS AND INTERFACES:

CLASS:

Our everyday environment consists of countless entities: plants, people, vehicles, food...I could go on for hours just listing them. Each entity is defined by a particular set of characteristics and behaviors that ultimately serves to define the entity for what it is.

In the vocabulary of OOP, such an embodiment of an entity's defining attributes and behaviors is known as a class. Classes are intended to represent those real-life items that you'd like to manipulate within an application.

PHP's generalized class creation syntax follows:

```
class Class_Name
{
// Field declarations defined here
// Method declarations defined here
}
```

Example:

```
class Employee
{
var $name;
function myfunc($propName, $propValue)
{
echo "Name is :".$propName." title is :".$propValue;
}
}
```

Titled Employee, this class defines three fields, name, title, and wage, in addition to two methods, clockIn and clockOut.

OBJECT:

An object is basically a type of data that allows you to group data and functions under a single variable. In PHP, the -> operator denotes a member of a specific object, that is, a property (piece of data) or method (function) belonging to an object. You can think of it as meaning "has a," with the arrow pointing from the thing doing the having to the thing being had.

Example:

```
<?php
class Employee
{
var $name;
function myfunc($propName, $propValue)
{
echo "Name is :".$propName." title is :".$propValue;
}
}
$employee = new Employee();
```

$employee->name = "Mario";

$employee->title = "Executive Chef";

$employee->myfunc($employee->name, $employee->title);

?>

The output of the previous code is as follows:

Name is: Mario title is :Executive Chef

Where objects really become useful is when you have a way to create them on demand from a single pattern.

A class defines a reusable template from which you can create as many similar objects (instances of the class) as you need.

For example, if you are writing an inventory system for a pet shop, you are likely to be working with information about lots of birds, and you can write a Bird class to represent a generic bird; each instance of this class then represents an individual bird. You can think of the class as a collection of variables and functions common to all birds. Variables attached to an object in this way are properties, and functions manipulating these variables are methods. Together, the methods and properties of an object are its members.

Note: A class is a template or prototype object. An instance of this class represents a particular case of this class. The word object can apply to classes and instances of classes alike.

A class provides a basis from which you can create specific instances of the entity the class models, better known as objects. For example, an employee management application may include an Employee class. You can then call upon this class to create and maintain specific instances, Sally and Jim, for example.

Objects are created using the *new* keyword, like this:

$employee = new Employee();

Once the object is created, all of the characteristics and behaviors defined within the class are made available to the newly instantiated object.

Fields:

Fields are attributes that are intended to describe some aspect of a class. They are quite similar to standard PHP variables, except for a few minor differences.

Declaring Fields:

The rules regarding field declaration are quite similar to those in place for variable declaration; essentially, there are none. Because PHP is a loosely typed language, fields don't even necessarily need to be declared; they can simply be created and assigned simultaneously by a class object, although you'll rarely want to do that. Instead, common practice is to declare fields at the beginning of the class. Optionally, you can assign them initial values at this time.

Example:

class Employee

{

public $name = "John";

private $wage;

}

In this example, the two fields, name and wage, are prefaced with a scope descriptor (public or private), a common practice when declaring fields.

Invoking Fields:

Fields are referred to using the -> operator and, unlike variables, are not prefaced with a dollar sign. Furthermore, because a field's value typically is specific to a given object, it is correlated to that object like this:

$object->field

For example, the Employee class includes the fields name, title, and wage. If you create an object named $employee of type Employee, you would refer to these fields like this:

$employee->name

$employee->title

$employee->wage

When you refer to a field from within the class in which it is defined, it is still prefaced with the → operator, although instead of correlating it to the class name, you use the $this keyword. $this implies that you're referring to the field residing in the same class in which the field is being accessed or manipulated. Therefore, if you were to create a method for setting the name field in the Employee class, it might look like this:

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

Field Scopes:

PHP supports five class field scopes: public, private, protected, final, and static.

Public: You can declare fields in the public scope by prefacing the field with the keyword public.

Example:

```
class Employee
{
public $name;
// Other field and method declarations follow...
}
```

Public fields can then be manipulated and accessed directly by a corresponding object, like so:

```
$employee = new Employee();
$employee->name = "Mary Swanson";
$name = $employee->name;
echo "New employee: $name";
```

Executing this code produces the following:

New employee: Mary Swanson

Private: Private Fields are only accessible from within the class in which they are defined.

Example:

```
class Employee
{
private $name;
private $telephone;
```

}

Fields designated as private are not directly accessible by an instantiated object, nor are they available to subclasses.

Consider the following example, in which a private field is manipulated by a public method:

class Employee
{
private $name;
public function setName($name) {
$this->name = $name;
}
}
$staff = new Employee;
$staff->setName("Mary");

Protected: Just like functions often require variables intended for use only within the function, classes can include fields used for solely internal purposes. Such fields are deemed protected and are prefaced accordingly.

Example:

class Employee
{
protected $wage;
}

Protected fields are also made available to inherited classes for access and manipulation, a trait not shared by private fields. Any attempt by an object to access a protected field will result in a fatal error. Therefore, if you plan on extending the class, you should use protected fields in lieu of private fields.

Final: Marking a field as final prevents it from being overridden by a subclass. A finalized field is declared like so:

class Employee
{
final $ssn;
}

Note: You can also declare methods as final.

Properties: Properties are a particularly convincing example of the powerful features OOP has to offer, ensuring protection of fields by forcing access and manipulation to take place through methods, yet allowing the data to be accessed as if it were a public field. These methods, known as *accessors* and *mutators*, or more informally as *getters* and *setters*, are automatically triggered whenever the field is accessed or manipulated, respectively.

Unfortunately, PHP does not offer the property functionality that you might be used to if you're familiar with other OOP languages such as C++ and Java. Therefore, you'll need to make do with using public methods to imitate such functionality. For example, you might create getter and setter methods for the property name by declaring two functions, getName() and setName(), respectively, and embedding the appropriate syntax within each.

Setting Properties: The mutator, or setter method is responsible for both hiding property assignment implementation and validating class data before assigning it to a class field. Its prototype follows:

boolean __set([string property name],[mixed value_to_assign])

It takes as input a property name and a corresponding value, returning TRUE if the method is successfully executed, and FALSE otherwise.

Example:

```
class Employee
{
var $name;
function __set($propName, $propValue)
{
echo "Nonexistent variable: \$$propName!";
}
$employee = new Employee ();
$employee->name = "Mario";
$employee->title = "Executive Chef";
```

This results in the following output:

Nonexistent variable: $title!

Getting Properties: The accessor, or mutator method, is responsible for encapsulating the code required for retrieving a class variable. Its prototype follows:

boolean __get([string property name])

It takes as input one parameter, the name of the property whose value you'd like to retrieve. It should return the value TRUE on successful execution, and FALSE otherwise.

Example:

```
class Employee
{
var $name;
var $city;
protected $wage;
function __get($propName)
{
echo "__get called!<br />";
$vars = array("name","city");
if (in_array($propName, $vars))
{
return $this->$propName;
} else {
return "No such variable!";
}
}
}
```

```
$employee = new Employee();
$employee->name = "Mario";
echo $employee->name."<br />";
echo $employee->age;
```

This returns the following:

```
Mario
__get called!
No such variable!
```

Creating Custom Getters and Setters: Frankly, although there are some benefits to the __set() and __get() methods, they really aren't sufficient for managing properties in a complex object-oriented application. Because PHP doesn't offer support for the creation of properties in the fashion that Java or C# does, you need to implement your own methodology. Consider creating two methods for each private field, like so:

```
<?php
class Employee
{
private $name;
// Getter
public function getName() {
return $this->name;
}
// Setter
public function setName($name) {
$this->name = $name;
}
}
?>
```

Constants: You can define constants, or values that are not intended to change, within a class. These values will remain unchanged throughout the lifetime of any object instantiated from that class. Class constants are created like so:

```
const NAME = 'VALUE';
```

Example: Suppose you create a math-related class that contains a number of methods defining mathematical functions, in addition to numerous constants:

```
class math_functions
{
const PI = '3.14159265';
const E = '2.7182818284';
const EULER = '0.5772156649';
// define other constants and methods here...
}
```

Class constants can then be called like this:

```
echo math_functions::PI;
```

Methods: A method is quite similar to a function, except that it is intended to define the behavior of a particular class. Like a function, a method can accept arguments as input and can return a value to the caller. Methods are also invoked like functions, except that the method is prefaced with the name of the object invoking the method, like this:

$object->method_name();

Declaring Methods: Methods are created in exactly the same fashion as functions but the only difference between methods and normal functions is, the method declaration is typically prefaced with a scope descriptor. The generalized syntax follows:

scope function functionName()
{
// Function body goes here
}

Example: Let us consider a public method titled calculateSalary() might look like this:

```
public function calculateSalary()
{
return $this->wage * $this->hours;
}
```

In this example, the method is directly invoking two class fields, wage and hours, using the $this keyword. It calculates a salary by multiplying the two field values together and returns the result just like a function might. Note, however, that a method isn't confined to working solely with class fields; it's perfectly valid to pass in arguments in the same way you can with a function.

Invoking Methods: Methods are invoked in almost exactly the same fashion as functions, continuing with the previous example, the calculateSalary() method would be invoked like so:

```
$employee = new Employee("Janie");
$salary = $employee->calculateSalary();
```

Method Scopes: PHP supports six method scopes: public, private, protected, abstract, final, and static.

Public: Public methods can be accessed from anywhere at any time. You declare a public method by prefacing it with the keyword public or by forgoing any prefacing whatsoever.

Example:

```
<?php
class Visitors
{
public function greetVisitor()
{
echo "Hello<br />";
}
function sayGoodbye()
{
echo "Goodbye<br />";
}
}
```

```
Visitors::greetVisitor();
$visitor = new Visitors();
$visitor->sayGoodbye();
?>
```

The following is the result:

Hello
Goodbye

Private: Methods marked as private are available for use only within the originating class and cannot be called by the instantiated object, nor by any of the originating class's subclasses. Methods solely intended to be helpers for other methods located within the class should be marked as private.

Example:

```
private function validateCardNumber($number)
{
if (! ereg('^([0-9]{4})-([0-9]{3})-([0-9]{2})') ) return FALSE;
else return TRUE;
}
```

Attempts to call this method from an instantiated object result in a fatal error.

Protected: Class methods marked as protected are available only to the originating class and its subclasses.

Example:

```
<?php
class Employee
{
private $ein;
function __construct($ein)
{
if ($this->verifyEIN($ein)) {
echo "EIN verified. Finish";
}
}
protected function verifyEIN($ein)
{
return TRUE;
}
}
$employee = new Employee("123-45-6789");
?>
```

Attempts to call verifyEIN() from outside of the class will result in a fatal error because of its protected scope status.

Abstract: Abstract methods are special in that they are declared only within a parent class but are implemented in child classes. Only classes declared as abstract can contain abstract methods. You might declare an abstract method if you want to define an application programming

interface (API) that can later be used as a model for implementation. Abstract methods are declared like this:

abstract function methodName();

Example:

```
abstract class Employee
{
abstract function hire();
abstract function fire();
abstract function promote();
abstract demote();
}
```

Final: Marking a method as final prevents it from being overridden by a subclass. A finalized method is declared like this:

```
class Employee
{
...
final function getName() {
...
}
}
```

Attempts to later override a finalized method result in a fatal error. PHP supports six method scopes: public, private, protected, abstract, final, and static.

INTERFACE:

An interface is a collection of unimplemented method definitions and constants that serves as a class blueprint. Interfaces define exactly what can be done with the class, without getting bogged down in implementation-specific details.

An interface defines a general specification for implementing a particular service, declaring the required functions and constants without specifying exactly how it must be implemented. Implementation details aren't provided because different entities might need to implement the published method definitions in different ways. The point is to establish a general set of guidelines that must be implemented in order for the interface to be considered implemented.

In PHP, an interface is created like so:

```
interface IinterfaceName
{
CONST 1;
...
CONST N;
function methodName1();
...
function methodNameN();
}
```

The following is the general syntax for implementing the preceding interface:

```
class Class_Name implements interfaceName
{
function methodName1()
{
// methodName1() implementation
}
function methodNameN()
{
// methodName1() implementation
}
}
```

Implementing a Single Interface:

```
interface IPillage
{
function emptyBankAccount();
function burnDocuments();
}
```

This interface is then implemented for use by the Executive class:

```
class Executive extends Employee implements IPillage
{
private $totalStockOptions;
function emptyBankAccount()
{
echo "Call CFO and ask to transfer funds to Swiss bank account.";
}
function burnDocuments()
{
echo "Torch the office suite.";
}
}
```

Implementing Multiple Interfaces:

```
<?php
interface IEmployee {...}
interface IDeveloper {...}
interface IPillage {...}
class Employee implements IEmployee, IDeveloper, iPillage {
...
}
class Contractor implements IEmployee, IDeveloper {
...
}
?>
```

CREATING INSTANCES USING CONSTRUCTORS:

To create a new instance of a class (also referred to as instantiating a class), you can use the new operator in conjunction with the class name called as though it were a function.

When used in this way, it acts as what is known as the class constructor and serves to initialize the instance.

This instance is represented by a variable and is subject to the usual rules governing variables and identifier names in PHP. For example, consider the following:

```
$tweety = new Bird('Tweety', 'canary');
```

This creates a specific instance of Bird and assigns it to the variable named $tweety. In other words, you have defined $tweety as a Bird.

Example:

```
<?php
class Bird
{
function __construct($name, $breed)
{
$this->name = $name;
$this->breed = $breed;
} }
?>
```

The $this keyword has a special purpose: it allows you to refer to the instance from within the class definition. It works as a placeholder and means, “the current instance of this class.” The Bird class constructor assigns the string 'Tweety' to the name property of the instance you are creating and the string 'canary' to its breed property. You can put this to the test like so:

```
<?php
class Bird
{
function __construct($name, $breed)
{
$this->name = $name;
$this->breed = $breed;
}       }
$tweety = new Bird('Tweety', 'canary');
printf("<p>%s is a %s.</p>\n", $tweety->name, $tweety->breed);
?>
```

The resulting output is as follows:

Tweety is a canary.

To determine the price you want to charge for Tweety, you could set the price property and output it like so:

```
<?php
// ...class defined and constructor called as previously shown...
$tweety->price = 24.95;
printf("<p>%s is a %s and costs \$%.2f.</p>\n",
```

```
$tweety->name, $tweety->breed, $tweety->price);
?>
```

The output from this is as follows:

Tweety is a canary and costs $24.95.

CONTROLLING ACCESS TO CLASS MEMBERS:

With the help of an example we will explain how to control the access to class members.

The Code:

```
<?php
// file bird-get-set.php
class Bird
{
function __construct($name='No-name', $breed='unknown', $price = 15)
{
$this->name = $name;
$this->breed = $breed;
$this->price = $price;
}
function setName($name)
{
$this->name = $name;
}
function setBreed($breed)
{
$this->breed = $breed;
}
```

Notice that we have written the setPrice() method in such a way that the price cannot be set to a negative value; if a negative value is passed to this method, the price will be set to zero.

```
function setPrice($price)
{
$this->price = $price < 0 ? 0 : $price;
}
function getName()
{
return $this->name;
}
function getBreed()
{
return $this->breed;
}
function getPrice()
{
return $this->price;
}
```

To save some repetitive typing of the printf() statement that you have been using to output all the information you have about a given Bird object, you can add a new method named display() that takes care of this task:

```
function display()
{
printf("<p>%s is a %s and costs \$%.2f.</p>\n",
$this->name, $this->breed, $this->price);
}
}
```

CONSTRUCTORS AND DESTRUCTORS:

Constructors: You often want to initialize certain fields and even trigger the execution of methods found when an object is newly instantiated. There's nothing wrong with doing so immediately after instantiation, but it would be easier if this were done for you automatically.

Such a mechanism exists in OOP, known as a constructor. Quite simply, a constructor is defined as a block of code that automatically executes at the time of object instantiation. OOP constructors offer a number of advantages:

1. Constructors can accept parameters, which are assigned to specific object fields at creation time.
2. Constructors can call class methods or other functions.
3. Class constructors can call on other constructors, including those from the class parent.

PHP recognizes constructors by the name __construct. The general syntax for constructor declaration follows:

```
function __construct([argument1, argument2, ..., argumentN])
{
// Class initialization code
}
```

Example:

```
<?php
class Book
{
private $title;
private $isbn;
private $copies;
public function _construct($isbn)
{
$this->setIsbn($isbn);
$this->getTitle();
$this->getNumberCopies();
}
public function setIsbn($isbn)
{
$this->isbn = $isbn;
}
```

```
public function getTitle() {
$this->title = "Beginning Python";
print "Title: ".$this->title."<br />";
}
public function getNumberCopies() {
$this->copies = "5";
print "Number copies available: ".$this->copies."<br />";
}
}
$book = new book("159059519X");
?>
```

This results in the following:

```
Title: Beginning Python
Number copies available: 5
```

Invoking Parent Constructors: PHP does not automatically call a parent constructor instead you can call it explicitly using the keyword parent.

Example:

```
<?php
class Employee
{
protected $name;
protected $title;
function __construct()
{
echo "<p>Staff constructor called!</p>";
}
}
class Manager extends Employee
{
function __construct()
{
parent::__construct();
echo "<p>Manager constructor called!</p>";
}
}
$employee = new Manager();
?>
```

This results in the following:

```
Employee constructor called!
Manager constructor called!
```

Invoking Unrelated Constructors: You can invoke class constructors that don't have any relation to the instantiated object simply by prefacing __constructor with the class name, like so:

```
classname::__construct()
```

Destructors:

Although objects were automatically destroyed upon script completion in PHP 4, it wasn't possible to customize this cleanup process. With the introduction of destructors in PHP 5, this constraint is no more. Destructors are created like any other method but must be titled __destruct().

Example:

```
<?php
class Book
{
private $title;
private $isbn;
private $copies;
function __construct($isbn)
{
echo "<p>Book class instance created.</p>";
}
function __destruct()
{
echo "<p>Book class instance destroyed.</p>";
}
}
$book = new Book("1893115852");
?>
```

Here's the result:

```
Book class instance created.
Book class instance destroyed.
```

When the script is complete, PHP will destroy any objects that reside in memory. Therefore, if the instantiated class and any information created as a result of the instantiation reside in memory, you're not required to explicitly declare a destructor. However, if less volatile data is created (say, stored in a database) as a result of the instantiation and should be destroyed at the time of object destruction, you'll need to create a custom destructor.

Static Class Members: Sometimes it's useful to create fields and methods that are not invoked by any particular object but rather are pertinent to and are shared by all class instances.

For example, suppose that you are writing a class that tracks the number of Web page visitors. You wouldn't want the visitor count to reset to zero every time the class is instantiated, and therefore you would set the field to be of the static scope:

```
<?php
class Visitor
{
private static $visitors = 0;
function __construct()
{
self::$visitors++;
}
```

```
static function getVisitors()
{
return self::$visitors;
}
}
/* Instantiate the Visitor class. */
$visits = new Visitor();
echo Visitor::getVisitors()."<br />";
/* Instantiate another Visitor class. */
$visits2 = new Visitor();
echo Visitor::getVisitors()."<br />";
?>
```

The results are as follows:

```
1
2
```

Because the $visitors field was declared as static, any changes made to its value are reflected across all instantiated objects. Also note that static fields and methods are referred to using the self keyword and class name, rather than via $this and arrow operators. This is because referring to static fields using the means allowed for their "regular" siblings is not possible and will result in a syntax error if attempted.

The instanceof Keyword: It was introduced with PHP 5. With it you can determine whether an object is an instance of a class, is a subclass of a class, or implements a particular interface, and do something accordingly. For example, suppose you want to learn whether an object called manager is derived from the class Employee:

```
$manager = new Employee();
...
if ($manager instanceof Employee) echo "Yes";
```

There are two points worth noting here. First, the class name is not surrounded by any sort of delimiters (quotes). Including them will result in a syntax error. Second, if this comparison fails, the script will abort execution. The instanceof keyword is particularly useful when you're working with a number of objects simultaneously.

EXTENDING CLASSES:

Extending classes is useful when you have multiple objects that have some but not all properties or methods in common. Rather than write a separate class for each object that duplicates the members that are common to all, you can write a generic class that contains these common elements, extend it with subclasses that inherit the common members, and then add those that are specific to each subclass.

The following is some PHP 5 code that implements these three classes. Bird has three properties ($name, $price, and $breed), all of which are private. You can set the first two of these with the public methods setName() and setPrice(), respectively, or in the class constructor.

You can set the breed only from the Bird class constructor; because the setBreed() method is private, it can be called only from within Bird, not from any other code. Since $breed has no default value, you will receive a warning if you do not set it in the constructor.

The Code:

```
<?php
// file: bird-multi.php
// example classes for inheritance example
class Bird
{
private $name;
private $breed;
private $price;
public function __construct($name, $breed, $price=15)
{
$this->setName($name);
$this->setBreed($breed);
$this->setPrice($price);
}
public function setName($name)
{
$this->name = $name;
}
private function setBreed($breed)
{
$this->breed = $breed;
}
public function setPrice($price)
{
$this->price = $price;
}
```

All the get methods of this class are public, which means you can call them at any time from within the Bird class, from within any subclasses of Bird that you might create, and from any instance of Bird or a Bird subclass. The same is true for the display() and birdCall() methods.

```
public function getName()
{
return $this->name;
}
public function getBreed()
{
return $this->breed;
}
public function getPrice()
{
return $this->price;
}
```

Each bird makes some sort of sound. Unless you override the birdCall() method in a subclass, you assume that the bird chirps.

Inheritance: People are quite adept at thinking in terms of organizational hierarchies; thus, it doesn't come as a surprise that we make widespread use of this conceptual view to manage many aspects of our everyday lives.

The characteristics and behaviors would be relevant to all types of employees, regardless of the employee's purpose or stature within the organization. Obviously, though, there are also differences among employees; for example, the executive might hold stock options and be able to pillage the company, while other employees are not afforded such luxuries. An assistant must be able to take a memo, and an office manager needs to take supply inventories.

Despite these differences, it would be quite inefficient if you had to create and maintain redundant class structures for those attributes that all classes share. The OOP development paradigm takes this into account, allowing you to inherit from and build upon existing classes.

Class Inheritance:

As applied to PHP, class inheritance is accomplished by using the keyword *extends*. The following example demonstrates this ability, first creating an Employee class and then creating an Executive class that inherits from Employee.

Inheriting from a Base Class:

```
<?php
// Define a base Employee class
class Employee {
private $name;
// Define a setter for the private $name member.
function setName($name) {
if ($name == "") echo "Name cannot be blank!";
else $this->name = $name;
}
// Define a getter for the private $name member
function getName() {
return "My name is ".$this->name."<br />";
}
} // end Employee class
// Define an Executive class that inherits from Employee
class Executive extends Employee {
// Define a method unique to Employee
function pillageCompany() {
echo "I'm selling company assets to finance my yacht!";
}
} // end Executive class
// Create a new Executive object
$exec = new Executive();
// Call the setName() method, defined in the Employee class
$exec->setName("Richard");
// Call the getName() method
echo $exec->getName();
// Call the pillageCompany() method
$exec->pillageCompany();
```

```
?>
```

This returns the following:

```
My name is Richard.
I'm selling company assets to finance my yacht!
```

Because all employees have a name, the Executive class inherits from the Employee class, saving you the hassle of having to re-create the name member and the corresponding getter and setter.

You can then focus solely on those characteristics that are specific to an executive, in this case a method named pillageCompany(). This method is available solely to objects of type Executive, and not to the Employee class or any other class, unless of course you create a class that inherits from Executive. *The following example demonstrates that concept, producing a class titled CEO*, which inherits from Executive:

```
<?php
class Employee {
...
}
class Executive extends Employee {
...
}
class CEO extends Executive {
function getFacelift() {
echo "nip nip tuck tuck";
}
}
$ceo = new CEO();
$ceo->setName("Bernie");
$ceo->pillageCompany();
$ceo->getFacelift();
?>
```

Because Executive has inherited from Employee, objects of type CEO also have all the members and methods that are available to Executive, in addition to the getFacelift() method, which is reserved solely for objects of type CEO.

Inheritance and Constructors:

If a parent class offers a constructor, it does execute when the child class is instantiated, provided that the child class does not also have a constructor. For example, suppose that the Employee class offers this constructor:

```
function __construct($name) {
$this->setName($name);
}
```

Then you instantiate the CEO class and retrieve the name member:

```
$ceo = new CEO("Dennis");
echo $ceo->getName();
```

It will yield the following:

My name is Dennis

However, if the child class also has a constructor, that constructor will execute when the child class is instantiated, regardless of whether the parent class also has a constructor. For example, suppose that in addition to the Employee class containing the previously described constructor, the CEO class contains this constructor:

```
function __construct() {
echo "<p>CEO object created!</p>"; }
```

Then you instantiate the CEO class:

```
$ceo = new CEO("Dennis");
echo $ceo->getName();
```

This time it will yield the following output because the CEO constructor overrides the Employee constructor:

```
CEO object created!
My name is
```

When it comes time to retrieve the name member, you find that it's blank because the setName() method, which executes in the Employee constructor, never fires. Of course, you're quite likely going to want those parent constructors to also fire. Not to fear because there is a simple solution. Modify the CEO constructor like so:

```
function __construct($name) {
parent::__construct($name);
echo "<p>CEO object created!</p>";
}
```

Again instantiating the CEO class and executing getName() in the same fashion as before, this time you'll see a different outcome:

```
CEO object created!
My name is Dennis
```

You should understand that when parent::__construct() was encountered, PHP began a search upward through the parent classes for an appropriate constructor.

USING ABSTRACT CLASSES AND METHODS:

An abstract method is one that is declared by name only, with the details of the implementation left up to a derived class. You should remember three important facts when working with class abstraction:

- Any class that contains one or more abstract methods must itself be declared as abstract.
- An abstract class cannot be instantiated; you must extend it in another class and then create instances of the derived class. Put another way, only concrete classes can be instantiated.
- A class that extends the abstract class must implement the abstract methods of the parent class or itself be declared as abstract.

Let's update the Bird class so that its birdCall() method is abstract. We will not repeat the entire class listing here—only two steps are necessary to modify Bird. The first step is to replace the method declaration for Bird::birdCall() with the following:

```
abstract public function birdCall();
```

An abstract method has no method body; it consists solely of the abstract keyword followed by the visibility and name of the function, the function keyword, a pair of parentheses, and a semicolon. What this line of code says in plain English is, “Any class derived from this one must include a birdCall() method, and this method must be declared as public.”

The second step is to modify the class declaration by prefacing the name of the class with the abstract keyword, as shown here:

abstract class Bird

The Code:

```
public function birdCall($singing=FALSE)
{
$sound = $singing ? "twitter" : "chirp";
printf("<p>%s says: *%s*</p>\n", $this->getName(), $sound);
}
```

Abstract Classes

An abstract class is a class that really isn’t supposed to ever be instantiated but instead serves as a base class to be inherited by other classes. A class is declared abstract by prefacing the definition with the word abstract, like so:

```
abstract class Class_Name
{
// insert attribute definitions here
// insert method definitions here
}
```

Attempting to instantiate an abstract class results an error. Abstract classes ensure conformity because any classes derived from them must implement all abstract methods derived within the class. Attempting to forgo implementation of any abstract method defined in the class results in a fatal error.

Abstract methods: Abstract methods are special in that they are declared only within a parent class but are implemented in child classes. Only classes declared as abstract can contain abstract methods. You might declare an abstract method if you want to define an application programming interface (API) that can later be used as a model for implementation. Abstract methods are declared like this:

abstract function methodName();

Example:

```
abstract class Employee
{
abstract function hire();
abstract function fire();
abstract function promote();
abstract demote();
}
```

USING INTERFACES:

As we know that abstract classes and methods allow you to declare some of the methods of a class but defer their implementation to subclasses. But what happens if you write a class that has all abstract methods? The answer to this question is: what you end up with is just one step removed from an interface.

You can think of an interface as a template that tells you what methods a class should expose but leaves the details up to you. Interfaces are useful in that they can help you plan your classes without immediately getting bogged down in the details. You can also use them to distill the essential functionality from existing classes when it comes time to update and extend an application.

To declare an interface, simply use the ***interface*** keyword, followed by the name of the interface.

The Code:

Looking at the Bird class, you might deduce that you are really representing two different sorts of functional units: a type of animal (which has a name and a breed) and a type of product (which has a price). Let's generalize these into two interfaces, like so:

```
interface Pet
{
public function getName();
public function getBreed();
}
interface Product
{
public function getPrice();
}
```

To show that a class implements an interface, you add the implements keyword plus the name of the interface to the class declaration. One advantage that interfaces have over abstract classes is that a class can implement more than one interface, so if you wanted to show that Bird implements both Pet and Product, you would simply rewrite the class declaration for Bird, as shown here:

```
abstract class Bird implements Pet, Product
```

Note: In PHP 5, interfaces may declare only methods. An interface cannot declare any variables.

Using interfaces can help you keep your classes consistent with one another. For example, if you need to write classes to represent additional pets for sale by the pet store, you can, by implementing Pet and Product in those classes, guarantee that they will have the same methods that Bird and its subclasses do.

USING CLASS DESTRUCTORS:

In PHP 5, classes can have destructors as well as constructors. A destructor is simply a method that is guaranteed to be invoked whenever an instance of the class is removed from memory, either as the result of a script ending or because of a call to the unset() function. For example, suppose that when a user of your e-commerce website—represented by an instance of a SiteUser class—leaves the site, you want to make sure that all the user's preference data is saved to the site's user database. Suppose further that SiteUser already has a savePrefs() method that accomplishes this task; you just need to make sure it is called when the user logs out. In that case, the class listing might include something like the following.

The Code:

```
class SiteUser
{
// class variables...
public function __construct()
{
// constructor method code...
}
// other methods...
public function savePref()
{
// code for saving user preferences...
}
// Here's the class destructor:
public function __destruct()
{
$this->savePrefs();
}
}
```

As you can see from this listing, all you need to do is to add a __destruct() method to the class containing whatever code you want to be executed when an instance of the class ceases to exist.

USING EXCEPTIONS:

PHP 5 introduces a much-improved mechanism for handling errors. The purpose of exceptions is to help segregate error-handling code from the parts of your application that are actually doing the work. A typical situation is working with a database.

The following is a bit of code showing how you might do this in PHP 4 or how you might do this in PHP 5 without using exceptions:

```
<?php
$connection = mysql_connect($host, $user, $password)
or die("Error #". mysql_errno() .": " . mysql_error() . ".");
mysql_select_db($database, $connection)
or die("Error: could not select database $database on host $hostname.");
$query = "SELECT page_id, link_text, parent_id FROM menus WHERE page_id='$pid'";
$result = mysql_query($query)
or die("Query failed: Error #". mysql_errno() .": " . mysql_error() . ".");
if(mysql_num_rows($result) == 0)
echo "<h2>Invalid page request -- click <a href=\"". $_SERVER["PHP_SELF"] . "?pid=1\">here</a> to continue.</h2>\n";
else
{
$value = mysql_fetch_object($result)
or die("Fetch operation failed: Error #". mysql_errno(). ": " . mysql_error() . ".");
// ...
```

```
}
// etc. ...
?>
```

Notice that every time you interact with the database, you include an explicit error check.

This is good in that you are practicing defensive programming and not leaving the user with a blank page or half-completed page in the event of an error. However, it is not so good in that the error checking is mixed up with the rest of the code. Using PHP 5 exception handling, the same block might look something like the following example.

The Code:

```
<?php
function errors_to_exceptions($code, $message)
{
throw new Exception($code, $message);
}
set_error_handler('errors_to_exceptions');
try
{
$connection = mysql_connect($host, $user, $password);
mysql_select_db($database, $connection);
$query = "SELECT page_id, link_text, parent_id FROM menus WHERE page_id='$pid'";
$result = mysql_query($query);
if(mysql_num_rows($result) == 0)
echo "<h2>Invalid page request -- click <a href=\"". $_SERVER["PHP_SELF"] . "?pid=1\">here</a>
to continue.</h2>\n";
else
{
$value = mysql_fetch_object($result);
// ...
}
// etc. ...
}
catch Exception $e
{
printf("<p>Caught exception: %s.</p>\n", $e->getMessage());
}
?>
```

The basic structure for exception handling looks like this:

```
try
{
perform_some_action();
if($some_action_results_in_error)
throw new Exception("Houston, we've got a problem...");
perform_another_action();
if($other_action_results_in_error)
```

throw new Exception("Houston, we've got a different problem...");

}

catch Exception $e

{

handle_exception($e); }

The try block contains any code that is to be tested for exceptions. When an exception is thrown, either automatically or by using the ***throw*** keyword, script execution immediately passes to the next catch block. (If PHP cannot find any catch block following the try block, then it will issue an Uncaught Exception error.)

Some PHP object-oriented libraries and extensions supply their own exception classes. For example, the Document Object Model (DOM) extension implements a DOMException class and raises an instance of this class whenever an illegal DOM operation is attempted. When using a new PHP 5 class library for the first time, be sure to check whether it includes its own Exception subclasses.

FILE HANDLING:

Although a web page may appear as a nicely collected and displayed grouping of images and text, the actual underlying structure comprises a wide collection of files that are sorted amongst well-structured directories. Website file structures generally consist of specifically formatted files that run on a server-enabled computer.

PHP 5 is more than adept at opening, closing, reading, writing, and manipulating text files, including Hypertext Markup Language (HTML), JavaScript, and Extensible Markup Language (XML) files.

Opening Files:

The first task you must accomplish when working with a file (be it a .txt file, an .xml file, or another file) is to open the file.

The following example uses two pertinent file-related functions: file_exists() checks (relatively) for a file's existence, and fopen() attempts to open a file. The prototypes for the functions are as follows:

bool file_exists (string filename)

resource fopen (string fname, string mode [, bool useincpth [, resource zcontext]])

The Code:

<?php

//First, declare the file you want to open.

$file = "samplefile1.txt";

//Now, you use the file_exists() function to confirm its existence.

if (file_exists ($file)){

//Then you attempt to open the file, in this case for reading.

try {

if ($readfile = fopen ($file, "r")){

//Then you can work with the file.

echo "File opened successfully.";

} else {

//If it fails, you throw an exception.

throw new exception ("Sorry, the file could not be opened.");

```
}
} catch (exception $e) {
echo $e->getmessage();
}
} else {
echo "File does not exist.";
}
?>
```

This results the following:

File opened successfully.

PHP 5 Arguments for Opening a File

Argument	Description
r	Opens a file for reading
r+	Opens a file for both reading and writing
w	Opens a file for writing only
w+	Opens a file for reading and writing
a	Opens a file for appending (write-only)
a+	Opens a file for appending (read/write)
x	Creates a file and opens it for writing only
x+	Creates a file and opens it for reading and writing

Reading from Files:

PHP 5 has a nice way of reading from files. Not only can you open a file for reading, but you can also have PHP attempt to create the file for you and then open it. Created files are owned by the server by default (on Linux servers) and should thus be given proper permissions by the script to ensure that you (and your script) can access the file later.

The most common means of acquiring information from a text file is by using the functions ***fgetc()***, ***fgets()***, and ***fread()***. Each of these functions is similar in use but has its own separate strengths and weaknesses; the idea is to pick the right one for the job. The methods are as follows:

- string fgetc (resource handle)
- string fgets (resource handle [, int length])
- string fread (resource handle, int length)

The function fgetc() returns a character from a line in a file, fgets() returns a single line from a file, and fread() returns a requested amount of bytes worth of data from a file. Typically, if you are on the lookout for only one character at a time, use the fgetc() method. If you need access to the entire line of a file, use the fgets() function; should you need a specific set of information for a line in a file, use the fread() method.

The Code:

```
<?php
//First, declare the file you want to open.
$file = "samplefile1.txt";
```

```
//Now, you use the file_exists() function to confirm its existence.
if (file_exists ($file)){
//Then you attempt to open the file, in this case for reading.
try {
if ($readfile = fopen ($file, "r")){
//Then you can work with the file.
//Get the current value of the counter by using fread().
$curvalue = fread ($readfile, filesize($file));
//Then you can output the results.
echo $curvalue;
} else {
//If it fails, throw an exception.
throw new exception ("Sorry, the file could not be opened.");
}
} catch (exception $e) {
echo $e->getmessage();
}
} else {
echo "File does not exist.";
}
?>
```

The result would be a numerical value such as the following:

9

Writing to Files:

When working with files, you will also need to know how to make changes to them, that is, how to write to files. Similar with reading from files, you can write to files in a multitude of ways. The most common way of accomplishing this task is by using the function ***fwrite()***, but several methods exist:

- int fwrite (resource handle, string string [, int length])
- int file_put_contents (string filename, mixed data [, int flags [, resource context]])
- fputs (alias of fwrite())

Generally, the fwrite() function acts as a simple way to write something to a file and should be used if you are opening and closing a file using the fopen() and fclose() functions. The function file_put_contents() allows you to open, write to, and then close a file—all at the same time. Lastly, the function fputs() merely acts as an alias to fwrite() and can be used in the same manner.

Since writing to files is a little more sensitive than reading from them (because you wouldn't want any script to overwrite sensitive information), you have to perform a few more checks to ensure that you can actually write to the file in question. Thankfully, PHP 5 supports the function ***is_writable()***, which will return a boolean value that lets you know whether you can write to a file.

The Code:

```
<?php
//First, declare the file you want to open.
$file = "samplefile1.txt";
//Now, you use the file_exists() function to confirm its existence.
if (file_exists ($file)){
//Then you attempt to open the file, in this case for reading.
try {
if ($readfile = fopen ($file, "r")){
//Then you can work with the file.
//Get the current value of the counter by using fread().
$curvalue = fread ($readfile,filesize($file));
//Increment our counter by 1.
$curvalue++;
//Then attempt to open the file for writing, again validating.
if (is_writable ($file)){
try {
if ($writefile = fopen ($file, "w")){
//Then write the new value to the file.
fwrite ($writefile, $curvalue);
echo "Wrote $curvalue to file.";
} else {
throw new exception ("Sorry, the file could not be opened");
}
} catch (exception $e){
echo $e->getmessage();
}
} else {
echo "File could not be opened for writing";
}
} else {
//If it fails, you throw an exception.
throw new exception ("Sorry, the file could not be opened.");
}
} catch (exception $e) {
echo $e->getmessage();
}
} else {
echo "File does not exist.";
}
?>
```

This will result in the following:

```
Wrote 11 to file.
```

Closing Files:

The last thing you must do, once you have finished working with your file, is to clean up the mess. By this we mean that you must close the current file pointer.

Closing a file in PHP is just as simple as opening one; you simply need to invoke the ***fclose()*** method, which will close the file pointer link. This is the prototype for fclose():

bool fclose (resource handle)

The following example finalizes the counter code by outputting the value of the counter to the website and cleaning up your work with the fclose() function.

The Code:

```
<?php
//First, declare the file you want to open.
$file = "samplefile1.txt";
//Now, you use the file_exists() function to confirm its existence.
if (file_exists ($file)){
//Then you attempt to open the file, in this case for reading.
try {
if ($readfile = fopen ($file, "r")){
//Then you can work with the file.
//Get the current value of our counter by using fread().
$curvalue = fread ($readfile,filesize($file));
//Close the file since you have no more need to read.
fclose ($readfile);
//Increment the counter by 1.
$curvalue++;
//Then attempt to open the file for writing, and again, validating.
if (is_writable ($file)){
try {
if ($writefile = fopen ($file, "w")){
//Then write the new value to the file.
fwrite ($writefile, $curvalue);
//Close the file, as you have no more to write.
fclose ($writefile);
//Then lastly, output the counter.
echo $curvalue;
} else {
throw new exception ("Sorry, the file could not be opened");
}
} catch (exception $e){
echo $e->getmessage();
}
} else {
echo "File could not be opened for writing";
}
} else {
```

```
//If it fails, throw an exception.
throw new exception ("Sorry, the file could not be opened.");
}
} catch (exception $e) {
echo $e->getmessage();
}
} else {
echo "File does not exist.";
}        ?>
```

This results in the following:

12

Getting the Number of Lines in a File:

Finding the number of lines in a file is also quite handy. Luckily, with PHP it is also rather easy to accomplish. Because the *file()* function creates an array filled with each line in a separate index, you can simply read a file into an array and use ***count()*** to retrieve what is essentially the number of lines in a file.

The Code:

```
<?php
//First, dictate a file.
$afile = "samplefile5.txt";
//Now, you use the file_exists() function to confirm its existence.
if (file_exists ($afile)){
//Read it using the file() function.
$rows = file ($afile);
//You can then use the count() function to tell you the number of lines.
echo count ($rows) . " lines in this file"; //Outputs 4 in this case
} else {
echo "Sorry, file does not exist.";
}
?>
```

This results in the following:

4 lines in this file