developerWorks   Technical topics   Open source   Technical library

Build seven good object-oriented habits in PHP

Make your PHP applications better with object orientation

With PHP's object-oriented (OO) language features, if you aren't already creating your applications with OO principles in mind, these seven habits will help you get started in the transition between procedural programming and OO programming.

Share:

Nathan Good lives in the Twin Cities area of Minnesota. Professionally, he does software development, software architecture, and systems administration. When he's not writing software, he enjoys building PCs and servers, reading about and working with new technologies, and trying to get his friends to make the move to open source software. He's written and co-written many books and articles, including Professional Red Hat Enterprise Linux 3, Regular Expression Recipes: A Problem-Solution Approach, and Foundations of PEAR: Rapid PHP Development.

28 October 2008

Also available in [Russian](#) [Japanese](#) [Vietnamese](#) [Portuguese](#)

In the early days of PHP programming, PHP code was limited to being procedural in nature. *Procedural code* is characterized by the use of procedures for the building blocks of the application. Procedures offer a certain level of reuse by allowing procedures to be called by other procedures.

However, without object-oriented language constructs, a programmer can still introduce OO characteristics into PHP code. It's a tad more difficult and can make the code more difficult to read because it's mixing paradigms (procedural language with pseudo-OO design). OO constructs in PHP code — such as the ability to define and use classes, the ability to build relationships between classes that use inheritance, and the ability to define interfaces — make it much easier to build code that adheres to good OO practices.

While purely procedural designs without much modularity run just fine, the advantages of OO design show up in the maintenance. Because a typical application will spend the bulk of its lifetime in maintenance, code maintenance is a large expense over the lifetime of an application. It can also be easily forgotten during development. If you're in a race to get your application developed and deployed, long-term maintainability can take a back seat to getting something to work.

Modularity— one of the key characteristics of good OO design — helps with this maintenance. Modularity helps encapsulate change, which will make it easier to extend and modify the application over time.

While there are more than seven habits to building OO software overall, the seven habits here are what you need to make your code fit basic OO design criteria. They give you a firm foundation upon which you can add more OO habits and build software that is easily maintained and extended. These habits target a couple of the key characteristics of modularity. For more information about the benefits of OO design that are language-independent, see [Resources](#).
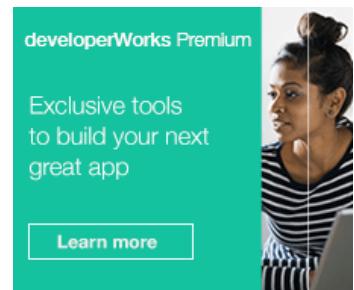
The seven good PHP OO habits are:

1. Be modest.
2. Be a good neighbor.
3. Avoid looking at Medusa.
4. Embrace the weakest link.
5. You're rubber; I'm glue.
6. Keep it in the family.
7. Think in patterns.

Be modest

To be modest is to avoid exposing yourself in your implementations of classes and functions. Hiding your information is a foundational habit. You will have a difficult time building any of the other habits until you have developed the habit of hiding the details of your implementations. Information-hiding is also known as *encapsulation.*

There are many reasons why exposing public fields directly is a bad habit, the most important of which is that it leaves you with no options should something in your implementation change. You use OO concepts to isolate change, and encapsulation plays an integral role in making sure that any changes you make aren't viral in nature. *Viral* changes are those that start small — like changing an array that holds three elements to one that contains only two. Suddenly, you find that you're changing more and more of your code to accommodate a change that should have been trivial.

One simple way to begin hiding your information is to keep fields private and to expose them with public accessors, which are like windows in your house. Instead of having an entire wall open to the outside, you have only a window or two. (I talk more about accessors in "Good habit: Use public accessors.")

In addition to allowing your implementation behind the curtain to change, using public accessors instead of directly exposing fields allows you to build upon your base implementation by overriding the implementation of an accessor to do something slightly different from the behavior of the parent. It also allows you to build an abstract implementation that defers the actual implementation to classes that override the base.

Bad habit: Expose public fields

In the bad code example in Listing 1, the fields of the `Person` object are exposed directly as public fields instead of with accessors. While this behavior is tempting, especially for lightweight data objects, it limits you.

Listing 1. Bad habit of exposing public fields

```php
<?php
class Person
{
    public $prefix;
    public $givenName;
    public $familyName;
    public $suffix;
}

$person = new Person();
$person->prefix = "Mr.";
$person->givenName = "John";

echo($person->prefix);
echo($person->givenName);

?>
```

If anything changes with the object, any code that uses it needs to change as well. For instance, if the person's given, family, and other names were to be encapsulated in a `PersonName` object, you would need to modify all your code to accommodate the change.

Good habit: Use public accessors

By using good OO habits (see Listing 2), the same object now has private fields instead of public fields, and the private fields are carefully exposed to the outside world by public `get` and `set` methods, called *accessors.* These accessors now provide a public way of getting information from your PHP class so that if something in your implementation changes, the likelihood is lessened that you need to change all the code that uses your class.

Listing 2. Good habit of using public accessors

```php
<?php
class Person
{
    private $prefix;
    private $givenName;
    private $familyName;
    private $suffix;

    public function setPrefix($prefix)
    {
        $this->prefix = $prefix;
    }

    public function getPrefix()
    {
        return $this->prefix;
    }

    public function setGivenName($gn)
    {
        $this->givenName = $gn;
    }

    public function getGivenName()
    {
        return $this->givenName;
    }

    public function setFamilyName($fn)
    {
        $this->familyName = $fn;
    }

    public function getFamilyName()
    {
        return $this->familyName;
    }

    public function setSuffix($suffix)
    {
        $this->suffix = $suffix;
    }

    public function getSuffix()
    {
        return $suffix;
    }

}

$person = new Person();
$person->setPrefix("Mr.");
$person->setGivenName("John");

echo($person->getPrefix());
echo($person->getGivenName());

?>
```

At first glance, this may seem like a lot more work, and it may actually be more work on the front end. Typically, however, using good OO habits pays off in the long run, as future changes are greatly solidified.

In the version of the code shown in Listing 3, I've changed the internal implementation to use an associative array for the name parts. Ideally, I'd have more error handling and would be more careful with checking whether the element exists, but the purpose of this example is to show how the code using my class

doesn't need to change — it is blissfully unaware of my class changes. Remember that the reason for adopting OO habits is to carefully encapsulate change so your code is more extensible and maintainable.

Listing 3. Another twist on this good habit with a different internal implementation

```php
<?php
class Person
{
    private $personName = array();

    public function setPrefix($prefix)
    {
        $this->personName['prefix'] = $prefix;
    }

    public function getPrefix()
    {
        return $this->personName['prefix'];
    }

    public function setGivenName($gn)
    {
        $this->personName['givenName'] = $gn;
    }

    public function getGivenName()
    {
        return $this->personName['givenName'];
    }

    /* etc... */
}

/*
 * Even though the internal implementation changed, the code here stays exactly
 * the same. The change has been encapsulated only to the Person class.
 */
$person = new Person();
$person->setPrefix("Mr.");
$person->setGivenName("John");

echo($person->getPrefix());
echo($person->getGivenName());

?>
```

Be a good neighbor

When you build a class, it should handle its own errors appropriately. If the class doesn't know how to handle the errors, it should package them in a format that its caller understands. In addition, avoid returning objects that are null or in an invalid state. Many times, you can do this simply by verifying arguments and throwing specific exceptions that tell why the supplied arguments are invalid. When you build this habit, it can save you — and people maintaining your code or using your objects — a lot of time.

Bad habit: Not handling errors

Consider the example shown in Listing 4, which accepts some arguments and returns a `Person` object with some of the values populated. However, in the `parsePersonName()` method, there is no validation to see whether the supplied `$val` variable is null, a zero-length string or perhaps a string in a unparsable format. The `parsePersonName()` method does not return a `Person` object, but returns null. Administrators or programmers using this method might be left scratching their heads and — at the very least — be in a place where they need to start setting breakpoints and debugging the PHP script.

Listing 4. Bad habit of not throwing or handling errors

```php
class PersonUtils
{
    public static function parsePersonName($format, $val)
    {
        if (strpos(",", $val) > 0) {
            $person = new Person();
            $parts = split(",", $val); // Assume the value is last, first
            $person->setGivenName($parts[1]);
            $person->setFamilyName($parts[0]);
        }
        return $person;
    }
}
```

The `parsePersonName()` method in Listing 4 could be modified to initialize the `Person` object outside the `if` condition, ensuring that you always get a valid `Person` object. However, you get a `Person` with no set properties, which doesn't leave you in a much better position.

Good habit: Each module handles its own errors

Instead of leaving your callers guessing, be proactive about validating arguments. If an unset variable can't produce a valid result, check for the variable and throw an `InvalidArgumentException`. If the string can't be empty or must be in a specific format, check for the format and throw an exception. Listing 5 demonstrates how to create your own exceptions, as well as some new conditions in the `parsePerson()` method that demonstrate some rudimentary validations.

Listing 5. Good habit of throwing errors

```php
<?php
class InvalidPersonNameFormatException extends LogicException {}


class PersonUtils
{
    public static function parsePersonName($format, $val)
    {
        if (! $format) {
            throw new InvalidPersonNameFormatException("Invalid PersonName format.");
        }

        if ((! isset($val)) || strlen($val) == 0) {
            throw new InvalidArgumentException("Must supply a non-null value to parse.");
        }


    }
}
?>
```

The bottom line is that you want people to be able to use your class without having to know the inner workings of it. If they use it incorrectly or in a way you didn't intend, they shouldn't have to guess why it didn't work. As a good neighbor, you understand that people reusing your class are not psychic, and, therefore, you take the guesswork out.

Avoid looking at Medusa

When I was first learning about OO concepts, I was doubtful that interfaces were really helpful. A colleague of mine drew the analogy that not using interfaces is like looking at the head of Medusa. In Greek mythology, Medusa was a female character with snakes for hair. Any person who looked at her directly turned to stone. Perseus, who killed Medusa, was able to confront her by looking at her reflection in his shield, thus avoiding being turned to stone.

Interfaces are your mirror in dealing with Medusa. When you use a specific, concrete implementation, your code must change if your implementation code changes. Using implementations directly limits many of your options, as you've essentially turned your classes to stone.

Bad habit: Not using interfaces

Listing 6 shows an example that loads the `Person` object from a database. It takes the person's name and returns the `Person` object in the database that matches.

Listing 6. Bad habit of not using interfaces

```php
<?php
class DBPersonProvider
{
    public function getPerson($givenName, $familyName)
    {
        /* go to the database, get the person... */
        $person = new Person();
        $person->setPrefix("Mr.");
```

```
        $person->setGivenName("John");
        return $person;
    }
}

/* I need to get person data... */
$provider = new DBPersonProvider();
$person = $provider->getPerson("John", "Doe");

echo($person->getPrefix());
echo($person->getGivenName());

?>
```

The code for loading `Person` from the database is fine until something changes in the environment. For example, loading `Person` from the database may be fine for the first version of the application, but for the second version, you may need to add the ability to load a person from a Web service. In essence, the class has turned to stone because it is directly using the implementation class and is now brittle to change.

Good habit: Use interfaces

Listing 7 shows an example of code that does not change as new ways of loading users become available and are implemented. The example shows an interface called `PersonProvider`, which declares a single method. If any code uses a `PersonProvider`, the code restrains from using the implementation classes directly. Instead, it uses `PersonProvider` as if it were a real object.

Listing 7. Good habit of using interfaces

```php
<?php
interface PersonProvider
{
    public function getPerson($givenName, $familyName);
}

class DBPersonProvider implements PersonProvider
{
    public function getPerson($givenName, $familyName)
    {
        /* pretend to go to the database, get the person... */
        $person = new Person();
        $person->setPrefix("Mr.");
        $person->setGivenName("John");
        return $person;
    }
}

class PersonProviderFactory
{
    public static function createProvider($type)
    {
        if ($type == 'database')
        {
            return new DBPersonProvider();
        } else {
            return new NullProvider();
        }
    }
}

$config = 'database';
/* I need to get person data... */
$provider = PersonProviderFactory::createProvider($config);
$person = $provider->getPerson("John", "Doe");

echo($person->getPrefix());
echo($person->getGivenName());
?>
```

When you use interfaces, try to avoid ever referring to the implementation classes directly. Instead, use something external to your object to give you the correct implementation. If your class loads the implementation based on some logic, it still needs to require the definitions of all the implementation classes, and that doesn't get you anywhere.

You can use a Factory pattern to create an instance of an implementation class that implements your interface. A `factory` method, by convention, begins with `create` and returns the interface. It can take whatever arguments are necessary for your `factory` to figure out which implementation class is the correct one to return.

In Listing 7, the `createProvider()` method simply takes a `$type`. If the `$type` is set to `database`, the factory returns an instance of `DBPersonProvider`. Any new implementation for loading people from a store does not require any changes in the class that uses the factory and interface. The `DBPersonProvider` implements the `PersonProvider` interface and has the actual implementation of the `getPerson()` method in it.

Embrace the weakest link

Loosely coupling your modules is a good thing; it's one of the properties that allows you to encapsulate change. Two of the other habits — "Be modest" and "Avoid looking at Medusa" — help you work toward building modules that are loosely coupled. To loosely couple your classes, develop the final characteristic by building the habit of lowering the dependencies of your classes.

Bad habit: Tight coupling

In Listing 8, lowering dependencies is not necessarily lowering the dependencies for the client using an object. Rather, the example demonstrates lowering your dependencies on the correct class and minimizing them elsewhere.

Listing 8. Bad habit of tight coupling from Address

```
<?php

require_once "./AddressFormatters.php";

class Address
{
    private $addressLine1;
    private $addressLine2;
    private $city;
    private $state; // or province...
    private $postalCode;
    private $country;

    public function setAddressLine1($line1)
    {
        $this->addressLine1 = $line1;
    }

              /* accessors, etc... */

    public function getCountry()
    {
        return $this->country;
    }

    public function format($type)
    {
        if ($type == "inline") {
            $formatter = new InlineAddressFormatter();
        } else if ($type == "multiline") {
            $formatter = new MultilineAddressFormatter();
        } else {
            $formatter = new NullAddressFormatter();
        }
        return $formatter->format($this->getAddressLine1(),
            $this->getAddressLine2(),
            $this->getCity(), $this->getState(), $this->getPostalCode(),
            $this->getCountry());
    }
}

$addr = new Address();
$addr->setAddressLine1("123 Any St.");
$addr->setAddressLine2("Ste 200");
$addr->setCity("Anytown");
$addr->setState("AY");
$addr->setPostalCode("55555-0000");
$addr->setCountry("US");

echo($addr->format("multiline"));
echo("\n");

echo($addr->format("inline"));
echo("\n");

?>
```

The code that calls the `format()` method on the `Address` object might look great — all it does is use the `Address` class, call `format()`, and it's done. In contrast, the `Address` class is not so lucky. It needs to know about the various formatters used to properly format it, which might not make the `Address` object very reusable for someone else, particularly if that someone else isn't interested in using the formatter classes in the `format()` method. Although the code using `Address` doesn't have many dependencies, the `Address` class does have quite a few when it probably should just be a simple data object.

The `Address` class is tightly coupled with the implementation classes that know how to format the `Address` object.

Good habit: Loose coupling between objects

When building good OO designs, it's necessary to think about a concept called *Separation of Concerns* (SoC). SoC means that you try to separate objects by what they should be really concerned about, thus, lowering the coupling. In the original `Address` class, it had to be concerned about how to format itself. That's probably not a good design. Rather, an `Address` class should think about the parts of the `Address`, while some type of formatter should worry about how to properly format the address.

In Listing 9, the code that formatted the address is moved to interfaces, implementation classes, and a factory — building the "use interfaces" habit. Now, the `AddressFormatUtils` class is responsible for creating a formatter and formatting an `Address`. Any other object now can use an `Address` without having to worry about also requiring the definitions of the formatters.

Listing 9. Good habit of loose coupling between objects

```php
<?php

interface AddressFormatter
{
    public function format($addressLine1, $addressLine2, $city, $state,
        $postalCode, $country);
}

class MultiLineAddressFormatter implements AddressFormatter
{
    public function format($addressLine1, $addressLine2, $city, $state,
        $postalCode, $country)
    {
        return sprintf("%s\n%s\n%s, %s %s\n%s",
            $addressLine1, $addressLine2, $city, $state, $postalCode, $country);
    }
}

class InlineAddressFormatter implements AddressFormatter
{
    public function format($addressLine1, $addressLine2, $city, $state,
        $postalCode, $country)
    {
        return sprintf("%s %s, %s, %s %s %s",
            $addressLine1, $addressLine2, $city, $state, $postalCode, $country);
    }
}

class AddressFormatUtils
{
    public static function formatAddress($type, $address)
    {
        $formatter = AddressFormatUtils::createAddressFormatter($type);

        return $formatter->format($address->getAddressLine1(),
            $address->getAddressLine2(),
            $address->getCity(), $address->getState(),
            $address->getPostalCode(),
            $address->getCountry());
    }

    private static function createAddressFormatter($type)
    {
        if ($type == "inline") {
            $formatter = new InlineAddressFormatter();
        } else if ($type == "multiline") {
            $formatter = new MultilineAddressFormatter();
        } else {
            $formatter = new NullAddressFormatter();
        }
        return $formatter;
```

```
        }
}

$addr = new Address();
$addr->setAddressLine1("123 Any St.");
$addr->setAddressLine2("Ste 200");
$addr->setCity("Anytown");
$addr->setState("AY");
$addr->setPostalCode("55555-0000");
$addr->setCountry("US");

echo(AddressFormatUtils::formatAddress("multiline", $addr));
echo("\n");

echo(AddressFormatUtils::formatAddress("inline", $addr));
echo("\n");
?>
```

The drawback, of course, is that whenever patterns are used, it often means the amount of artifacts (classes, files) goes up. However, this increase is offset by the decreased maintenance in each class and can be decreased even more when proper reusability is gained.

You're rubber; I'm glue

Highly cohesive OO designs are focused and organized in related modules. Learning about "concerns" is important in determining how to organize functions and classes to be tightly cohesive.

Bad habit: Low cohesion

When a design has *low cohesion,* it has classes and methods that are not grouped well. The term *spaghetti code* is often used to describe classes and methods that are bunched together and have low cohesion. Listing 10 provides an example of spaghetti code. The relatively generic `Utils` class uses many different objects and has many dependencies. It does a bit of everything, making it difficult to reuse.

Listing 10. Bad habit of low cohesion

```
<?php

class Utils
{
    public static function formatAddress($formatType, $address1,
        $address2, $city, $state)
    {
        return "some address string";
    }

    public static function formatPersonName($formatType, $givenName,
        $familyName)
    {
        return "some person name";
    }

    public static function parseAddress($formatType, $val)
    {
        // real implementation would set values, etc...
        return new Address();
    }

    public static function parseTelephoneNumber($formatType, $val)
    {
        // real implementation would set values, etc...
        return new TelephoneNumber();
    }
}

?>
```

Good habit: Embrace high cohesion

High cohesion means that classes and methods that are related to each other are grouped. If methods and classes are highly cohesive, you are able to easily split off entire groups without affecting the design. Designs with high cohesion offer the opportunity for lower coupling. Listing 11 shows two of the methods that are better organized into classes. The `AddressUtils` class contains methods for dealing with `Address` classes and shows high cohesion among the address-related methods. Likewise, `PersonUtils`

contains methods that deal specifically with `Person` objects. These two new classes with their highly cohesive methods have low coupling because they can be used completely independently of one another.

Listing 11. Good habit of high cohesion

```php
<?php

class AddressUtils
{
    public static function formatAddress($formatType, $address1,
        $address2, $city, $state)
    {
        return "some address string";
    }

    public static function parseAddress($formatType, $val)
    {
        // real implementation would set values, etc...
        return new Address();
    }

}

class PersonUtils
{
    public static function formatPersonName($formatType, $givenName,
        $familyName)
    {
        return "some person name";
    }

    public static function parsePersonName($formatType, $val)
    {
        // real implementation would set values, etc...
        return new PersonName();
    }
}

?>
```

Keep it in the family

I often tell people on the software teams on which I've been a technical lead or architect that the greatest enemy of OO languages is a copy-and-paste operation. When used in the absence of an up-front OO design, nothing creates more havoc than copying code from one file to the next. Wherever you're tempted to copy code from one class to the next, stop and consider how you can use class hierarchies to leverage similar or identical functionality. You will find that in most cases, with good design, copying code is completely unnecessary.

Bad habit: Not using class hierarchies

Listing 12 shows a simple example of partial classes. They start with duplicate fields and methods — not good in the long term when the application might need to change. If there was a defect in the `Person` class, there would most likely be a defect in the `Employee` class as well because it appears as though the implementation was copied between the two.

Listing 12. Bad habit of not using hierarchies

```php
<?php
class Person
{
    private $givenName;
    private $familyName;
}

class Employee
{
    private $givenName;
    private $familyName;
}

?>
```

How likely are you to recommend developerWorks to a friend/colleague?

0　1　2　3　4　5　6　7　8　9　10

Not likely at all　　　　　　　　　　　Extremely likely

Inheritance is a difficult habit to start using because often, the ar[...]models can take a lot of time. Conversely, using **Ctrl+C** and **Ctrl+V** to b[...]y

seconds. But the time is usually offset rather quickly in maintena [Send] ually
spend most of its time.

Good habit: Leverage inheritance

In Listing 13, the new `Employee` class extends the `Person` class. It now inherits all the common methods
and doesn't reimplement them. Additionally, Listing 13 shows the use of an abstract method to demonstrate
how basic functionality can be put into a base class and specific functionality can be deterred to an
implementation class.

Listing 13. Good habit of leveraging inheritance

```php
<?php
abstract class Person
{
    private $givenName;
    private $familyName;

    public function setGivenName($gn)
    {
        $this->givenName = $gn;
    }

    public function getGivenName()
    {
        return $this->givenName;
    }

    public function setFamilyName($fn)
    {
        $this->familyName = $fn;
    }

    public function getFamilyName()
    {
        return $this->familyName;
    }

    public function sayHello()
    {
        echo("Hello, I am ");
        $this->introduceSelf();
    }

    abstract public function introduceSelf();

}

class Employee extends Person
{
    private $role;

    public function setRole($r)
    {
        $this->role = $r;
    }

    public function getRole()
    {
        return $this->role;
    }

    public function introduceSelf()
    {
        echo($this->getRole() . " " . $this->getGivenName() . " " .
            $this->getFamilyName());
    }
}
?>
```

Think in patterns

Design patterns are common interactions of objects and methods that have been proven over time to
resolve certain problems. When you think in design patterns, you're forcing yourself to be aware of how
classes interact with each other. It's an easy way to build classes and their interactions without having to
make the same mistakes other people have made in the past an

**How likely are you to recommend
developerWorks to a friend/colleague?**

0   1   2   3   4   5   6   7   8   9   10

Not likely at all   Extremely likely

Bad habit: Consider one object at a time

There is really no adequate code example that demonstrates thi   plenty
of good examples showing pattern implementations). However, g   e

considering only one object at a time when the following criteria

Send

You don't diagram an object model ahead of time.

You start coding the implementation of single methods without much of the model stubbed out.

You don't use design pattern names when talking and would rather talk about implementation.

Good habit: Adding objects, in concert, composed in patterns

Generally speaking, you are thinking in patterns when you:

Model classes and their interactions ahead of time.

Stereotype classes according to their patterns.

Use pattern names, like *Factory, Singleton*, and *Facade*.

Stub out large portions of the model, then start adding implementation.

Conclusion

Building good OO habits in PHP helps you build more stable, easily maintainable, and more easily extensible applications. Remember:

Be modest.

Be a good neighbor.

Avoid looking at Medusa.

Embrace the weakest link.

You're rubber, I'm glue.

Keep it in the family.

Think in patterns.

As you build and get into these habits, you're likely to be surprised at the changes in the quality of your applications.

Resources

Learn

Read the Wikipedia entry on Module programming.

Read more about object-oriented design in Bertrand Meyer's book titled *Object-Oriented Software Construction*.

Visit the official PHP Web site's PHP manual section on Classes and Objects.

Read about Design Patterns in PHP in "Five common PHP design patterns" and "Five more PHP design patterns."

PHP.net is the central resource for PHP developers.

Check out the "Recommended PHP reading list."

Dig deeper into Open source on developerWorks

Overview

New to Open source

Projects

Technical library (tutorials and more)

Forums

Events

How likely are you to recommend developerWorks to a friend/colleague?

0 1 2 3 4 5 6 7 8 9 10

Not likely at all   Extremely likely

emium

your next

bs

Browse all the PHP content on developerWorks.

Expand your PHP skills by checking out IBM developerWorks' PHP project resources.

To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.

Using a database with PHP? Check out the Zend Core for IBM, a seamless, out-of-the-box, easy-to-install PHP development and production environment that supports IBM DB2 V9.

Stay current with developerWorks' Technical events and webcasts.

Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.

Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.

Get products and technologies

Innovate your next open source development project with IBM trial software, available for download or on DVD.

Download IBM product evaluation versions, and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

Participate in developerWorks blogs and get involved in the developerWorks community.

Participate in the developerWorks PHP Forum: Developing PHP applications with IBM Information Management products (DB2, IDS).

innovators
xperiment
with.

IBM evaluation software
Evaluate IBM software and solutions, and transform challenges into opportunities.

Send

How likely are you to recommend developerWorks to a friend/colleague?

0   1   2   3   4   5   6   7   8   9   10

Not likely at all   Extremely likely