

Website Architecture

Lezione 9

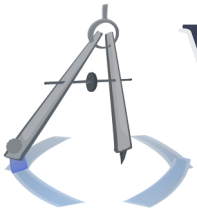
PHP

A walk through the language features of PHP, especially as they differ from those of C++ and typical compiled languages. Includes a survey of common problems and solutions appropriate for PHP, as well as the most common ways in which PHP can interact with the Web server.

Michael Serritella

Summer 2010





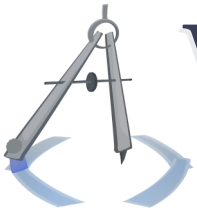
Website Architecture

Lezione 1 | PHP

Intro to PHP

PHP is a scripting language which is well positioned to serve Web pages. It has a few built-in features that help you to interact with the server and the HTTP headers, and it has loads of easy string-processing functions. Thus, PHP is one of the most popular programming languages in the world and is the *de facto* standard Web programming language. PHP has an OOP model that is taken from Java, and its syntax much like any C++-based language. It can be characterized by the ability to write quick, hack programs with fast and loose disregard of semantic and logic errors, though it may be used to write elegant programs.





Website Architecture

Lezione 9 | PHP

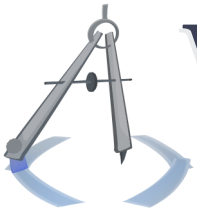
Getting started

Running PHP scripts requires a PHP interpreter; most Web servers have one, and there is also a version for the command line, so that they may be run like any shell script. The latter is called PHP CLI, which stands for Command-Line Interface. On Unix, you may have to get a package called **php-cli** or something similar. After you have a working environment, there is some mandatory syntax for the PHP document, but it is pretty slim.

Script execution

If on a Web server, simply drop the .php file into a directory that is served to the outside world and then go to that URI (e.g. "http://localhost/test.php"). If on the Unix command line, type "**php <scriptPath>**" to execute the script. You may also execute the script just like a shell script, wherein you set the execute permissions and refer to the PHP binary via the first line of the script (e.g. "#! /bin/php").





Website Architecture

Lezione 9 | PHP

PHP parser tags

Within your PHP script, you actually have to encompass your code in special tags, or else it won't work. Enclose them in "<?php" and "?>". You could use "<?" for the opening tag, but this is less responsible nowadays, since XML has processing instructions with the same syntax.

Here is an example:

PHP

```
<?php  
print("Hello, world!");  
?>
```

All of that text may occur on one line. Interestingly enough.. The parser tags may occur many times within the document, like this:

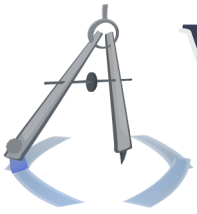
PHP

```
<?php print("Hello, world!"); ?>
```

Some text that will be printed (untouched)

```
<?php  
// Some more PHP  
?>
```





Website Architecture

Lezione 9 | PHP

Slipping in and out of the parser like this is not really a good idea; it incurs a CPU overhead and may confuse the reader. What's even worse is this:

PHP

```
<?php
for ($i = 0; $i < 10; $i++)
{
?>
```

This will be printed 10 times!

```
<?php
}
?>
```

Don't do things like this.

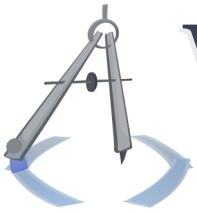
Inclusion of other scripts

You may include other scripts, of course. Do it like so:

PHP

```
<?php
include 'anotherFile.php';
?>
```





Website Architecture

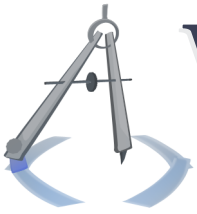
Lezione 9 | PHP

When you include the other file, *it must also have parser tags*, just as usual. And - this is important - in most PHP files and especially includes, you want to be sure that *no whitespace or text exists outside the parser tags*. This means that you can't do the slipping-in-and-out-of-the-parser trick. We'll see the reason for this when we talk about Web servers and HTTP headers.

You can include a file idempotently with `include_once (docs)`, and, using `require (docs)` instead of `include`, you can force an error if the include fails.

You can do all kinds of stupid things with the include directive; you can include a plaintext file (which isn't too terribly stupid, but you better be sure that no one injected any code), include a file from some elsewhere on the Internet, "return a value" at the end of an included file, and more. See the **documentation** for more, but be wary of any of these exotic uses. PHP wasn't written by grandmaster computer scientists or security experts, so not every feature is a good idea, especially these core/aged features.





Website Architecture

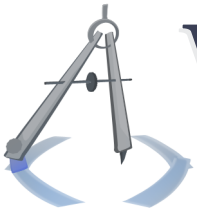
Lezione 9 | PHP

Documentation

When getting started with PHP, you should check out its documentation. It is well organized and thorough, with a fairly active community comments section. If you suspect any feature of being flawed or insecure, or if you just want help getting started, check its comments.

See the **PHP Manual** for the top of the tree. You can read the main branches and selectively read (or skip) the Security and Function Reference sections. The Security section is a decent introduction, but don't fall under the impression that it is a comprehensive treatment of computer security. Computer security is hardly any good unless it is comprehensive, so you may want to supplement or replace this section with a well-reviewed security book. The Function Reference section contains an overload of information for functions which you probably don't care about. Most may not even be included in your version of PHP. Peruse the list and look for any titles that look relevant.





Basic language features

A look at the basics - variables and functions.

Variables

PHP is a loosely-typed language characterized by lazy evaluation and reference semantics.

Declaration (not)

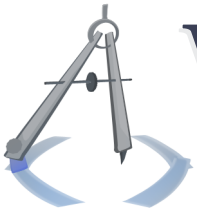
In PHP, if you use a variable name, it exists. You don't have to declare variables. This is a small strength and a moderate-sized weakness, as it allows for subtle errors. For instance:

PHP

```
<?php
$aVariable = 5;
print($avariabLe); // Prints the empty string.
?>
```

You can configure PHP to alert you to errors like this or to ignore them; in the most typical configuration, they are ignored.





Website Architecture

Lezione 9 | PHP

Indirect variable access

You can refer to variables by a sort of level of indirection; you can store a variable name as a string, and then you can work with the variable whose name is in the string. To do this, you use double-dollar syntax:

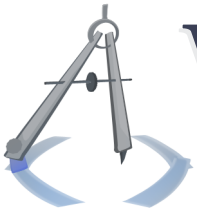
```
PHP
<?php
$aVariable = 'hey';
$aVariableName = 'aVariable';
print($$aVariableName);
?>
```

Type system

Since PHP is loosely typed, any variable can be any type; this includes boolean, integer, float, string, array, object, resource, and the special values **null** and **undefined**. You can do something like this:

```
PHP
<?php
$something = null;
$something = 5;
$something = 'Hello';
?>
```





Website Architecture

Lezione 9 | PHP

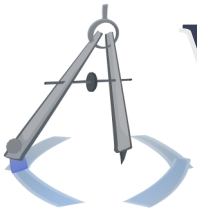
And since you can do something like that, it would help to be able to know a variable's type. You can do this in two ways. You can use the **variable-handling functions**, which also help in type conversion. Or you can check for value and equality together. The former is a general solution and the latter is more directly useful in some instances.

PHP

```
<?php
$something = 0;
if ($something == false)
{
    // This would execute
}
if ($something === false)
{
    // This would NOT execute;
    // could also use !==
}
?>
```

This is called a "strict equality" check. You may also refer to an article in the PHP documentation called "**Type Juggling**".





Website Architecture

Lezione 9 | PHP

Reference semantics

In the same way as in JavaScript and other scripting languages, PHP treats object variables as references. For a quick run-down, look at this example:

PHP

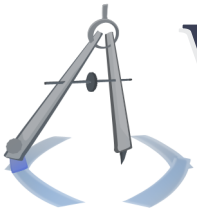
```
<?php
$obj = new SomeClass();
$sameObj = $obj;
$sameObj->DoSomething(); // Affects original object.
$obj = null; // One reference gone; one remains.
$sameObj = null; // No references; object destroyed.
?>
```

Now - more importantly than this - PHP supports references similar to those in C++ when you pass a value by reference. You can pass values by reference to PHP functions, and you can store references to other variables. For example:

PHP

```
<?php
$num = 7;
$sameNum = &$num;
$sameNum++;
print($num); // Prints 8.
?>
```





Website Architecture

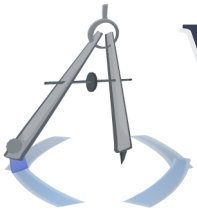
Lezione 9 | PHP

You can read a sizeable explanation of references in the **documentation**. They're not as scary as they sound, mainly because you're a computer scientist and no one else is.

Resources

The PHP compiler and interpreter are written in C, and most of PHP's built-in runtime functionality is written in C. This includes the implementations of most functions which come pre-installed. Furthermore, some PHP extensions are written in C (see **PECL**; say "pickle"). For example, the regular-expression engine is written in C. And so are things like database libraries. Some libraries give the PHP user an object or handle that they can keep in order to maintain some kind of state or continuity between calls to library functions (e.g. a database connection handle). In PHP parlance, that handle is a resource. Resources are simple tokens. You pass them to functions whose implementations are written in C libraries. They are inert, and you cannot do anything with those variables directly from within PHP ("userspace").





Website Architecture

Lezione 9 | PHP

Strings

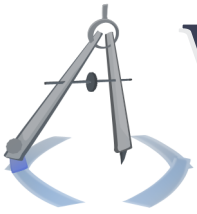
We will see string functions later, of which there are dozens, but now we concern ourselves with the string type. The type is simple, really, but it does have some exceptional behavior.

Quotes and operators

A string can be delimited by either single quotes or double quotes. If you use single quotes, the string does not have much special behavior, and there is only one escape sequence: the escape for a single quote. If you use double quotes, however, you can use many familiar escape sequences from C++, like `"\n"` and `"\t"`, enabling you to write special characters into strings. If you incorporate the `$`, unescaped, you can refer to variable names and their values will be expanded and written into the string.

The string concatenation operator is the period (dot). There is also the `".="` syntax for appending to an existing string variable.





Website Architecture

Lezione 9 | PHP

For a cumulative example:

PHP

```
<?php
$username = 'Bob Smith';
$greeting = "Hello, $username.";
// Can re-combine with a single-quoted string; no big deal.
$fullMessage = $greeting.' How are you?';

// Message string already built; this does not affect it.
$username = '(something else)';

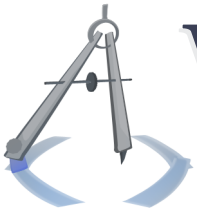
// Prints "Hello, Bob Smith. How are you?"
print($fullMessage);
?>
```

This describes the most common usage of strings, but there are more corner cases. See the **documentation**, and note that **print()** and **echo** are synonymous. To conclude: If you don't need special characters or parsing, you should probably use single quotes.

Type coercion

PHP allows for some cheeky tricks. You can do this:





Website Architecture

Lezione 9 | PHP

PHP

```
<?php
$aNumber = 10;
$aString = $aNumber.' pounds';

$anotherVariable = 5;
$anotherVariable .= ' feet';
?>
```

This coerces the `$aNumber` expression and `$anotherVariable` variable to strings.

Arrays

Arrays in PHP are loose and powerful; they are a strong feature of PHP.

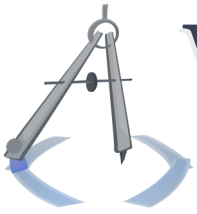
Declaration and element insertion

Array declaration syntax is unfortunately an anachronism in modern PHP. It looks like this:

PHP

```
<?php
$array = array();
?>
```





Website Architecture

Lezione 9 | PHP

There is no "new"; just `array()`;

To declare an array and initialize it with some starting elements, you can do this:

```
PHP
<?php
$array = array("hello", 88, true);
?>
```

To insert into an array, you can just specify the index like this:

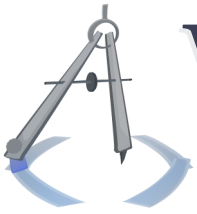
```
PHP
<?php
$array[3] = "something";
?>
```

Or use a special syntax to insert at the end of the array:

```
PHP
<?php
$array[] = "something else";
?>
```

But hey - wait! Arrays in PHP are also associative, which





Website Architecture

Lezione 9 | PHP

means you can do this:

PHP

```
<?php
$array["Hello"] = "Buongiorno";
$array["Goodbye"] = "Arrivederci";
?>
```

And this:

PHP

```
<?php
$array = array("Hello"=>"Hola", "Goodbye"=>"Adios");
?>
```

And this, even - a heterogeneously-keyed array:

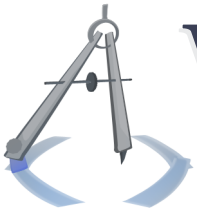
PHP

```
<?php
$array = array("Hello"=>"Hola", "Goodbye"=>"Adios", 1=>"Asdf");
?>
```

In an associative array like these most recent ones, the order of insertion is preserved, so you may iterate through the array in that same order.

For more complicated insertion and extraction patterns, see





Website Architecture

Lezione 9 | PHP

the extensive library of **array functions**.

Iterating over elements: foreach

As in most modern languages, PHP has a foreach loop, which lets you iterate over the elements in the array. However, its syntax is a little unique. You will just have to memorize it.

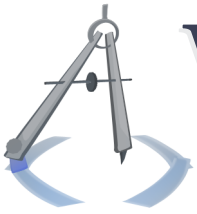
PHP

```
<?php
$array = array("Hello"=>"Hola", "Goodbye"=>"Adios", 1=>"Asdf");

foreach ($array as $element)
{
    print("Element: $element\n");
}
?>
```

Within the loop, **\$element** takes the value of each element. However.. it is only a copy of those elements. If you try to modify **\$element** directly, you will only be modifying a copy, and you will have no effect on the array. In order to do that, you will have to invoke a more complicated spell:





Website Architecture

Lezione 9 | PHP

PHP

```
<?php
foreach (array_keys($anArray) as $aKey)
{
    // Convert all to uppercase.
    $anArray[$aKey] = strtoupper($anArray[$aKey]);
}
?>
```

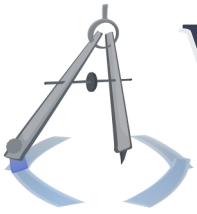
The function `array_keys()` takes an array and returns a numerically-indexed (regular/plain) array of its keys. Then, in this loop, you use the key to index back into the array and change its elements.

In a similar vein, you can loop through the keys and values of an array like this:

PHP

```
<?php
foreach ($anArray as $aKey => $aValue)
{
    print("Key $aKey; Value $aValue\n");
}
?>
```





Website Architecture

Lezione 9 | PHP

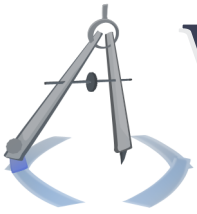
Internal pointers

Using a certain set of functions, you can exploit this peculiar feature; arrays can keep track of a cursor, and you can manipulate it and retrieve the key & element to which it points. See the functions `reset()`, `current()`, and `next()` to get started.

As a mathematical set (element existence checker)

You can hijack arrays to behave somewhat like mathematical sets, in that you can use them to quickly check if an element already exists in a set, quickly insert into a set, or quickly delete from a set. You actually use the array's keys as the set. PHP uses a hash table to keep the array keys and to look them up when they are accessed, and a hash table is basically the fastest possible data structure for this purpose. So, in the course of its normal operations, it needs to do all of these existence checks. How would you exploit them to emulate a mathematical set? Here's an example:





Website Architecture

Lezione 9 | PHP

PHP

```
<?php
$arrayAsSet = array("Bob"=>1, "Joe"=>1, "Adam"=>1);

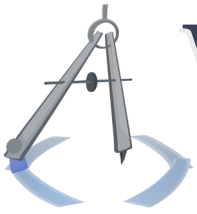
// Try to add 'Mark'; check if it's already there.
if (!isset($arrayAsSet["Mark"]))
{
    $arrayAsSet["Mark"] = 1;
}

// A way to simply get the elements in the set.
$listOfNames = array_keys($arrayAsSet);
?>
```

Functions

Functions in PHP look a lot like those in JavaScript. They are more or less variadic, and they don't have return type designations or parameter types (mostly).





Website Architecture

Lezione 9 | PHP

Syntax example

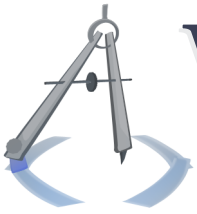
PHP

```
<?php
function DoIt($ParamOne,
              array $ParamTwoMustBeArray,
              AClassName $ParamThreeMustBeAParticularClass,
              AnotherClassName $SortOfStrange=null)
{
    return new Blah();
}
?>
```

Return types and parameter types

No return type is required, and functions may or may not return a value. There is no way to constrain the return type of a function, including from the caller's side. Simple enough. Parameters are untyped, with a few exceptions. Sometimes, you can use type hinting (**docs**) to force a parameter to be of a certain type. This only works for arrays and objects, in which case you can specify that the parameter must be of a certain class or its descendants (or, in the case of an interface, the parameter must implement the interface or one of its descendants).





Website Architecture

Lezione 9 | PHP

Variadic calling and default parameters

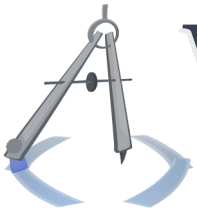
You can give default values to parameters, much like you can in C++ and many other languages. However, there is something strange afoot with the parameter in the prior example called `$SortOfStrange`. Normally, if you use type hinting to enforce that a variable is an object, the parameter cannot be null. But if you combine type hinting with a default value, the default value can be null. Hmm.

In addition to giving too few arguments (i.e. using default parameters), you can also give too many arguments. PHP provides functions which let you retrieve the arguments as an array. See `func_get_args()` ([docs](#)).

Function-handling functions

As we're starting to see, there are functions which give support for advanced function behaviors. See the ([documentation](#)) for the function-handling functions.





Website Architecture

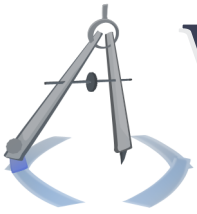
Lezione 9 | PHP

Classes and OOP

PHP offers an OOP model much like Java, which is a sort of simplified version of the model in C++. However, in recent years, PHP has been expanding its OOP model to include features that are new to C++ programmers and perhaps even to Java programmers (who knows).

We will go over the core parts of the OOP model, but for a full explanation, you can read the **documentation**.





Website Architecture

Lezione 9 | PHP

Syntax example

PHP

```
<?php
class AClass
    extends PerhapsAnotherClass
    implements ThisInterface,
               ThatInterface
{
    protected static $AProtectedAndStaticVariable;
    protected static $APublicAndStaticVariable;
    private $APrivateVariable;
    public $APublicVariable;

    public function __construct($SomeParam)
    {
        // constructor..
    }

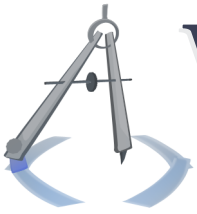
    public static function DoStat() { }

    public function DoThat() { }

    public function DoIt()
    {
        print($this->APrivateVariable);
        $this->DoThat();
        // ..
    }
} // End class AClass

$anObject = new AClass(5);
?>
```





Website Architecture

Lezione 9 | PHP

Member access

To access a member of a class, from inside the class or outside the class, you use the following syntax:

PHP

```
<?php
$obj->DoIt();
$obj->APublicVariable = 99;
AClass::DoStat(); // Access a static function.
AClass::$APublicAndStaticVariable = 5; // Access a static variable.
?>
```

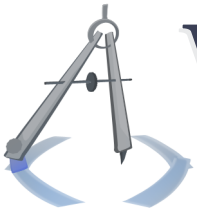
Access restrictions

Members of a class may be public, private, or protected, like in C++. There is no class friendship like there is in C++. There is no grouping members with a **public:** grouping or anything like that; access modifiers have to come before each member.

OOP model and inheritance

The OOP model is characterized by single inheritance (no multiple inheritance) and use of interfaces.





Website Architecture

Lezione 9 | PHP

Interfaces

Interfaces are sort of like "templates" (not like in C++) or "contracts"; essentially, they are class definitions that only have declarations for public methods. The methods are not defined (i.e. there is no method body). A class can "implement an interface", which is like deriving from it, as seen in the above example. If it implements the interface, then it must define all the functions that are outlined in the interface.

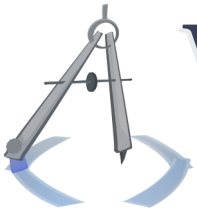
Outside the class (or not), a function can use type hinting to say that it accepts a certain interface, like this:

```
PHP
<?php
function AFunc(iSomeInterface $SomeObject)
{
}
?>
```

(A typical naming convention is to start them with "i" or "I").

One distinguishing feature of interfaces is that a class can implement any number of interfaces; we saw in the initial example how a class may implement two interfaces.





Website Architecture

Lezione 9 | PHP

Abstract classes

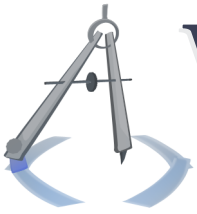
Abstract classes are very much like abstract classes in C++. They are just like regular classes except that at least some of its methods are unimplemented. Thus, they cannot be instantiated. You can simply label an abstract class like so:

```
PHP
<?php
abstract class AnAbstractClass
{
    // An implemented method.
    public function DoSomething() { return 7; }

    abstract public function DoSomethingElse();
}
?>
```

They have a few interesting behaviors. Abstract classes can implement interfaces and derive from other abstract classes. Only the final link in the chain - the instantiable, regular class - has to worry about implementing everything from its cumulative ancestry. The abstract classes don't have to worry about mentioning methods from their interfaces if they want to defer them to normal classes. In fact, if you mention it, you get some kind of mysterious error, at the time of this writing.





Website Architecture

Lezione 9 | PHP

Example for clarity:

PHP

```
<?php
interface iAnInterface
{
    public function OneMethod();
    public function TwoMethod();
}

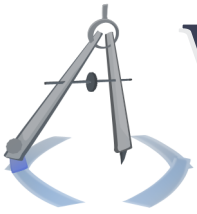
abstract class AbstractClassA
    implements iAnInterface
{
    public function OneMethod() { return 9; }
    // Do not even mention TwoMethod(); don't list it here with
    // an 'abstract' qualifier in front of it, for instance.

    abstract protected function MethodABC();
}

abstract class AbstractClassB
    extends AbstractClassA
{
    abstract private function MethodXYZ();
}

class AConcreteClass
    extends AbstractClassB
{
    public function TwoMethod() { return 11; }
    protected function MethodABC() { return 13; }
    private function MethodXYZ() { return 17; }
}
?>
```





Website Architecture

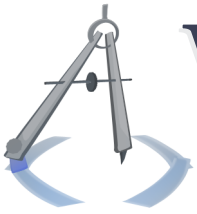
Lezione 9 | PHP

Abstract classes have some powers that interfaces don't. You can define variables in an abstract class, and you can define method bodies. The abstract "contract" can include private and protected methods. The downside, of course, is that a class can only derive from one abstract class at a time; they are no exception to that rule.

Overriding and parent-member access

By default, all methods in PHP are virtual. What may be surprising is that even intra-class method calls are virtual (though this is like C++). For example:





Website Architecture

Lezione 9 | PHP

PHP

```
<?php
class BaseClass
{
    protected function PrintIt()
    {
        print("The base class says hey!\n");
    }

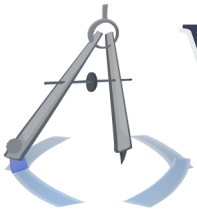
    public function DoIt($x)
    {
        $y = $x + 4;
        $this->PrintIt();
    }
}

class DerivedClass
{
    protected function PrintIt()
    {
        print("The derived class says HEY!\n");
    }
}

$anObject = new DerivedClass();
$anObject->DoIt();           // Eventually calls the "HEY" one.
$anObject->PrintIt();       // Calls the "HEY" one, naturally.
?>
```

To access the parent methods, you have to use a special syntax from within the class:





Website Architecture

Lezione 9 | PHP

PHP

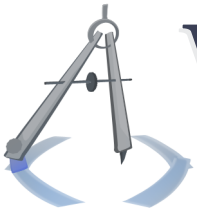
```
<?php
class DerivedClass
{
    protected function PrintIt()
    {
        print("First, the base class says:\n ");
        parent::PrintIt();
        print("The derived class says HEY!\n");
    }
}
?>
```

Unfortunately, you cannot do `parent::parent::`. There would seem no way to directly access a grandparent's method if the parent has overridden it.

Constructor chaining (not)

Now comes a low point in PHP. There is no implicit constructor chaining like there is in any good language. And no destructor chaining! Barbaric. You have to string this together explicitly, using `parent::__construct()`. There would seem no great way to circumvent this deficiency; you (and those who extend your classes!) just have to remember to do it right.





Website Architecture

Lezione 9 | PHP

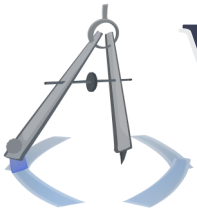
New-fangled frivolity

PHP's OOP model is probably its fastest-growing language component, and many of the new features are helpful, but some of them are too inefficient for performance-critical applications. For instance, some new "magic methods" (**docs**) allow you to catch any attempted method calls for methods which don't exist, and others let you define getter/setter behavior for properties (class variables) which don't exist.

That is not bad as a means to ensure that your program doesn't crash, but unless you need that kind of infinite variability for your class semantics, you should probably avoid it in your production code, especially if you may use it an infinite number of times per each program execution. If you read its documentation, you will see how it is inefficient.

What's perhaps worse is **object iteration**. This is nice syntax, but unless you really need this kind of behavior to accommodate *users of your class*, you may want to design around it. The nature of these iterators implies that they may be called an infinite number of times, and so you will probably feel a decent



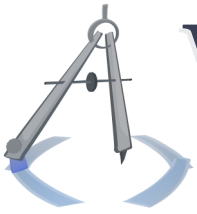


Website Architecture

Lezione 9 | PHP

performance penalty if you use these enough (i.e. if you use them for their intended purpose). Consider internalizing the algorithms which would make use of this feature, so you just provide a function which walks through your object's members in a hard-coded fashion and does all the work.





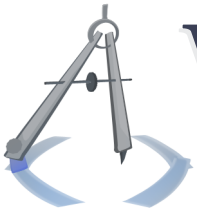
PHP is bad at finding things

When you're learning PHP, it may help to keep this principle in mind so that the language's design choices and limitations make more sense: PHP is bad at finding things. Perhaps it's because PHP is an interpreted language; compiled languages have luxurious amounts of time to analyze the program before running it, and PHP can't afford to spend that time. In any case, here are some features of PHP which exemplify its lack of skill in finding things.

Variable scope resolution

How do you determine whether a variable is local to a function, local to a class, or global? Well, you have to tell PHP. Local variables use the default syntax. For global variables, you have to use a special syntax (**docs**) before using it within a function:





Website Architecture

Lezione 9 | PHP

PHP

```
<?php
$aGlobalVariable;

function Func()
{
    global $aGlobalVariable;

    print($aGlobalVariable);
}
?>
```

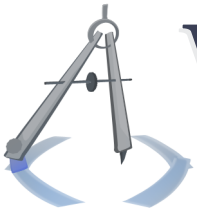
And you recall the **\$this** syntax from before. See, you have to help PHP from time to time.

Finally, there are "superglobals". These are variables which are in scope everywhere, without need for qualification. We will see examples later when we talk about integration with the Web server.

Function scope resolution

Functions are similar. A function is expected to be global unless otherwise stated. You have to either use **\$this** or the **::** operator for static functions, as shown in above examples. Later in





Website Architecture

Lezione 9 | PHP

this section, we will actually see some of PHP's strengths with static members.

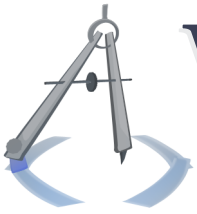
No function overloading

C++ actually uses a highly complex matching algorithm in order to find out which function you are talking about when making an overloaded function call. PHP doesn't have nearly that much game. In PHP, there is no function overloading (unless by magic methods, which are inefficient and whose "magic lie within you" (i.e. you do it yourself)). This is ostensibly because PHP couldn't resolve the function call in a reasonable time frame.

No nested classes or functions

Note that you have to tell PHP how to find some things. If you just make a function or method call, it will look in the global context or the current class (and its parents), and that's it. If you wanted nested classes or functions, PHP would probably have to invent a new operator so you can tell it how to do its job. So, no dice.





Website Architecture

Lezione 9 | PHP

You *can* try, though, in which case you'll get unbelievably quirky results:

PHP

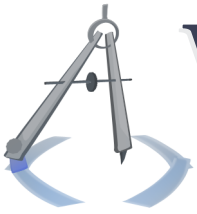
```
<?php
function Func()
{
    // This actually is a simple, global function, and it
    // *isn't defined until Func() is called*, after which
    // point it is available in the global context.
    // Do not do this.
    function NestedFuncOrNotReally()
    {
        return 5;
    }

    return 7;
}
?>
```

No block-scope declarations

You may expect to be able to do something like this:





Website Architecture

Lezione 9 | PHP

PHP

```
<?php
$x = 5;

{
    // Hey - a new copy of $x, right?
    $x = 10;
}

// Man, I hope $x is still 5.

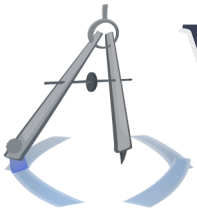
?>
```

You can't. There are no such variable-resolving rules that are aware of blocks, and blocks have no such semantic.

PHP is pretty OK at finding static things

Now for a sort of high point. PHP has some flexible rules for finding static members, which actually improves the expressibility of the language. Static members are inherited by derived classes in a way that integrates them more closely with the derived class than you would see in most other languages.





Website Architecture

Lezione 9 | PHP

For example:

PHP

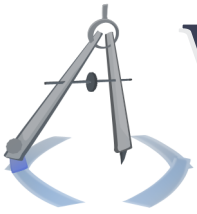
```
<?php
class A
{
    public static function Hello() { print("Hello\n"); }
}

class B
    extends A
{
    public static function Goodbye() { print("Goodbye\n"); }
}
?>

B::Hello();                // Use as if it is a member of B.
B::Goodbye();              // Typical use.
```

This is a nice interface for users outside of the class. You can make static behavior slightly more extensible by incorporating static methods from many classes in your inheritance tree into one cohesive interface; the caller does not need to know where the static methods reside. From inside the class, you can do the same thing, but you also have some new keywords available to you, similar to `$this`.





Website Architecture

Lezione 9 | PHP

There is a **self** keyword, which can be used to refer to static members in your own class or any of your ancestor classes, using the same inheritance semantics as demonstrated above. Example:

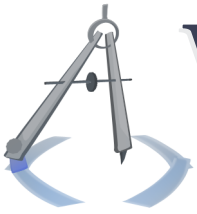
```
PHP
<?php
class A
{
    protected static $AStaticVariable;

    protected static function ParentStaticMethod() { return 2; }
}

class B
    extends A
{
    public function DoIt()
    {
        self::$AStaticVariable = 3;
        self::ParentStaticMethod();
    }
}
?>
```

As of the brand-new PHP 5.3.0, there is also a **static::** usage which can be used to resolve some of these potentially ambiguous or strange behaviors; see the **documentation**.





Website Architecture

Lezione 9 | PHP

Finally, you can access a static method like this:

PHP

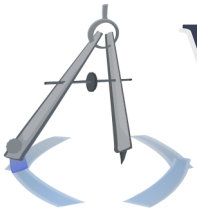
```
<?php
class AClass
{
    public static function AStaticMethod() { return 2; }
}
$anObject = new AClass();
$anObject->AStaticMethod();
?>
```

But you probably shouldn't encourage that kind of misunderstanding of software engineering, either by yourself or your class users.

Namespaces and aliases

Namespaces and aliases are brand-spankin' new features, found in PHP 5.3.0. They are new features by which you can add namespaces to your code, which work as you would expect from C++, and make more convenient aliases for just about anything. Read the **documentation** for more, though you may want to wait a little while before expecting to use it on a shared Web server or before releasing your code to the public in an open-source project.





Common problems and solutions

PHP has a clown car full of functions that help you solve common problems. Most are string functions or array functions. Here are some of the most common of the common.

Determining PHP's configuration

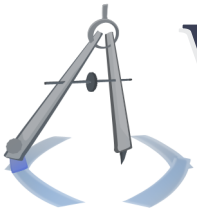
This is almost a joke answer, but in case you want to determine PHP's compile-time configuration, simply call the function `phpinfo()`.

Parsing or creating a delimited string

Assume you have a string like "one,two,three", and you want to separate it out into an array with "one", "two", and "three". You want PHP's `explode()` function ([docs](#)).

Example:





Website Architecture

Lezione 9 | PHP

PHP

```
<?php
$delimitedString = "ABC::XYZ::123";
$arrayOfSubstrings = explode("::", $delimitedString);
?>
```

The delimiter can be a string of any length. The "delimited string" may or may not contain a delimiter, and it will still work, as long as the string is not empty.

If you want to do the opposite, you can use **implode()** (**docs**). This allows you write to comma-delimited lists very easily.

Example:

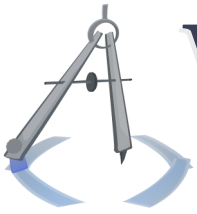
PHP

```
<?php
$arrayOfStrings = array("red", "green", "blue");

// Use this version to write the Oxford comma.
// $arrayOfStrings = array("red", "green", "and blue");

$delimitedString = implode(", ", $arrayOfStrings);
?>
```





Website Architecture

Lezione 9 | PHP

Trimming and transforming a string

You may have a string with extra whitespace on the left or right sides. Removal of this whitespace is called trimming. You can use `trim()` ([docs](#)), `ltrim()` ([docs](#)), or `rtrim()` ([docs](#)).

PHP

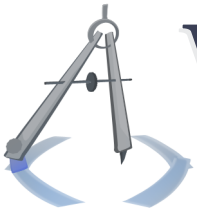
```
<?php
$untrimmedString = " \t Hello ";
$trimmedString = trim($untrimmedString);
?>
```

You may also want to transform a string to uppercase, transform it to lowercase, or capitalize each word. Use `strtoupper()` ([docs](#)), `strtolower()` ([docs](#)), or `ucwords()` ([docs](#)).

You may want to convert newlines to `
` tags so you can output some basic multi-line text as HTML. Use `nl2br()` ([docs](#)).

Finally, you can make arbitrary translations. For simple ones, like for one translation at a time, see `str_replace()` ([docs](#)). For more complex translations or bulk translations, like where you specify a mapping of multiple translations, see `preg_replace()` ([docs](#)).





Website Architecture

Lezione 9 | PHP

Text search

There are simple text-search functions, case-insensitive variants, and yet more subtle variants. Then there are more powerful, regular-expression-based search functions. For simple searches, start with `strpos()` ([docs](#)) and check the related functions at the bottom of the documentation page. For more complex ones, look at `preg_match()` ([docs](#)).

The PHP library has some awful inconsistencies in its earlier functions, and this includes `strpos()`. The parameters are in the opposite order as some other search functions. Be sure to check the documentation for any built-in function you use in PHP.

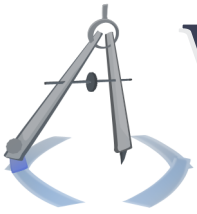
Examples:

PHP

```
<?php
$positionOfASDF = strpos("sdfafdagASDFsdfs", "ASDF");
$whetherPatternMatches
    = (preg_match("/P(-P)*owerBook", "P-P-Powerbook" > 0);
?>
```

To check if a substring exists at all, without using patterns, you





Website Architecture

Lezione 9 | PHP

essentially do a text search with `strpos()` and check the return value to see if it gives `false`. If it gives `false`, the substring was not found. As the manual tells you in big, bright letters, be sure to use strict equality to test that return value.

The function `preg_match()` may give you much more information, including the number of times the pattern matches, the parts of the string which match sub-patterns, etc.

Extracting substrings

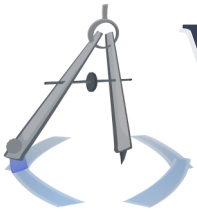
To retrieve a slice of a string, use `substr()` (`docs`) and branch out from there.

Sanitizing or escaping strings

You may want to build strings that are appropriate for output as C strings (or something with the same requirements) or HTML, just to name a few formats. There are functions which help to condition strings for these purposes.

For output as C strings (i.e. backslash-escaped quotes, etc.),





Website Architecture

Lezione 9 | PHP

look at `addslashes()` ([docs](#)) and its variants.

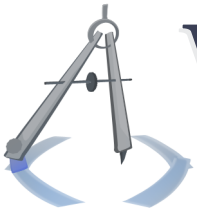
If you have text which should *not* contain HTML tags, like if it comes from user input and will eventually go into your document, then you want to remove any tags that may be there. Use `strip_tags()` ([docs](#)).

If you have trusted or sanitized text that may contain special characters, like ©, ampersands, or accented characters, you will want to escape the string to translate those to HTML character entity references. Use `htmlspecialchars()` ([docs](#)) or `htmlentities()` ([docs](#)). The latter is more thorough in a sense, because it uses the entity names (like "©") when applicable rather than the numeric codes. Of course, it is slower.

Array search

The function `array_search()` ([docs](#)) works like `strpos()`, although the order of the parameters is different.





Website Architecture

Lezione 9 | PHP

Random-number generation

Not too complicated; use `rand()` ([docs](#)) or the "better" `mt_rand()` ([docs](#)). Do not use them for cryptography, though - just to add spice to the user experience. Or perhaps for server maintenance tasks. Cryptographically-secure random number generators are a special breed. Read some of the comments on those pages for tips toward quick solutions.

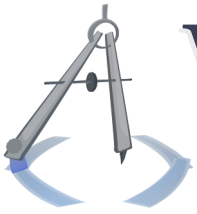
Hash functions

A hash function has very many uses. Two most common uses are:

- Compact representation of data - usually of a string. A hash function can take an arbitrary-length string and reduce it down to a fixed-length string, in a sense. It maps large numbers to small numbers, hopefully as uniformly and unpredictably as possible.

If you have a very long string, you may want a short analogue. Consider this problem: a user's full name is "Jonathan Bartholemew Copperwhipple III, Esquire". You have a large object full of other data on him, and you want to stick all such objects in a mapping, indexed by the people's names, since their full names





Website Architecture

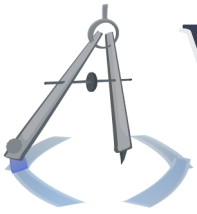
Lezione 9 | PHP

are unique. Should you make the object's key value "Jonathan Bartholemew Copperwhipple III, Esquire"? You could. But if all the key values are about 50 characters in length, it will take unnecessarily long to look up an entry. What if you could reduce the person's name to 20 bytes and still have a set of unique names? Clearly, that would be a win. So, what you would do in this case is hash all the names and then use the hash values as the map keys. In fact, PHP does this automatically for its array indices.

- Checking if two copies of some large data match each other. This builds off of the first one. If you have a 4 GB file and your friend on the Internet has a 4 GB file, how do you verify that they're the same? You could transfer the entire file to him and check it byte by byte, but that would take forever. You should hash your file, he should hash his file, and then you compare the hashes. Just 20 bytes. Hash functions are designed such that a small change in the original data should produce a large change in the hash, so this should work with overwhelmingly large probability. You could use a few different (independently-designed) hash functions to check; sending two or three hash values is still vastly cheaper than sending the whole thing.

Additionally, hash functions are supposed to be quick and practically irreversible (though someone could still find out by trial and error).





Website Architecture

Lezione 9 | PHP

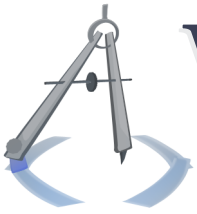
Hash functions are very useful and have a firm place in Web technology. An old standby is the MD5 algorithm, although it has lost some of its strength against cryptographic threats. It still may be used for the practical purposes stated above. Look into the `md5()` ([docs](#)) and `md5_file()` ([docs](#)) functions. Read their comments if you want to use them for passwords or anything serious, or search the Internet for such strategies (search for "md5 password"); you will find a lot.

A sort of competitor from the same generation is the SHA-1 function; see `sha1()` ([docs](#)).

A newer alternative to both is the SHA-256 algorithm, which is much more secure. However, PHP doesn't yet have a built-in function for this.

You will often want to combine hash functions with random numbers.





Interacting with the Web server

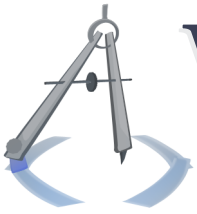
So far, we've mostly seen PHP's features which make it easier to produce Web pages. But PHP is also well-integrated into the Web server environment, including most notably the awareness and manipulation of HTTP headers.

The superglobals

If you want to get information from the server, your bread and butter is the variable called `$_SERVER` ([docs](#)). This basically contains the information that is usually sent to CGI programs via environment variables, though it has been nicely parsed into the PHP array. This variable is defined even if PHP is not installed as a CGI program but rather as an Apache module.

You can get pretty much whatever you need from this variable, in some indirect way or another. Use `print_r()` ([docs](#)) to see the contents of the array.





Website Architecture

Lezione 9 | PHP

Working as an Apache module

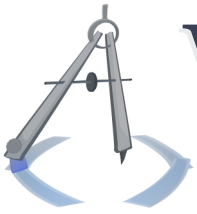
If you are installed as an Apache module, you can work more tightly with Apache - even bidirectionally. You will have some additional functions at your disposal. You can get the request headers line by line, instead of just getting their derived information from `$_SERVER`. And you can make sub-requests, or "virtual requests", via the abominably-named `virtual()` ([docs](#)). This is one way of doing a server-side include.

The entire list of Apache-specific functions is [here](#).

Reading request headers

If you want to read a request header, you may be able to do it directly or somewhat indirectly, as we have just mentioned. If you are installed as an Apache module, you can use `apache_request_headers()` ([docs](#)). Otherwise, you can get it from `$_SERVER`. If you use `$_SERVER`, you can (either read the documentation or) guess that if it's an important header, you can probably access it like this:





Website Architecture

Lezione 9 | PHP

PHP

```
<?php  
print($_SERVER["HTTP_ACCEPT"]); // Prints "xyz" of "Accept: xyz".  
?>
```

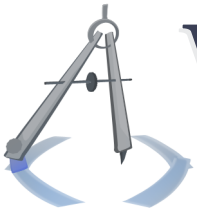
Writing response headers

Writing response headers is simple - perhaps too simple. Use the function `header()` (**docs**). One large caveat is that you must call this before any real text is sent over the wire. This includes whitespace, and, as mentioned in the section on PHP includes, this even includes the whitespace outside of your PHP parser tags (`<?php ?>`). If you send any real text, then PHP (and thus the Web server) will flush any headers that it already has, write a blank line in the transmission, and then send your text as the start of message body. So: be warned. This isn't really a flaw in PHP; it's just how HTTP works.

Transmission control and output buffering

So wait a second - you can buffer and flush the transmission? Yes; if you weren't aware of this before, almost all output in





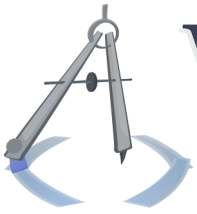
Website Architecture

Lezione 9 | PHP

computers is held in some buffer and sent to output when the buffer has gotten big enough that the transfer is worthwhile. This is fine for default behavior, and you should generally leave it alone unless you have some reason to send text as soon as physically possible. If you do want to send any pending text, which you would have written but the user would not have yet received, then use `flush()` (`docs`).

Separate from this, you can buffer the output in your own string variables, so that you may postprocess it or do some other kind of text-processing algorithm. Normally, you should avoid this, but it becomes almost prudent in some cases. You may need to use it to wrangle some unruly library which writes text to output before you can process it or customize it in some way, and you may not want to modify the library. You should try to avoid any significantly-sized buffering algorithm on Web servers, for a multitude of reasons but primarily due to hogging RAM. But in any case, in case you want to do this, check the **documentation** on output-control functions.





Website Architecture

Lezione 9 | PHP

Using cookies

Now, we'll touch on the subject of cookies. We'll come back to this later in more depth when we talk about sessions, which is the more modern technique that you should generally use instead of cookies. But for now: cookies it is.

A cookie in its raw form looks something like this:
name1=value&name2=value..

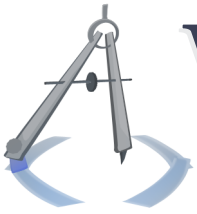
You can set each of those variables easily, in an individual manner, and you can read them just as easily. Any cookie values are parsed out and put into the superglobal variable **\$_COOKIE**:

PHP

```
<?php  
print($_COOKIE["name1"]);  
?>
```

To set a cookie, use the function **setcookie()** ([docs](#)). This also allows you to customize the cookie, including its lifespan and its domain specificity. Note that since this effectively sets the HTTP response header **Set-Cookie**, you have to call **setcookie()** before you write any real output.





Configuration options

PHP has some configuration options which you can set in advance or on the fly. There are compile-time configuration options for when you actually compile the PHP engine, but you probably only care about runtime configuration options. You can read the documentation on runtime configuration options [here](#); we will survey them in this section.

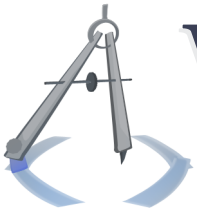
In general, a configuration option is a simple scalar value, like a boolean, integer, or string. They may be conceptually grouped together, but they are really just a bunch of mostly-independent options; the structure is fairly simple.

Opportunities for setting options

You may set options in a text file or at runtime. There are two types of text files; there is a main text file, and there are directory-specific text files. These files are called `php.ini` by default.

Each option - each integer/flag/etc. - has a certain level of im-





Website Architecture

Lezione 9 | PHP

portance. Some may only be set in the main configuration file, some may be set in any kind of configuration file, and some may be set by any of the three possibilities. Here is the **list of directives**.

To set a configuration option at runtime, use the function `ini_set()` ([docs](#)). You may also want to restore a configuration option to what it was before you screwed it up; see `ini_restore()` ([docs](#)).

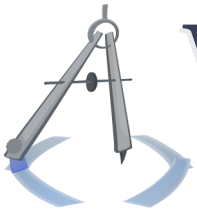
File syntax

The syntax of a configuration file is darn simple. Put a "name = value" pair on its own line, and boom. Comment lines start with a semicolon.

Important options

There are many configuration options, but here are some that give you an idea of what configuration options can do.





Website Architecture

Lezione 9 | PHP

Script execution time

By default, PHP scripts may execute for a maximum of 30 seconds (or whatever the `php.ini` says). After that point, they will simply stop, and depending on your error-reporting level, you may see an error. See `max_execution_time`. This just may be the reason that your script appears to fail; don't forget about it.

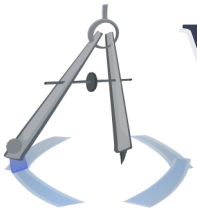
Include and file paths

When you open a file or make a PHP include, PHP actually searches a path in order to find your given URI, just like the `$PATH` environment variable in Unix. And, just like in Unix, it may contain multiple paths. These two configuration options are `include_path` and, of course, `open_basedir`.

Display of error messages

When your site goes live, you really don't want the user to see your PHP error messages, which would include a file and line number. Hide them with the option `display_errors`.





Website Architecture

Lezione 9 | PHP

There is a function `error_reporting()` ([docs](#)), but that has slightly different semantics. You may want to log or catch errors but still not display them to the user. Thus, you should do whatever handling you want and then set the configuration to not display errors.

The PHP documentation has a **whole section on error handling**. For a quick treat: the `@` operator may be placed before a function call to suppress errors:

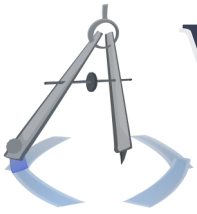
```
PHP
<?php
$x = @DangerousCall_SuppressingIsProbablyDumb_MaybeForDebugging();
?>
```

See the documentation [here](#).

Safe mode

PHP's most powerful configuration option is `safe_mode`. Safe mode is a sort of failed attempt at sandboxing: some functions, classes, and paths may be disabled, for instance. It turns out that this was more trouble than it was worth, and wasn't even guaranteed to be effective, so it has been off-





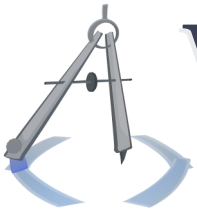
Website Architecture

Lezione 9 | PHP

cially deprecated in the newest version of PHP, after years of unofficial groaning and fist-rattling. The reason safe mode is important is that you may *be* in safe mode on your particular Web server, and you should know it. Just to be aware.

Safe mode as a whole was canned, but two of its related configuration options, **disable_functions** and **disable_classes**, remain. And they basically work. No, they don't just disable all function use or all class use. They let you specify which functions or classes should be disabled.





Performance notes

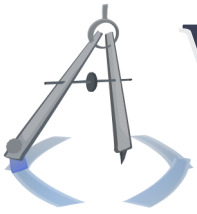
This is some advice that has been gleaned from around 7 years of use and 1-2 years of inspecting the C code whenever curious. This is written for PHP 5.2.6.

Remember the computer-science mantra: "Premature optimization is the root of all evil." It's true. So, you can consider these things, but consider them after you have a clear architecture, a correct implementation, and an actual need for better performance.

PHP components written in C

PHP's parsing & execution engines are written in C. This is a good thing for performance, but don't overestimate it simply because C is an efficient language. PHP's execution is basically a loop through all the PHP statements in a program, with a giant switch statement that goes through all possible statement types. It is a little blunt in this sense. If you've taken a course on CPU design, you can understand this in terms of opcodes; language interpreters are much like architectural simulators.



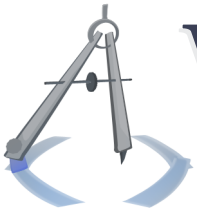


Website Architecture

Lezione 9 | PHP

In that same vein: Imagine the branch misses on that giant switch statement. You may want to run a though experiment and imagine how PHP is designed; all of the actual branches that the CPU evaluates are the branches in the C code. You may imagine that there are tons of branches in the C code, including ones which correspond to your PHP branches but also many others. There is probably one branch in C which corresponds to all branches within PHP; or maybe there are a handful. Thus, the behavior of the C branch is dissociated from the behavior of the PHP branches, looking from a statistical perspective. A branch predictor would be much less effective than it typically is with native execution, as it will almost certainly not pick up on the PHP program's behavior. Consequently, there is a trend in Web server hardware in which CPUs do not even have branch prediction; if there is ever a stall, they just switch to a new thread, since there is bound to be a constant demand of threads. Hence, it is good to have fairly computationally simple, streaming algorithms for Web programs, since it matches the hardware characteristics.





Website Architecture

Lezione 9 | PHP

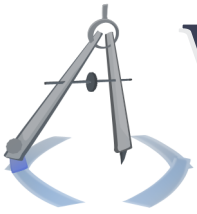
Now, back to higher levels of abstraction. As mentioned with PECL, some PHP (userspace) functionality is implemented in C. This is much better than the alternative, of course. So if you are somehow posed with the choice of writing your own library in PHP or using an existing one, which may be written in C (verify it), you should probably pick the one written in C. You may even want to write your own library in C if performance is that important.

String building

It often happens that you want to append to a string many times - perhaps you build a string by going through a loop and adding a space or a newline to the string ten times. Each time a string's length is modified in PHP, a new C string is allocated and the string value is copied over. Dynamic memory allocation is considered a more expensive operation, so you should minimize the frequency at which you do this.

If you regularly build the same string from scratch, you can prepare it in advance and just store it. The space tradeoff and perhaps even the break in semantics may be well worth it.





Website Architecture

Lezione 9 | PHP

If you cannot do that, maybe you want to minimize the number of times you allocate a new string - something like this:

PHP

```
<?php
function AddNewlines($SourceString, $NumberOfNewlines)
{
    $modifiedString = $SourceString;

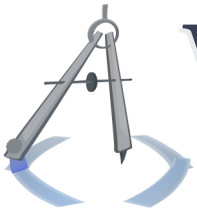
    // Knock off blocks of 10 if you can.
    while ($NumberOfNewlines >= 10)
    {
        $modifiedString += "\n\n\n\n\n\n\n\n\n\n";
        $NumberOfNewlines -= 10;
    }

    // Add the remaining newlines.
    for ($i = 0; $i < $NumberOfNewlines; $i++)
    {
        $modifiedString += "\n";
    }

    return $modifiedString;
}
?>
```

This is just an example. You decide whether something similar is appropriate for your application - if you can expect so many iterations that you can save by doing some in bulk, etc.





Website Architecture

Lezione 9 | PHP

Array access

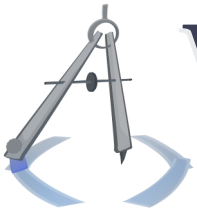
This is just to confirm that array access is generally pretty fast. A hash function is used to index into arrays, as arrays are really just hash tables in the implementation. Numbered arrays are no different from associative arrays in their performance.

Finally, in a similar manner to string building, arrays have to be reallocated and copied over once they reach certain size thresholds. There are far fewer reallocations, but the point isn't to memorize the number of allocations. The point is that if you think your application may resize the array often, perhaps to enormous sizes, you may want to just pre-allocate the array. It's a tradeoff and it's your call. Run some tests on this one. But you may want to use `array_fill()` ([docs](#)) and fill the array with something distinctive, like `null`.

Virtual function invocation

Typically, you might assume that virtual functions are more expensive if the override is called vs. the original implementation. In PHP, it appears to be the opposite.





Website Architecture

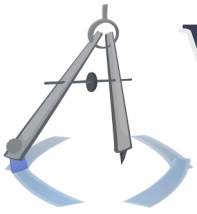
Lezione 9 | PHP

There are global variables in the PHP interpreter, some of which pertain to the execution context; they describe the current userspace context. One of these holds a pointer to the information structure of the "current class," which would be used to resolve method calls. Note that this is not the base class of a hierarchy - it's just the class of the object as it was instantiated. To resolve members that are in parent classes, the class information structure has an array of elements which point to the ancestors' information structures. So, ostensibly, the "current" class is checked first, and then the parents are traversed.

So if a method is defined in the most derived class, it is found right away. If it is defined in an ancestor class, it may actually take a few more pointer resolutions to find it.

Note that variables in PHP are untyped, so there is no such dilemma as "referring to a derived object by a pointer to the base".





Website Architecture

Lezione 9 | PHP

Type hinting

Keep in mind the above discussion on class information structures ("class entries"). When you use type hinting to enforce that a specific class - or its descendant - is accepted as a function parameter, that has to be checked somehow. That is checked at runtime, and it is re-checked every time a method is called; there are unfortunately no huge optimizations there. New classes may be defined at any time via includes or whatever dumb idea, so the interpreter cannot make those optimizations. Plus, nothing in PHP is that advanced.

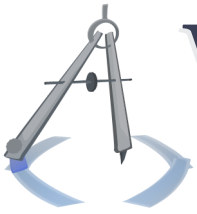
So, it checks the incoming object every time, and using its class entry, it traverses the array of ancestors to see if any of them match the one that was given in the type hint. Interfaces are lumped in the array with classes¹. The array may be organized forwards or backwards, so to speak². If you type hint the most ancestral class or the class's direct parent³, you may get significantly different performance. Run tests on your version & installation of PHP to be sure. And you could circumvent this

¹Their class entries are basically the same except for one flag.

²It's something the PHP implementors could conceivably change, so don't depend on one or the other without testing.

³The parent is actually kept in a separate variable next to the hierarchy array, for what that's worth.





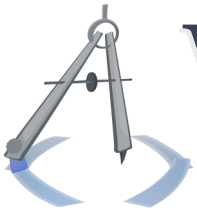
Website Architecture

Lezione 9 | PHP

class-entry traversal entirely by type hinting only for the class itself.

Type hinting is a great tool for software engineering, but you should just know that it isn't exactly cheap. If your application is suffering, look here for a possible optimization, though it may be a challenge to avoid sacrificing too much of your software's semantics.





Future directions

PHP has a large number of libraries and functions that can solve diverse sets of problems. For instance, you can create an image from scratch by using some PHP functions. You should be roughly aware of what PHP can do. Here are some modules which may help you in some common, obscure problems. There are more, including database libraries, but we will see those when we study databases.

These may or may not be in your installation of PHP, but they probably are.

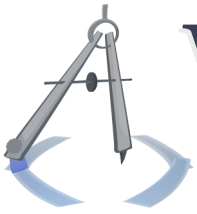
Filesystem (docs) A hodgepodge of analogues to Unix filesystem commands, C FILE* functions, and higher-level PHP conveniences. [here](#).

Directory (docs) Used to traverse directories.

Regular Expressions (docs) Perl-Compatible Regular Expressions (PCRE), which are the preferred flavor. Used mainly to search for patterns in text and to make replacements.

Multibyte strings (docs) Used to handle UTF-8, which is basically the Web standard for dealing with international alphabets.





Website Architecture

Lezione 9 | PHP

This provides alternate versions of the usual string functions which work on these strings. Also see `utf8_encode()` ([docs](#)).

SimpleXML ([docs](#)) Used to extract pieces of data from XML.
More XML-related modules [here](#).

GD ([docs](#)) Used to create images.

PDF ([docs](#)) Used to create PDF documents.

zlib ([docs](#)) Used to read and write gzip-compressed data.

zip ([docs](#)) Used to read and write zip-compressed data.

bzip ([docs](#)) Used to read and write bzip-compressed data.

