PHP

PHP Hypertext preprocessor

Information Technology Institute

COURSE MATERIALS

You can access the course materials via this link

http://goo.gl/ev41na

CONTENTS-DAY05

- Functions
- Closures
- Generators
- Traits
- Object Oriented
- XML Porcessing

REUSING CODE & WRITING FUNCTIONS

- I have said it many times before but when you have the same functionality multiple times in your code, then you are risking that when change comes you forget something. And it is not just that code that you have duplicated, you also have duplicate tests.

USING `require()` AND `include()`

- Using `a require()` or `include()` statement, you can load a file into your PHP script.
- `require_once()` and `include_once().` The purpose of these constructs is, as you might guess, to ensure that an included file can be included only once.

```
require( ‘reusable.php’ );
```

CALLING FUNCTIONS

- The following line is the simplest possible call to a function:

```
function_name();
```

- If you attempt to call a function that does not exist, you will get an error message:

```
Fatal error: calling to undefined function
```

- Note that calls to functions are not case sensitive, so calls to `function_name()`, `Function_Name()`, or `FUNCTION_NAME()` are all valid.

FUNCTION DEFINITION

- The declaration begins with the keyword function, provides the function name and parameters required, and contains the code that will be executed each time this function is called.

```
function my_function() {
   echo ‘My function was called’;
}
```

FUNCTION NAMING

- when naming your functions is that the name should be short but descriptive
- A few restrictions follow:
    - Your function cannot have the same name as an existing function.
    - Your function name can contain only letters, digits, and underscores.
    - Your function name cannot begin with a digit

VARIABLES SCOPE

- Variables declared **inside** a function are in scope from the statement in which they are declared to the closing brace at the end of the function. This is called ***function scope***. These variables are called ***local variables***.
- Variables declared outside functions are in scope from the statement in which they are declared to the end of the file, but ***not inside functions***. This is called ***global scope***. These variables are called ***global variables***.
- The special **superglobal** variables are visible both inside and outside functions.
- Using `require()` and `include()` statements does not affect scope. If the statement is used within a function, **function scope applies**. If it is not inside a function, **global scope applies**.

VARIABLES SCOPE

- The keyword **global** can be used to manually specify that a variable defined or used within a function will have **global scope**.
- Variables can be manually deleted by calling `unset($variable_name)`. A variable is no longer in scope if it has been unset.

OPTIONAL PARAMETERS

- Functions can have optional parameters.

```
function hello($name, $style = 'Formal'){
    switch ($style) {
        case 'Formal':
            print "Good Day $name";
            break;
        case 'Informal':
            print "Hi $name";
            break;
        case 'Australian':
            print "G'day $name";
            break;
        default:
            print "Hello $name";
            break;
    }
}
hello('Alice');  // Good Day Alice
hello('Alice', 'Australian');   // G'day Alice
```

VARIABLE-LENGTH ARGUMENT LISTS

- PHP 5.6 introduced variable-length argument lists, using the ... token before the argument name to indicate that theparameter is variadic.

```
function variadic_func($nonVariadic,
...$variadic) {
    echo json_encode($variadic);
}


variadic_func(1, 2, 3, 4); // prints [2,3,4]
```

PASSING BY REFERENCE VS BY VALUE

- If you want to write a function called `increment()` that allows you to increment a value, you might be tempted to try writing it as follows:

```
function increment($value, $amount = 1) {
    $value = $value +$amount;
}
```

- output from the following test code will be 10:

```
$value = 10;
increment ($value);
echo $value;
```

PASSING BY REFERENCE VS BY VALUE

```
function increment(&$value, $amount = 1) {
$value = $value +$amount;
}
```

- output from the following test code will be 11:

```
$value = 10;
increment ($value);
echo $value;
```

USING THE `return` KEYWORD

- The keyword `return` stops the execution of a function. When a function ends because either all statements have been executed or the keyword return is used.

```
function test_return() {
echo "This statement will be executed";
return;
echo "This statement will never be executed";
}
```

RETURNING VALUES FROM FUNCTIONS

- You can write the larger() function as follows:

```
function larger ($x, $y) {
if ((!isset($x)) || {!isset($y))) {
return false;
} else if ($x>=$y) {
return $x;
} else {
return $y;
}
}
```

WHAT IS CLOUSER?

- A closure is the PHP equivalent of an anonymous function, eg. a function that does not have a name. Even if that is technically not correct, the behavior of a closure remains the same as a function's, with a few extra features.
- A closure is nothing but an object of the Closure class which is created by declaring a function without a name. For example:

```
$myClosure = function() {
    echo 'Hello world!';
};
$myClosure(); // Shows "Hello world!"
```

USING EXTERNAL VARIABLES

- You can go further by creating "dynamic" closures. It is possible to create a function that returns a specific calculator, depending on the quantity you want to add. For example:

```
$quantity = 1;


$calculator = function($number)
use($quantity) {
    return $number + $quantity;
};


var_dump($calculator(2)); // Shows "3"
```

USING EXTERNAL VARIABLES

- It is possible, inside a closure, to use an external variable with the special keyword **use**. For instance:

```
function createCalculator($quantity) {
    return function($number) use($quantity) {
        return $number + $quantity;
    };
}
$calculator1 = createCalculator(1);
$calculator2 = createCalculator(2);
var_dump($calculator1(2)); // Shows "3"
var_dump($calculator2(2)); // Shows "4"
```

CLOSURE BINDING AND SCOPE

- You can bind a closure to a class using `bindTo`:

```
$myClosure = function() {
    echo $this->property;
};
class MyClass{
    public $property;
    public function __construct($propertyValue)    {
        $this->property = $propertyValue;
    }
}
$myInstance = new MyClass('Hello world!');
$myBoundClosure = $myClosure->bindTo($myInstance);

$myBoundClosure(); // Shows "Hello world!"
```

CLOSURE BINDING AND SCOPE

- Changing the property visibility to either **protected** or **private**. You get a **fatal** error indicating that you do not have access to this property.
- The only way for a property to be accessed if it's private is that it is accessed from a scope that allows it, ie. the class's scope

```
$myInstance = new MyClass('Hello world!');

$myBoundClosure = $myClosure-
>bindTo($myInstance, , MyClass::class);


$myBoundClosure(); // Shows "Hello world!"
```

CLOSURE BINDING AND SCOPE

- A closure created inside a method's class which is invoked in an object context will have the same scope as the method's:

```
class MyClass {
    private $property;
    public function __construct($propertyValue)    {
        $this->property = $propertyValue;
    }
    public function getDisplayer()      {
        return function() {
              echo $this->property;
        };
      }
}
$myInstance = new MyClass('Hello world!');
$displayer = $myInstance->getDisplayer();
$displayer(); // Shows "Hello world!"
```

BINDING A CLOSURE FOR ONE CALL

- Since PHP7, it is possible to bind a closure just for one call, thanks to the call method. For instance:

```
class MyClass {
    private $property;
    public function __construct($propertyValue)     {
        $this->property = $propertyValue;     }
}
$myClosure = function() {
    echo $this->property;
};
$myInstance = new MyClass('Hello world!');
$myClosure->call($myInstance); // Shows "Hello world!‟
```

- As opposed to the `bindTo` method, there is no scope to worry about. The scope used for this call is the same as the one used when accessing or invoking a property of `$myInstance`.

WHAT IS GENERATOR?

- Generators provide an easy way to implement simple iterators without the overhead or complexity of implementing a class that implements the Iterator interface.
- A `yield` statement is similar to a `return` statement, except that instead of stopping execution of the function and returning, yield instead returns a **Generator object** and pauses execution of the generator function.
- Generators are useful when you need to **generate a large collection** to later iterate over. They're a simpler alternative to creating a class that implements an Iterator

WHAT IS GENERATOR?

- This function generates an array that's filled with random numbers.
- What if we want to generate one million random numbers? randomNumbers(1000000) will do that for us, but at a cost of memory. One million integers stored in an array uses approximately 33 megabytes of memory.

```
function randomNumbers(int $length){
    $array = [];
    for ($i = 0; $i < $length; $i++) {
        $array[] = mt_rand(1, 10);
    }
    return $array; }
```

WHAT IS GENERATOR?

- Our `randomNumbers()` function can be re-written to use a generator.

```
function randomNumbers(int $length){
    for ($i = 0; $i < $length; $i++) {
        // yield tells the PHP interpreter that this value
        // should be the one used in the current iteration.
        yield mt_rand(1, 10);
    }
}
foreach (randomNumbers(10) as $number) {
    echo "$number\n";
}
```

OBJECT-ORIENTED PHP

- Class.
- Object.
- Methods.
- Attributes.

OBJECT-ORIENTED PHP

- Encapsulation - binds together the data and functions that manipulate the data, and that keeps both safe from outside. Data encapsulation led to the important OOP concept of data hiding.
- Inheritance - This allows classes to be arranged in a hierarchy that represents "is-a-type-of" relationships.
- Polymorphism. - Ability to take more than one form
- Abstraction - means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones.
- Reflection - allows inspection of classes, interfaces, fields and methods at runtime

REFLECTION

- Consider this example:

```
// Without reflection
$foo = new Foo();
$foo->hello();

// With reflection
$reflector = new ReflectionClass('Foo');
$foo = $reflector->newInstance();
$hello = $reflector->getMethod('hello');
$hello->invoke($foo);
```

CLASS DEFENTION

- You must use the keyword class, A minimal class definition looks like this:

```
class classname
{
}
```

CONSTRUCTOR

- Classes can define a special `__construct()` method, which is executed as part of object creation. This is often used to specify the initial values of an object:

```
class classname
{
   function __construct($param){
      echo "Constructor called with parameter ".$param."<br />";
   }
}
```

DESTRUCTOR

- You can use `__destruct()` to unset variables before the object goes out of scope:

```
class classname{
    function __destruct(){
    }
}
```

ANONYMOUS CLASSES

- Anonymous classes were introduced into PHP 7 to enable for quick one-off objects to be easily created. They can take constructor arguments, extend other classes, implement interfaces, and use traits just like normal classes can:

```
new class("constructor argument") {
    public function __construct($param) {
        var_dump($param);
    }
};
```

ACCESS MODIFIERS

```
class classname{
    $attribute;
    public $attribute1;
    Protected $attribute2;
    function operation2($param1, $param2){
    }
    private function operation1(){
    }
}
```

ACCESS MODIFIERS

- The default option is ***public***, meaning that if you do not specify an access modifier for an attribute or method, it will be public. Items that are public can be accessed from inside or outside the class.
- The ***private*** access modifier means that the marked item can be accessed only from inside the class. You may also choose to make some methods private, for example, if they are utility functions for use inside the class only. Items that are private will not be inherited.
- The ***protected*** access modifier means that the marked item can be accessed only from inside the class. It also exists in any subclasses; again, you can think of protected as being halfway in between private and public.

INSTANTIATE OBJECT

```
class classname
{
public $attribute;
function operation($param)
{
$this->attribute = $param
echo $this->attribute;
}
}
$a = new classname("First");
```

ACESSING ATRRIBUTES & METHODS

```
$a = new classname();

$a->attribute = "value";

echo $a->attribute;

$a-> operation(“value”);
```

DYNAMIC SETTERS AND GETTERS

```
class classname
{
public $attribute;
function __get($name)
{
return $this->$name;
}
function __set ($name, $value)
{
$this->$name = $value;
}
}
```

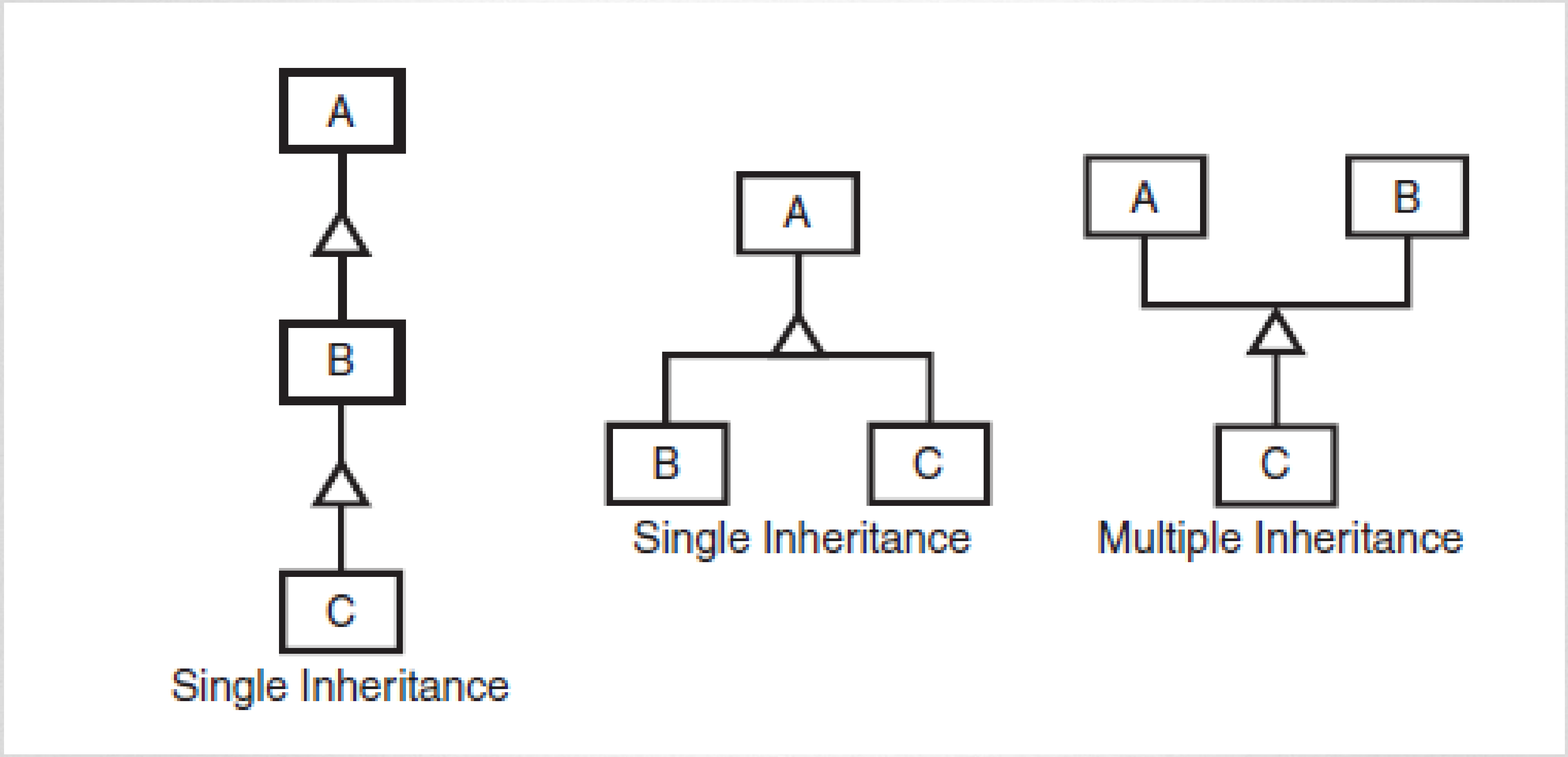
INHERTANCE

```
class B extends A
{
public $attribute2;
function operation2()
{
}
}
```

If the class A was declared as

```
class A
{
public $attribute1;
function operation1()
{
}
}
```

INHERTANCE TYPE



CALLING PARENT CONSTRUCTOR

```
class Foo {
   function __construct($args) {
         echo 'parent';
      }


}
class Bar extends Foo {
      function __construct($args) {
         parent::__construct($args);
      }
}
```

PREVENTING INHERITANCE & OVERRIDING

- PHP uses the keyword final. When you use this keyword in front of a function declaration, that function cannot be **overridden** in any subclasses. For example, you can add it to class A in the previous example, as follows:

```
final class A
{
    ……
}
```

PREVENTING INHERITANCE & OVERRIDING

```
class A

{

public $attribute = "default value";

final function operation()

{

echo "Something<br />";

echo "The value of \$attribute is ". $this-
>attribute."<br />";

}

}
```

STATIC METHODS AND CONSTANTS

```
class Math
{
const pi = 3.14159;
static function squared($input)
{
return $input*$input;
}
}
echo Math::squared(8);
```

USING self, $this AND static

- Use `$this` to refer to the current object. Use `self` to refer to the current class. In other words, use `$this->member` for non-static members, use `self::$member` for static members.

```
class Person {
    private $name;
public function __construct($name) {
        $this->name = $name;
    }
    public function getName() {
        return $this->name;
    }
    public function getTitle() {
        return $this->getName()." the person";
    }
    public function sayHello() {
        echo "Hello, I'm ".$this->getTitle()."<br/>";
    }
}
```

USING self, $this AND static

```
class Car {
    protected static $brand = 'unknown';
    public static function brand() {
        return self::$brand."\n";
    }
}
class Mercedes extends Car {
    protected static $brand = 'Mercedes';
}
class BMW extends Car {
    protected static $brand = 'BMW';
}
echo (new Car)->brand();
echo (new BMW)->brand();
echo (new Mercedes)->brand(); //unknown
```

USING self, $this AND static

```
class Car {
    protected static $brand = 'unknown';

    public static function brand() {
        return static::$brand."\n";
    }
}
class Mercedes extends Car {
    protected static $brand = 'Mercedes';
}
class BMW extends Car {
    protected static $brand = 'BMW';
}
echo (new Car)->brand(); // unknown
echo (new BMW)->brand();
echo (new Mercedes)->brand();
```

ABSTRACT CLASS

- `abstract operationX($param1, $param2);`
- Any class that contains abstract methods must itself be abstract, as shown in this example:

```
abstract class A

{

abstract function operationX($param1, $param2);

}
```

- The main use of abstract methods and classes is in a complex class hierarchy where you want to make sure each subclass contains and overrides some particular method; this can also be done with an interface.

INTERFACE

- The idea of an interface is that it specifies a set of functions that must be implemented in classes that implement that interface.

```
interface Displayable{
    function display();
}
class webPage implements Displayable
{
    function display(){
    // ...
    }
}
```

TRAITS

- PHP only allows single inheritance. Traits allow you to basically "copy and paste" a portion of a class into your main class

```
trait PrintableArticle {
    public function print() {
            /* code to print the article */
    }
}
class Article implements Cachable, Printable
{
    use PrintableArticle;
    use CacheableArticle;
}
```

POLYMORPHISM

- **Overloading.**
- **Overriding.**

DYNAMIC BINDING

- Dynamic binding, also referred as method overriding is an example of run time polymorphism that occurs when multiple classes contain different implementations of the same method, but the object that the method will be called on is *unknown* until run time.

```
interface Animal {
    public function makeNoise();
}
class Cat implements Animal {
    public function makeNoise
    {
        $this->meow();
    }
}
class Dog implements Animal {
    public function makeNoise {
        $this->bark();
    }
}
```

DYNAMIC BINDING

```
class Person {
    const CAT = 'cat';
    const DOG = 'dog';
    private $petPreference;
    private $pet;
    public function isCatLover(): bool {
        return $this->petPreference == self::CAT;
    }
    public function isDogLover(): bool {
        return $this->petPreference == self::DOG;
    }
    public function setPet(Animal $pet) {
        $this->pet = $pet;
    }
    public function getPet(): Animal {
        return $this->pet;
    }
}
```

DYNAMIC BINDING

```
if($person->isCatLover()) {
    $person->setPet(new Cat());
} else if($person->isDogLover()) {
    $person->setPet(new Dog());
}

$person->getPet()->makeNoise();
```

OVERLOADING METHODS WITH __call()

```
public function _call($method, $p){
   if ($method == "display") {
      if (is_object($p[0])) {
         $this->displayObject($p[0]);
      } else if (is_array($p[0])) {
         $this->displayArray($p[0]);
      } else {
         $this->displayScalar($p[0]);
      }
   }
}
```

CLONING OBJECTS

- The clone keyword, which allows you to copy an existing object, can also be used in PHP. For example,

```
$c = clone $b;
```

- You can also change this behavior. If you need nondefault behavior from clone, you need to create a method in the base class called __clone().

AUTOLOADING

- It is not a class method but a standalone function; that is, you declare it outside any class declaration. If you implement it, it will be automatically called when you attempt to instantiate a class that has not been declared.

```
function __autoload($name){

include_once $name.".php";

}


//PSR-0

//PSR-4

// Composer autoloading
```

OTHER MAGIC METHODS

`__sleep() and __wakeup()` // methods that are related to the serialization process. serialize function checks if a class has a __sleep method.

`__toString()` // Whenever an object is treated as a string, the __toString() method is called. This method should return a string representation of the class.

```
class User {
    public $first_name;
    public $last_name;
    public $age;
    public function __toString() {
        return "{$this->first_name} {$this->last_name}
($this->age)";
    }
}
// Casting to string calls __toString()
$user_as_string = (string) $user;
```

OOP

Made with ❤ by

MAGIC CONSTANTS

- `__FUNCTION__` returns only the name of the function whereas `__METHOD__` returns the name of the class along with the name of the function.
- `__CLASS__` magic constant returns the name of the class where it was defined.
- You can get the name of the current PHP file (with the absolute path) using the `__FILE__` magic constant.
- To get the absolute path to the directory where the current file is located use the `__DIR__` magic constant.

WHAT ARE NAMESPACES?

- The PHP community has a lot of developers creating lots of code. This means that one library's PHP code may use the same class name as another library. When both libraries are used in the same namespace, they collide and cause trouble.

- Namespaces solve this problem. As described in the PHP reference manual, namespaces may be compared to operating system directories that namespace files; two files with the same name may co-exist in separate directories. Likewise, two PHP classes with the same name may co-exist in separate PHP namespaces.

DECLARING NAMESPACES

- A namespace declaration can look as follows:

`namespace MyProject;` - Declare the namespace MyProject

`namespace MyProject\Security\Cryptography;` - Declare a nested namespace

`namespace MyProject { ... }` - Declare a namespace with enclosing brackets.

- It is recommended to only declare a single namespace per file, even though you can declare as many as you like in a single file

ACCESSING A NAMESPACE

- A namespace declaration can look as follows:

```
namespace MyProject\Shapes;
class Rectangle { ... }
```

- To use this namespace

```
$rectangle = new
MyProject\Shapes\Rectangle();
Or
use MyProject\Shapes\Rectangle;
$rectangle = new Rectangle();
```

- To use the global namespace just use \ before the class name

```
$date = new \DateTime();
```

WHAT IS THE DOM?

- It stands for Document Object Model
- The DOM defines a standard for accessing documents like XML and HTML.
- The DOM is a W3C (World Wide Web Consortium) standard.
- The DOM is separated into 3 different parts / levels:
  - Core DOM - standard model for any structured document
  - XML DOM - standard model for XML documents
  - HTML DOM - standard model for HTML documents

WHAT IS THE XML DOM?

- A standard object model for XML
- A standard programming interface for XML
- Platform- and language-independent
- A W3C standard.
- The XML DOM defines the objects and properties of all XML elements, and the methods (interface) to access them.

DOM NODES

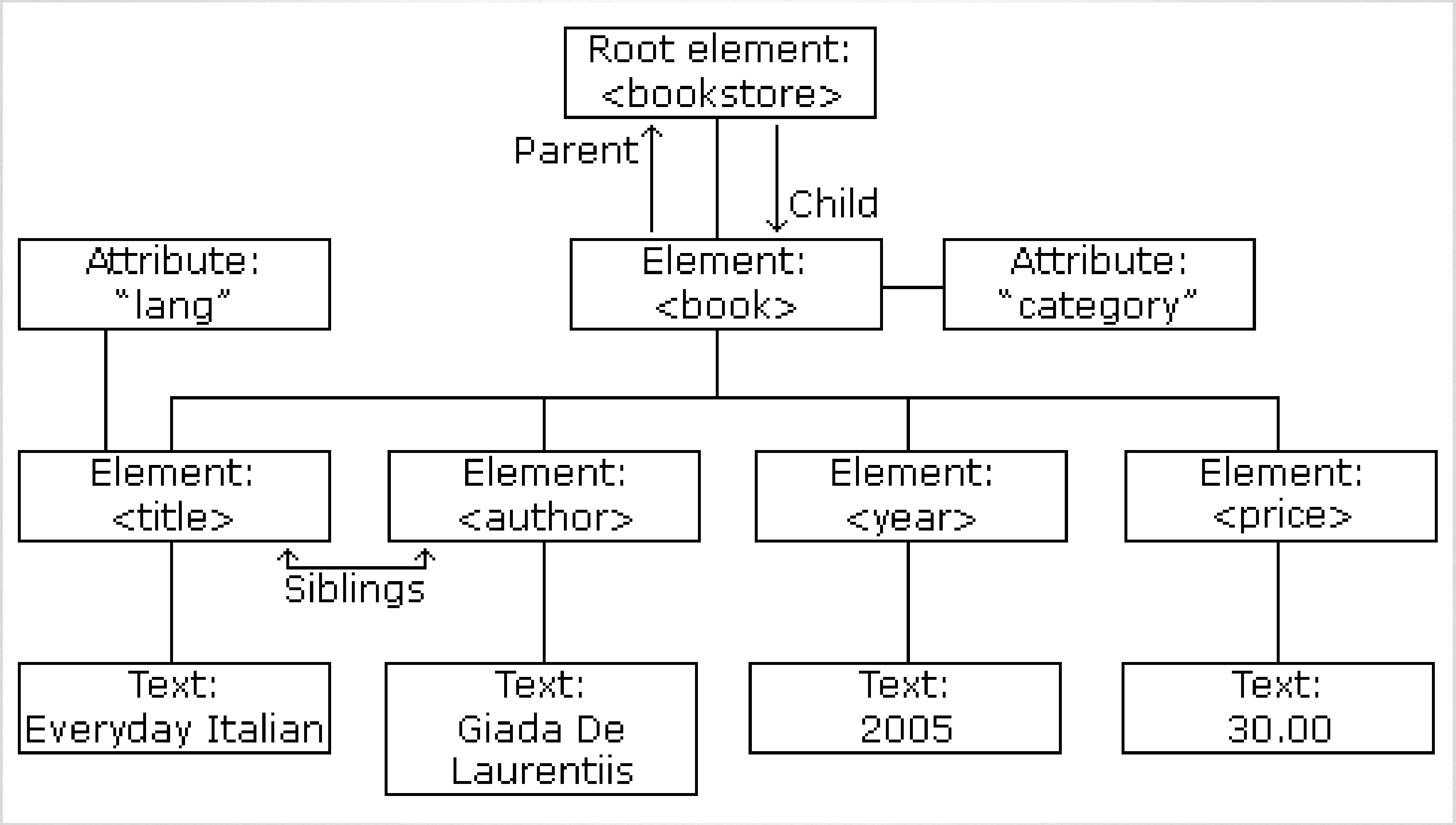
- According to the DOM, everything in an XML document is a node.
- The DOM says:
- The entire document is a document node
- Every XML element is an element node
- The text in the XML elements are text nodes
- Every attribute is an attribute node
- Comments are comment nodes

DOM NODE TREE

- Example : Books.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
<bookstore>
```

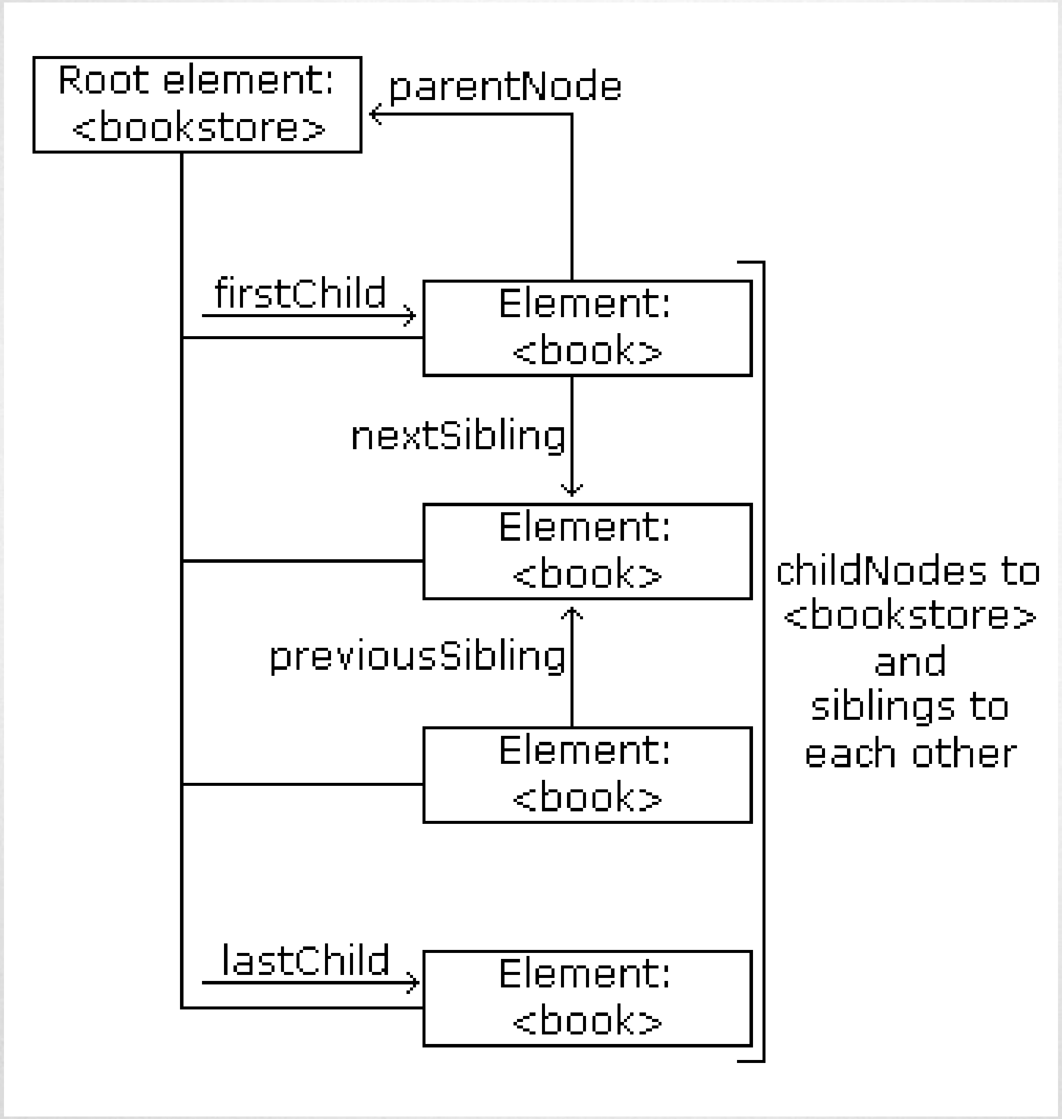
DOM NODE TREE



NODE PARENT, CHILDREN & SIBLINGS

- The nodes in the node tree have a hierarchical relationship to each other.`<?xml version="1.0" encoding="UTF-8"?>`
- The terms parent, child, and sibling are used to describe the relationships. Parent nodes have children. Children on the same level are called siblings (brothers or sisters).

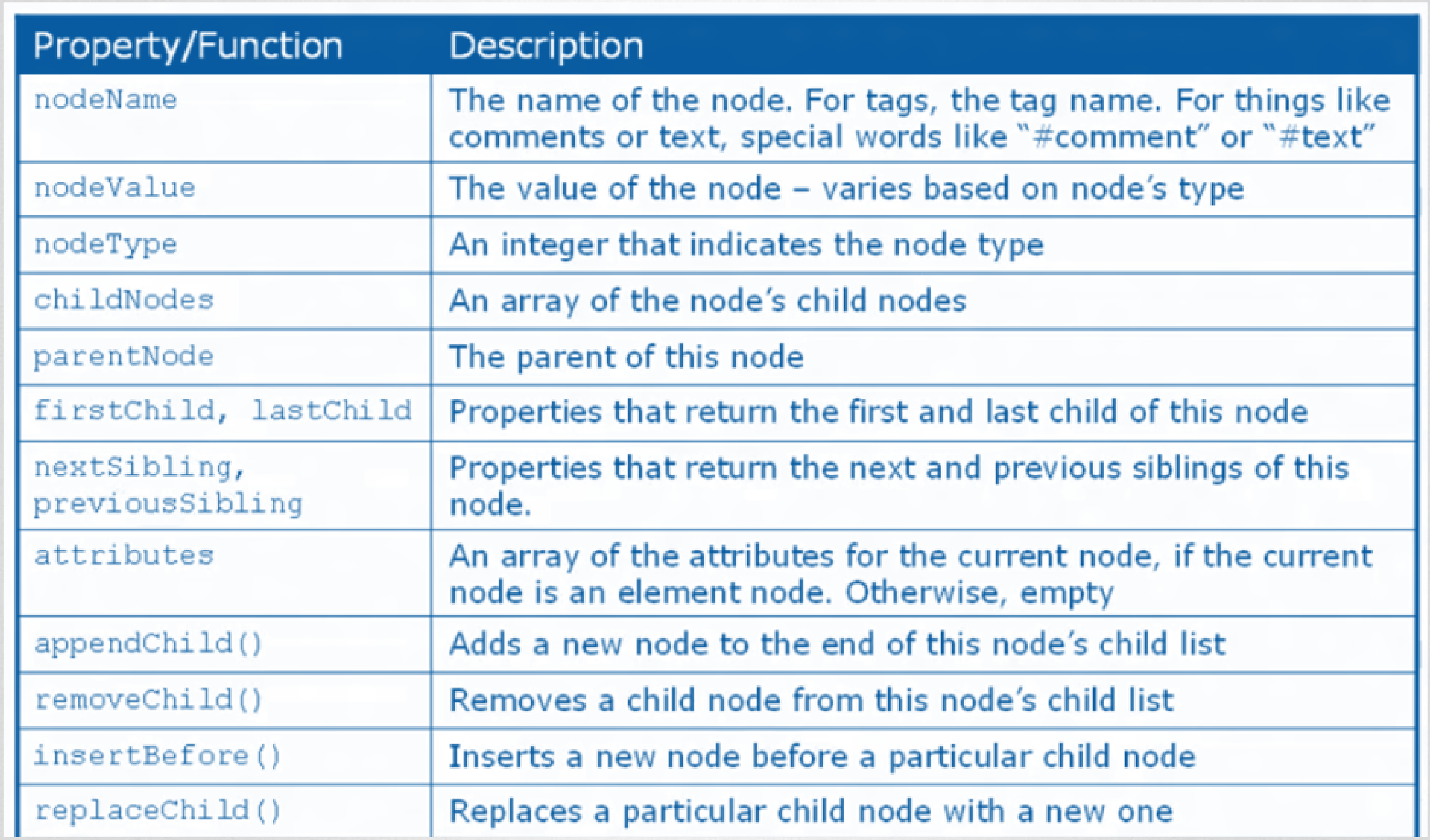
NODE PARENT, CHILDREN & SIBLINGS



READING XML

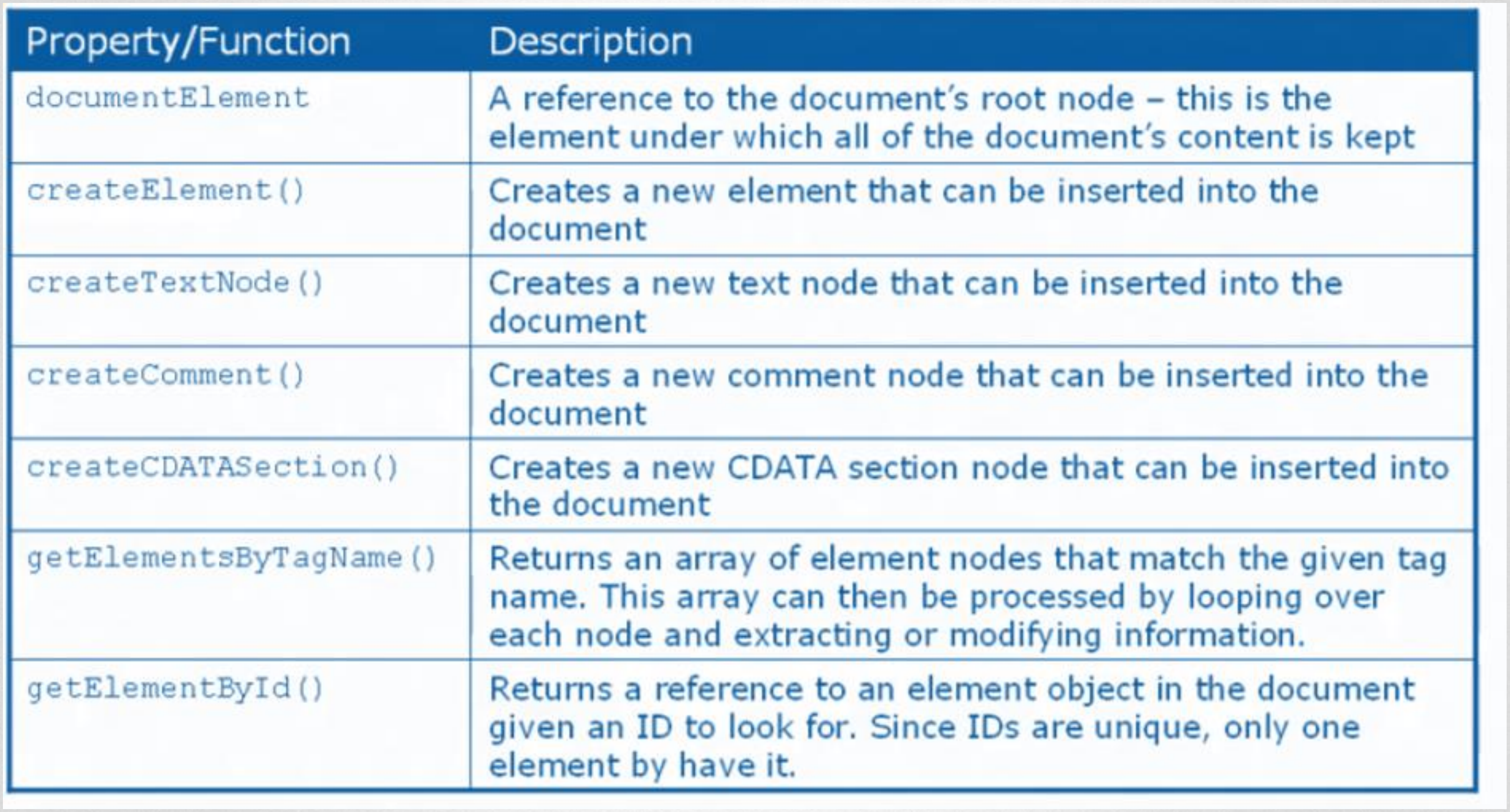
```
$doc = new DOMDocument();
  $doc->load( 'books.xml' );

  $books = $doc->getElementsByTagName( "book" );
  foreach( $books as $book )
  {
  $authors = $book->getElementsByTagName( "author" );
  $author = $authors->item(0)->nodeValue;

  $titles = $book->getElementsByTagName( "title" );
  $title = $titles->item(0)->nodeValue;

  echo "$title - $author\n";
  }
```

THE DOCUMENT OBJECT MODEL



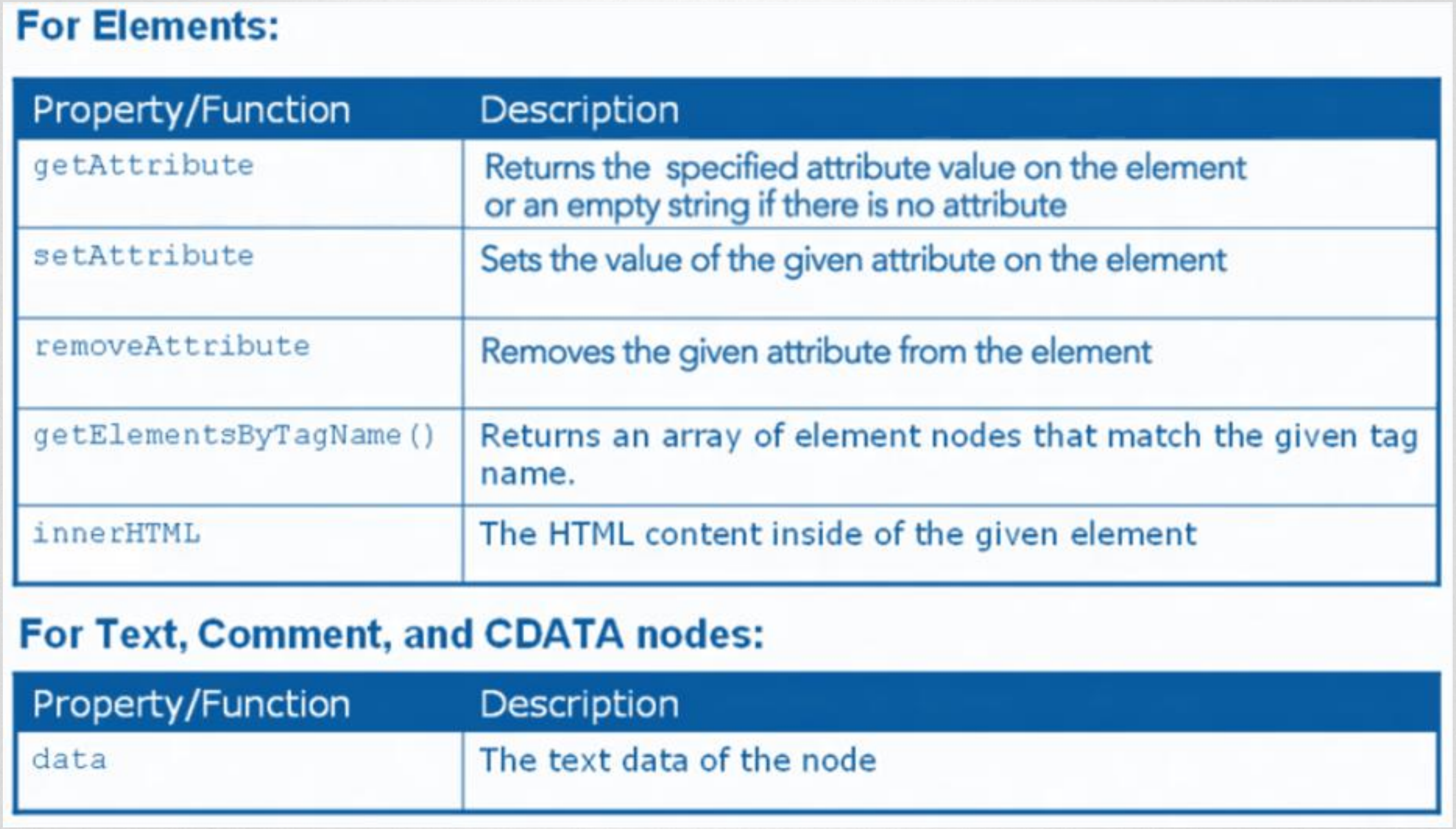
Property/Function	Description
`nodeName`	The name of the node. For tags, the tag name. For things like comments or text, special words like "#comment" or "#text"
`nodeValue`	The value of the node – varies based on node's type
`nodeType`	An integer that indicates the node type
`childNodes`	An array of the node's child nodes
`parentNode`	The parent of this node
`firstChild, lastChild`	Properties that return the first and last child of this node
`nextSibling, previousSibling`	Properties that return the next and previous siblings of this node.
`attributes`	An array of the attributes for the current node, if the current node is an element node. Otherwise, empty
`appendChild()`	Adds a new node to the end of this node's child list
`removeChild()`	Removes a child node from this node's child list
`insertBefore()`	Inserts a new node before a particular child node
`replaceChild()`	Replaces a particular child node with a new one

THE DOCUMENT OBJECT MODEL



Property/Function	Description
`documentElement`	A reference to the document's root node – this is the element under which all of the document's content is kept
`createElement()`	Creates a new element that can be inserted into the document
`createTextNode()`	Creates a new text node that can be inserted into the document
`createComment()`	Creates a new comment node that can be inserted into the document
`createCDATASection()`	Creates a new CDATA section node that can be inserted into the document
`getElementsByTagName()`	Returns an array of element nodes that match the given tag name. This array can then be processed by looping over each node and extracting or modifying information.
`getElementById()`	Returns a reference to an element object in the document given an ID to look for. Since IDs are unique, only one element by have it.

THE DOCUMENT OBJECT MODEL



For Elements:

Property/Function	Description
`getAttribute`	Returns the specified attribute value on the element or an empty string if there is no attribute
`setAttribute`	Sets the value of the given attribute on the element
`removeAttribute`	Removes the given attribute from the element
`getElementsByTagName()`	Returns an array of element nodes that match the given tag name.
`innerHTML`	The HTML content inside of the given element

For Text, Comment, and CDATA nodes:

Property/Function	Description
`data`	The text data of the node

THE DOCUMENT OBJECT MODEL

- Retrieving document information uses the “get” functions and informational properties

```
getElementByld()

getElexnentByTagName ()

getAttribute()

childNodes

firstChild, lastChild

data
```

- Once you have retrieved data, work with it like you would any other data in script
  - Append strings, add numbers, etc.

WRITING XML

```
<?php
  $books = array();
  $books [] = array(
  'title' => 'PHP Hacks',
  'author' => 'Jack Herrington',
);
  $books [] = array(
  'title' => 'Podcasting Hacks',
  'author' => 'Jack Herrington',
);
```

WRITING XML

```
$doc = new DOMDocument();
  $doc->formatOutput = true;


  $r = $doc->createElement( "bookstore" );
  $doc->appendChild( $r );
```

WRITING XML

```
foreach( $books as $book )
  {
  $b = $doc->createElement( "book" );
  $author = $doc->createElement( "author" );
  $author->appendChild(
  $doc->createTextNode( $book['author'] )
  );
  $b->appendChild( $author );
  $title = $doc->createElement( "title" );
  $title->appendChild(
  $doc->createTextNode( $book['title'] )
  );
  $b->appendChild( $title );
 $r->appendChild( $b );
}
echo $doc->saveXML();
```

THE DOCUMENT OBJECT MODEL

- Creating document content uses the create() functions available on the document object

```
createElement()
createTextNode ()
createComment()
createCDATASect ion ()
setAttribute()
```

- Once the content has been created, it must be added to the document

```
appendChild()
insertBefore()
Setting the innerHTML property directly
```

THE DOCUMENT OBJECT MODEL

- Before you can do anything DOM-related in PHP, you need to create a new instance of the DOM object, called DOMDocument:

```
$dom = new DOMDocument;
```

- Now you can load XML into `$dom`. DOM differentiates between XML stored as a string and XML stored in a file. To read from a string, call `loadXML( );` to read from a file, call `load( ):`

```
// read from a string
$dom->loadXML('<string>I am XML</string>');
// read from a file
$dom->load('address-book.xml');
```

THE DOCUMENT OBJECT MODEL

- If DOM encounters problems reading the XML—for example, the XML is not well-formed, your file does not exist, or you try to pass in an array instead of a string—DOM emits a warning. This example tries to load a string that's invalid XML:

```
$dom = new DOMDocument; // read non well-
formed XML document $dom->loadXML('I am not
XML');
```

- It causes DOM to give a PHP Warning that begins like this:

```
PHP Warning: DOMDocument::loadXML( ): Start
tag expected, '<' not found
```

THE DOCUMENT OBJECT MODEL

- Whitespace is considered significant in XML, so spaces between tags are considered text elements. For example, there are five elements inside the person element:

```
<person>
<firstname>Rasmus</firstname>
<lastname>Lerdorf</lastname>
</person>
```

THE DOCUMENT OBJECT MODEL

- However, removing the whitespace makes the document hard for humans to read. Happily, you can tell DOM to ignore whitespace:

```
$dom = new DOMDocument; // Whitespace is no
longer significant

$dom->preserveWhiteSpace = false;

$dom->loadXML('<string>I am XML</string>');
```

THE DOCUMENT OBJECT MODEL

- The root element of an XML document is stored as the documentElement property of a DOM object:

```
$dom = new DOMDocument;
$dom->preserveWhiteSpace = false;
$dom->load('address-book.xml');
$root = $dom->documentElement;
```

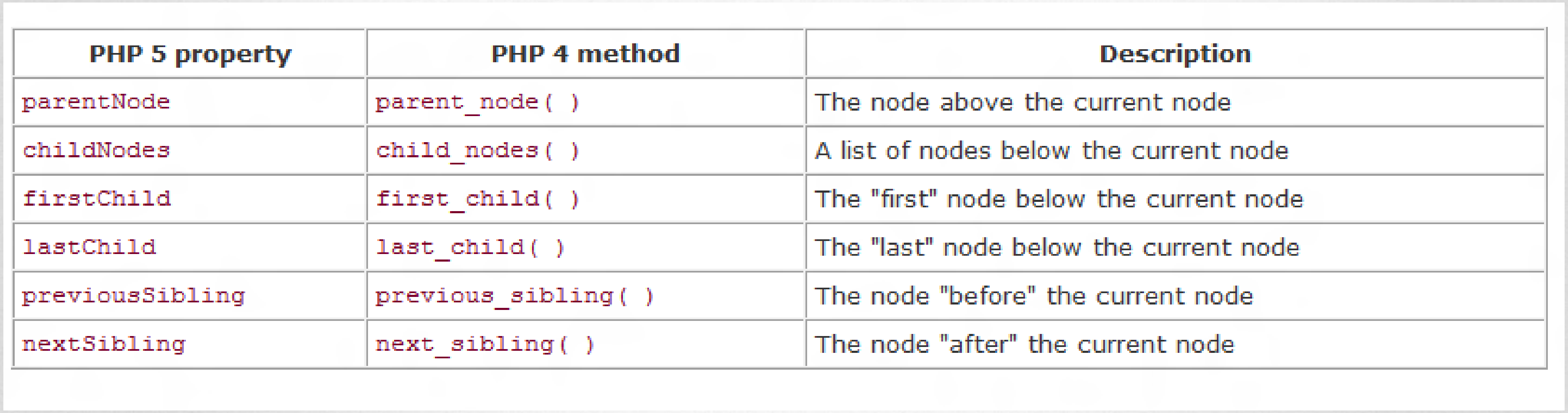
The $root variable now holds a pointer to the document root.

```
foreach ($root->childNodes as $person) {

    process($person);

}
```

THE DOCUMENT OBJECT MODEL

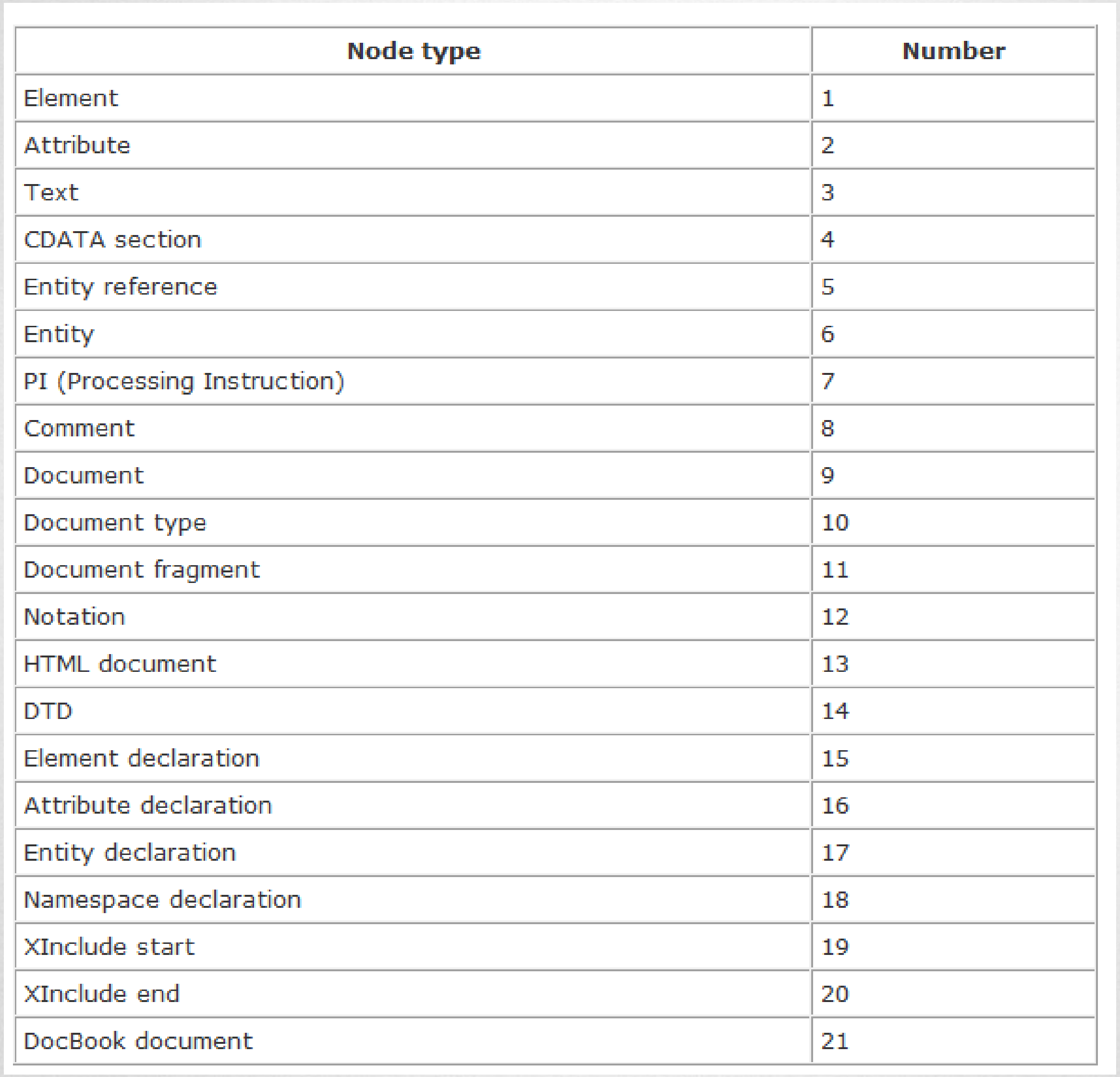
- For instance, the documentElement is always an element:

```
$root = $dom->documentElement;

print $root->nodeType;
```



PHP 5 property	PHP 4 method	Description
parentNode	parent_node( )	The node above the current node
childNodes	child_nodes( )	A list of nodes below the current node
firstChild	first_child( )	The "first" node below the current node
lastChild	last_child( )	The "last" node below the current node
previousSibling	previous_sibling( )	The node "before" the current node
nextSibling	next_sibling( )	The node "after" the current node

THE DOCUMENT OBJECT MODEL



Node type	Number
Element	1
Attribute	2
Text	3
CDATA section	4
Entity reference	5
Entity	6
PI (Processing Instruction)	7
Comment	8
Document	9
Document type	10
Document fragment	11
Notation	12
HTML document	13
DTD	14
Element declaration	15
Attribute declaration	16
Entity declaration	17
Namespace declaration	18
XInclude start	19
XInclude end	20
DocBook document	21

SIMPLEXML

- SimpleXML works by reading in the entire XML file at once and converting it into a PHP object containing all the elements of that XML file chained together in the same way. Once the file has been loaded, you can simply pull data out by traversing the object tree.
- The advantage of SimpleXML is that you no longer need to write any complicated code to access your XML,you simply load it, then read in attributes as you would expect to be able to

SIMPLEXML

```
<employees>
            <employee>
                    <name>Anthony Clarke</name>
                    <title>Chief Information
Officer</title>
                    <age>48</age>
            </employee>

            <employee>
                    <name>Laura Pollard</name>
                    <title>Chief Executive
Officer</title>
                    <age>54</age>
            </employee>
    </employees>
```

SIMPLEXML

- The base element is a list of employees, and it contains several employee elements. Each employee has a name, a title, and an age. Now take a look at this basic SimpleXML script:

```
$employees =
simplexml_load_file('employees.xml');
    var_dump($employees);
```

SIMPLEXML

```
object(simplexml_element)#1 (1) {
            ["employee"]=>
            array(2) {
                    [0]=>
                    object(simplexml_element)#2 (3) {
                            ["name"]=>
                            string(14) "Anthony Clarke"
                            ["title"]=>
                            string(25) "Chief Information
Officer"
                            ["age"]=>
                            string(2) "48"
                    }
```

SIMPLEXML

```
[1]=>
                    object(simplexml_element)#3
(3) {
                            ["name"]=>
                            string(13) "Laura
Pollard"
                            ["title"]=>
                            string(23) "Chief
Executive Officer"
                            ["age"]=>
                            string(2) "54"
                    }
             }
     }
```

SIMPLEXML

- you should be able to see that the base element has an array employee, containing two elementsone for each of the employees in the XML file. Each element in that array is another object, containing the name, the title, and the age of each employee.
- Now, consider the following script, using the same XML file:

```
    $employees =
simplexml_load_file('employees.xml');


    foreach ($employees->employee as $employee) {
            print "{$employee->name} is
{$employee->title} at age {$employee->age}\n";
    }
```

SIMPLEXML

- SimpleXML allows you to access attributes of XML elements as if the element were an array. Here's some very simple XML with attributes:

```
 <cakes>
          <cake type="sponge">
<name language="english">Victoria Cake</name>
          </cake>
   </cakes>
```

SIMPLEXML

- In that example, the cake element has a type attribute, and the name element has a language attribute. This next script accesses them both:

```
    $xml = simplexml_load_file("cakes.xml");
    print "{$xml->cake[0]["type"]}\n";
    print "{$xml->cake[0]-
>name["language"]}\n";
```

SIMPLEXML

- While `simplexml_load_file( )` loads XML data from a file, `simplexml_load_string( )` loads XML data from a string. This is generally not as useful, but it does allow you to load several XML files into one string, then use that inside one SimpleXML structure.

```
$employees = <<<EOT
<employees>
<employee ID="2" FOO="BAR">
<name>Anthony Clarke</name>
<title>Chief Information Officer</title>
<age>48</age>
</employee>
<employee ID="2" BAZ="WOM">
<name>Laura Pollard</name>
<title>Chief Executive Officer</title>
<age>54</age>
</employee>
</employees>
EOT;
```

SIMPLEXML

```
 $employees =
simplexml_load_string($employees);


    foreach ($employees->employee as
$employee) {
            print "{$employee->name} is
{$employee->title} at age {$employee-
>age}\n";
    }
```