PHP

André Restivo

Index

Introduction

PHP

- Originally called **P**ersonal **H**ome **P**age, it now stands for **P**HP: **H**ypertext **P**reprocessor, which is a recursive acronym
- Created by Rasmus Lerdorf in 1994.
- It is a **dynamically typed** programming language.
- Usually used to create dynamic web pages but can also be used to create standalone programs.

Hello World

The infamous hello world example in PHP:

```
<?php echo 'Hello World'; ?>
```

or even shorter

```
<?='Hello World;
```

PHP Delimiters

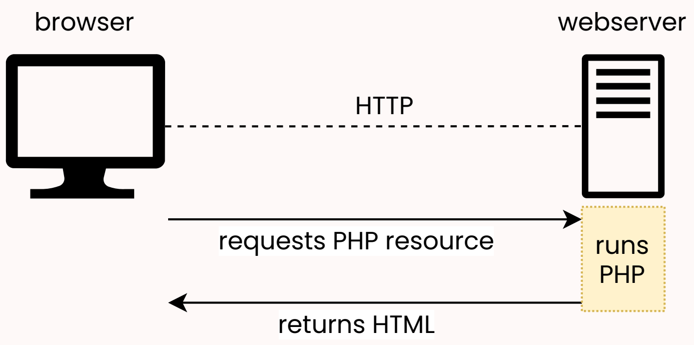
- The PHP interpreter only executes PHP code within its delimiters.
  Anything outside its delimiters is not processed by PHP.
- PHP code is delimited using "<?php" and "?>".
  In some cases "<?" and "?>" or "<script language="php">" and "</script>" also work.
- The purpose is to separate PHP code from non-PHP code (*e.g.*, HTML).
- After processing, the PHP code blocks are replaced by their output.

```
<section>
  <p><?php echo 'Hello World'?></p>
</section>
```

Becomes:

```
<section>
  <p>Hello World</p>
</section>
```

How It Works



1. Browser **asks** the server for a **resource** that corresponds to a PHP script.
2. Server **runs** PHP script.
3. Server **returns** result to the browser. Normally an HTML document.

Echo

- The echo function outputs one or more strings.
- It is not actually a function (it is a language construct), so you are not required to use parentheses with it.
- It also has a shortcut syntax, where you can immediately follow the opening tag with an equals sign.

```php
<?php echo 'Hello World'; ?>
```

The same as:

```php
<?='Hello World'?>
```

Comments

There are two ways of creating single-line comments:

```
echo 'Hello World'; // This line prints Hello Wor
echo 'Hello World'; # This line prints Hello Worl
```

Multi-line comments can also be used:

```
/**
 * The following line
 * prints Hello World
 */
echo 'Hello World';
```

Resources

- References:
  - http://php.net/manual/en/
- Books:
  - http://www.phptherightway.com/

Variables

Variables

- Variables are represented by a dollar sign followed by the variable's name.
- The variable's name is **case-sensitive**.
- There are no explicit type definition in variable declarations.
- A variable's type is determined by the context in which the variable is used.

```
$name = 'John';  // string
$age = 25;       // int
```

Data Types

PHP supports the following scalar types:

- bool: **case-insensitive** *true* or *false*.
- int: integer between *PHP_INT_MIN* and *PHP_INT_MAX*.
  Range is platform-dependent; converted to float in the case of an overflow.
- float: IEEE 754 double precision format.
- string: A series of characters.

We can find the type of a variable using the gettype function:

```php
$name = 'John';
echo gettype($name); // string
```

Assignment

- The variable type is defined when a value is assigned to it.
- Type can change when values of another type are assigned.
- Assignment is done by value unless the **&** sign is used.

```
$foo = 5;          // int
$foo = 'John';     // string

$bar = &$foo;      // by reference, bar and foo are
$foo = 'Mary';

echo $bar;         // Mary
```

Type Juggling

- PHP does automatic **type conversion** whenever it is needed.
- For example, the **+** (sum) operator expects two numerical values.

```
echo 5 + '10 potatoes'; // 15
```

- PHP automatically converts the string into an integer.

More on type juggling and why you should be careful.

Null Value

The special **null** value represents a variable with no value.

A variable is considered *null* if:

- it has been assigned the constant *null*.
- it has not been set to any value yet.
- it has been unset().

```
// $a starts as null
$a = 5;     // 5
$a = null; // null
$a = 10;   // 10;
unset($a); // null;
```

The constant **null** is **case-insensitive**.

Null Coalesce

The isset function determines if a variable is declared and different from *null*:

```
$bar = isset($bar) ? $bar : $some_default_value;
```

An easier way to acccomplish this would be to use the *null coalesce* operator:

```
$bar = $bar ?? $some_default_value;
```

Or even simpler:

```
$bar ??= $some_default_value;
```

Var Dump

The var_dump function displays structured information about one or more expressions, including their type and value.

Arrays and objects are explored recursively and their values are indented to show their structure.

```
$a = 10.5;
$b = true;
var_dump($a, $b);
```

```
float(10.5)
bool(true)
```

Very useful for dirty and straightforward debugging.

An alternative is print_r, a simplified form of *var_dump*.

Control Structures

Not so different from other languages

While

Executes the nested statement(s) repeatedly, as long as the while expression evaluates to *true*.

```
while($expr)
  do_something();
```

```
while($expr) {
  do_something();
  do_something_more();
}
```

```
while($expr):
  do_something();
  do_something_more();
endwhile;
```

Do While

Do...while loops are similar to *while loops*, but the expression is checked at the end of each iteration instead of at the beginning.

```
do {
  do_something();
} while($expr);
```

For

In for loops, the **first** expression is executed once unconditionally at the beginning of the loop.

At the beginning of each iteration, the **second** expression is evaluated. If it evaluates to *false*, the execution of the loop ends.

At the end of each iteration, the **third** expression is executed.

```
for ($i = 0; $i < 10; $i++)
  do_something($i);
```

```
for ($i = 0; $i < 10; $i++) {
  do_something($i);
  do_something_more($i);
}
```

If

Only if the expression evaluates to *true* does the following code execute.

```
if ($expr)
  do_something();
```

```
if ($expr) {
  do_something();
  do_something_more();
}
```

Else

The else statement extends an *if* statement to execute alternative code if the expression in the *if* statement evaluates to *false*.

```
if ($expr)
  do_something();
else
  do_something_else();
```

```
if ($expr)
  do_something();
else {
  do_something_else();
  do_something_more();
}
```

Break and Continue

Break ends execution of the current **for**, **foreach**, **while**, **do-while** or **switch** structure.

Continue skips the rest of the current loop iteration and continues execution at the condition evaluation.

```
while ($expr) {
  do_something();
  if ($foo) break;
  if ($bar) continue;
  do_something_more();
}
```

Switch

The switch statement is similar to a series of *if statements* on the same expression.

After finding a *true* condition, PHP continues to execute the statements until the end of the switch block, or the first time it sees a break statement.

```
switch($name) {
  case "John":
    do_something():
    do_something_more():
    break;
  case "Mary":
    do_something():
    break;
  default:
    do_something_else();
}
```

Die and Exit

Both die and exit stop the execution of the current PHP script.

- **die**: can receive a status string that will be printed before stopping
- **exit**: can receive a result integer that will be the exit status and not printed.

```
if ($something == "wrong") die ("Something is W
```

```
if ($everything == "ok") exit(0);
```

Loose and Strict Comparisons

Comparisons can be tricky in PHP. There are two types of equality operators:

Loose comparison: Types can be converted before comparison.

```
if ($a == $b) {   // != gives the opposite result
  do_something();
}
```

Strict comparison: Types must be the same.

```
if ($a === $b) {   // !== gives the opposite resu
  do_something();
}
```

Loose and Strict Comparisons

Some Examples:

```
if (1 == true)  // true - true is casted into the
if (1 === true) // false;
```

```
if (1 == "1")  // true - "1" is casted into the i
if (1 === "1") // false;
```

```
if (null == false)  // true
if (null === false) // false;
```

```
if ("Car" == true)  // true
if ("Car" === true) // false;
```

Learn more.

Strings

Strings

A string is a series of characters.

The simplest way to specify a string is to enclose it in *single quotes*.

```
$name = 'John';
```

A *single quote* inside a string defined using *single quotes* must be escaped using a **backslash**. To specify a literal backslash, double it.

```
$title = 'Ender\'s Game';
```

Single quoted strings don't recognize any other escape sequences.

Double Quote

If the string is enclosed in *double quotes*, more escape sequences for special characters are allowed (e.g. \r, \n, \t, \\, \", \$):

```
$title = "The quick brown fox\njumps over the laz
// The quick brown fox
// jumps over the lazy dog
```

Double quoted strings also expand any variables inside them.

```
$name = 'John';

echo 'This car belong to $name'; // This car belo
echo "This car belong to $name"; // This car belo
```

Some developers consider it a best practice to use single quotes when assigning string literals as they denote that there are no variables inside them.

Concatenation

The *sum* operator expects two numeric values as we have seen before. If used with strings, it will try to cast the strings into numbers, and sum them.

A different operator is used to concatenate strings together.

```
$name = 'John';
echo 'Hello World!' . " This is $name.";
```

Some String Functions

Returns the **length** of the given string:

```
int strlen (string $string)
```

```
echo strlen('John')   // 4
```

Find the numeric position of the **first occurrence** of *needle* in the *haystack* string starting at *offset*. Returns false if not found.

```
mixed strpos (string $haystack, mixed $needle [,
```

```
echo strpos ('abccba', 'bc');    // 1
echo strpos ('abccba', 'a');     // 0
echo strpos ('abccba', 'a', 2);  // 5
echo strpos ('abccba', 'bc', 2); // false
```

Some String Functions

Returns the **string portion** specified by the *start* and *length* parameters.

```
string substr (string $string, int $start [, int
```

```
echo substr('abcdefgh', 2, 4); // cdef
```

Returns a string with all occurrences of search in subject **replaced** with the given replace value. Also works with arrays.

```
mixed str_replace (mixed $search, mixed $replace,
                   mixed $subject [, int &$count
```

```
$text = str_replace("cd", "--", "abcdabcd", $coun
echo $text;  // ab--ab--
echo $count; //2
```

Some String Functions

Returns an array of strings, each of which is a substring of the initial string formed by **splitting** it on the boundaries defined by the string *delimiter*.

```
array explode (string $delimiter , string $string
```

Joins *pieces* from an array with a *glue* string.

```
string implode (string $glue , array $pieces)
```

```
$pieces = explode(' ', 'a b c'); // $pieces = arr
$text = implode('-', $pieces);   //$text = 'a-b-c
```

Arrays

Arrays

At first glance, PHP arrays might seem similar to arrays in other classical languages.

```
$values[0] = 5;  // although they don't need to b
$values[1] = 10; // and they don't have a fixed s
$values[2] = 20;

// count returns the size of the array
for ($i = 0; $i < count($values); $i++)
  $sum = $sum + $values[$i];

echo $sum / count($values); // calculates average
```

Arrays

An array is an **ordered map** organized as an ordered collection of **key-value** pairs.

Keys can be either **integers** or **strings**, and values can hold any data type. They can even hold different kinds of data in the same array.

```
$values['name'] = 'John';
$values['age'] = 45;
$values[3] = 'Car';
```

Creating Arrays

Arrays can be created just by using a variable as an array. Or they can be explicitly created using the **array()** function.

```
$values = array(); // this creates an empty array
```

They can also be initialized with some values.

```
$values = array(1, 2, 3, 'John');
// 0 => 1, 1 => 2, 2 => 3, 3 => 'John'
```

```
$values = array('name' => 'John', 'age' => 45, 3
```

Using Arrays

PHP will increment the **largest previously used** integer key when a key is not provided.

```
$values = array('name'=>'John', 'age'=>45, 2=>'Ca
$values[] = 'Boat';
// 'name'=>John, 'age'=>45, 2=>'Car', 3=>'Bicycle
```

Note that the largest previously used integer key does not need to exist in the array. It needs only to have been used as a key since the last time the array was re-indexed.

We can also have arrays as an array value:

```
$people = array(
  array('name' => 'John', 'age' => 45),
  array('name' => 'Mary', 'age' => 35);
);
echo $people[0]['name']; // John
```

Cycling Arrays

As arrays might not have sequential keys, like in other languages, in PHP we use the following construct to cycle through their **values**:

```
$values = array('name'=>'John', 'age'=>45, 2=>'Ca
foreach ($values as $value)
  echo "$value\n";
```

A similiar construct can be used to cycle through the **keys and values** simultaneously:

```
$values = array('name'=>'John', 'age'=>45, 2=>'Ca
foreach ($values as $key => $value)
  echo "$key = $value\n";
```

Some Array Functions

Searching for data:

Searches *haystack* for *needle* using loose comparison unless strict is set. Returns true if found, false otherwise.

```
bool in_array (mixed $needle,
               array $haystack [, bool $strict =
```

Returns the **key** for needle if it exists in the array, *false* otherwise.

```
mixed array_search (mixed $needle,
                    array $haystack [, bool $stri
```

Returns *true* if the given key **exists** in the array, *false* otherwise.

```
bool array_key_exists (mixed $key, array $array)
```

Some Array Functions

Sorting data:

Sorts an array such that array indexes **maintain** their **correlation** with the array elements they are associated with.
arsort does the same but in reverse.

```
bool asort (array &$array [, int $sort_flags = SO
```

Sorts an array by key, **maintaining** key to data **correlations**.
krsort does the same but in reverse.

```
bool ksort (array &$array [, int $sort_flags = SO
```

Sort Flags: **SORT_REGULAR**, **SORT_NUMERIC**, **SORT_STRING**, **SORT_LOCALE_STRING**, **SORT_NATURAL** and **SORT_FLAG_CASE**.

Learn more: php.net – array sorting

Some Array Functions

Random arrays:

This function **randomizes** the order of the elements in an array. Returns *true* on success or *false* on failure.

```
bool shuffle (array &$array)
```

Picks **one or more random entries** out of an array and returns the random entries' key (or keys).
When picking only one entry, returns the key, otherwise returns an array of keys.

```
mixed array_rand (array $array [, int $num = 1 ])
```

Some Array Functions

Used to assign a list of variables in one operation; not really a function but a language construct.

```
array list ( mixed $var1 [, mixed $... ] )
```

```
$values = array('John', 45, 'Bicycle');
list($name, $age, $vehicle) = $values;
echo $name;      // John
echo $age;       // 45
echo $vehicle;   // Bicycle
```

```
$values = array('John', 45, 'Bicycle');
list($name, , $vehicle) = $values; // skipping so
```

Many more functions: php.net – arrays

Functions

Functions

Any valid PHP code may appear inside a function, even other functions and class definitions.

Functions need not be defined before they are referenced, except when a function is conditionally defined.

Function names are **case-insensitive**.

To create a function, we use the *function* keyword:

```php
function doSomething() {
  echo "done";
}

doSomething(); // prints done
```

Parameters

By default, function parameters are passed by value. Parameters passed by reference are preceded by an ampersand (&).

```php
function sum($a, &$b) {
  return $a++ + $b++;
}

$a = 1; $b = 2;

echo sum($a, $b); // prints 3

echo $a;          // prints 1
echo $b;          // prints 3
```

Default Values

Function parameters can have default values.

Any parameters with a default value should appear after all parameters without defaults.

```
function sum($a, $b = 0, $c = 0) {
  echo $a + $b + $c;
}

sum(1);     // prints 1
sum(1,2);   // prints 3
sum(1,2,3); // prints 6
```

Returning Values

Functions can return values.

The type of the returned value does not need to be specified. A function can even return different types of values depending on some condition.

Functions that do not return a value return *null*.

```php
function sum($a, $b = 0, $c = 0) {
  return $a + $b + $c;
}

echo sum(1);     // prints 1
echo sum(1,2);   // prints 3
echo sum(1,2,3); // prints 6
```

Returning Multiple Values

There is no way for a function to return multiple values.

But we can achieve a similar result using *arrays* and the *list* construct.

```
function sort2($a, $b) {
  if ($a > $b) return array($b, $a);
  else return array($a, $b);
}

list($smaller, $larger) = sort2(10, 5);

echo $smaller; // 5
echo $larger;  // 10
```

Global

As PHP variables do not need to be defined before usage, we need to declare global variables as global to use them inside functions.

```php
function foo() {
  echo $baz;
}

function bar() {
  global $baz;
  echo $baz;
}

$baz = 10;

foo(); // prints nothing, may result in a warning
bar(); // prints 10
```

Coercive Typing

PHP is a dynamically typed language, but since PHP 7, it is possible to add type hints to function **parameters**:

```
function add($a, $b) {
  return $a + $b;
}

echo add(1, 4);             // 5
echo add(1.2, 3.6);         // 4.8
echo add("1.2", "3.6");     // 4.8
```

With *type hints*, types are **coerced** into the correct type (if possible):

Otherwise an error is thrown.

```
function add(int $a, int $b) {
  return $a + $b;
}

echo add(1, 4);             // 5
echo add(1.2, 3.6);         // 4
echo add("1.2", "3.6");     // 4
```

Coercive Typing

Type hints can also be added to **return values**:

```
function add($a, $b) : int {
  return $a + $b;
}

echo add(1, 4);            // 5
echo add(1.2, 3.6);        // 4
echo add("1.2", "3.6");  // 4
```

Returned values are **coerced** into the correct type (if possible).
Otherwise an error is thrown.

Besides the four scalar types, the following are also possible type hints: *array*, *object*, a *class/interface name*, *callable*, *iterable*, *self*, and *parent*.

Strict Typing

It is possible to enable **strict mode** on a per-file basis by using this directive at the beginning of a PHP file:

```
declare(strict_types=1);
```

In strict mode, only values corresponding to the type declaration will be accepted.
The only exception to this rule is that an **int** value will pass a **float** type declaration.

```
declare(strict_types=1);

function add(int $a, int $b) : int {
  return $a + $b;
}

echo add(1, 4);           // 5
echo add(1.2, 3.6);       // Error
echo add("1.2", "3.6");   // Error
```

Nullable Types

Sometimes we want to declare a type but also **allow the null value**.

This can be achieved by **prefixing** the type name with a '?':

```
declare(strict_types=1);

function add(?int $a, ?int $b) : ?int {
  if ($a === null || $b === null) return null;

  return $a + $b;
}

echo add(1, 4);           // 5
echo add(1, null);        // null
```

Nullable types also work with return values.

Classes

Classes

PHP 5 marked the introduction of a brand new **object model** for PHP.

Every class starts with the word *class* followed by its *name* and the *class definition* (inside curly brackets):

```
class Car {
  // class definition goes here
}
```

Properties

Properties are defined by using the visibility keywords **public**, **protected**, or **private**, followed by a variable declaration.

This declaration may include an initialization, but this initialization must be a constant value.

```
class Car {
  private $plate = '12-34-AB';
  private $driver = 'John Doe';
}
```

Properties can also be *coercively* and *strictly* typed:

```
class Car {
  private string $plate = '12-34-AB';
  private string $driver = 'John Doe';
}
```

Methods

Methods are like functions that have access to the private properties of the class. They also have the same visibility keywords as properties.

However, due to the dynamic typed nature of PHP, to access these properties the pseudo-variable **$this** must be used:

```
class Car {
  private $plate;
  private $driver = 'John Doe';

  public function getDriver() : string {
    return $this->driver; // return $driver would
  }
}
```

Methods can also be *coercively* and *strictly* typed.

Creating

To create an instance of a class, we use the **new** keyword.

An object will always be created unless the object has a constructor defined that throws an exception on error.

```php
$car = new Car();
```

Constructors

PHP allows developers to declare constructor methods for classes.

Classes that have a constructor method, call this method on each newly-created object.

The constructor method is always called **__construct** and can receive any number of parameters. The destructor method is, as expected, called **__destruct**.

```
class Car {
  private $plate;
  private $driver;

  public function __construct($driver, $plate) {
    $this->driver = $driver;
    $this->plate = $plate;
  }

}

$car = new Car('John Doe', '12-34-AB');
```

Extends

A class can inherit the methods and properties of another class by using the keyword extends in the class declaration.

Extending from multiple classes is impossible; a class can only inherit from one base class.

```
class RaceCar extends Car {

  // Specific race car definitions

}
```

Static

The static keyword allows us to define static properties and methods shared between all class instances.

```
class Car {
  static public $mile = 1.609344; //km
  // ...
}

echo Car::mile;
```

Static members can be accessed using the name of the class and the :: operator.

Obviously, **$this** cannot be used inside a static method.

Scope

These are used to access **static properties or methods** from inside the class definition:

- **self::** - the current class
- **parent::** - the parent class
- **static::** - the class of the current object

```
class Car {
  static private $mile = 1.609344; //km

  public function __construct($driver, $plate) {
    parent::__construct($driver, $plate);
  }

  public static function milesToKm($miles) {
    return $miles * static::mile;
  }
}

echo Car::milesToKm(10);
```

Self vs. Static

```php
class Foo
{
  protected static $bar = 'fizz';

  public function print() {
    echo static::$bar;
    echo self::$bar;
  }
}

class Bar extends Foo
{
  protected static $bar = 'buzz';
}

$foo = new Foo();
$bar = new Bar();

$foo->print();  // fizz fizz
$bar->print();  // buzz fizz
```

Read more.

Abstract

- Classes defined as abstract may not be instantiated.
- Classes that contain abstract methods must be abstract.
- Methods defined as abstract do not have an implementation.

```php
abstract class Car {
  private $plate;
  private $driver = 'John Doe';

  public function getDriver() {
    return $this->driver;
  }

  abstract public function getPlate();
}
```

Interfaces

- We use the **interface** keyword to define an interface, just as we use the *class* keyword to define a class.
- Interfaces are just like classes, but their methods do not have an implementation.
- The **implements** keyword specifies that a specific class implements the interface.

```php
interface Car {
  public function getDriver() : string;
  public function getPlate() : string;
}

class RaceCar implements Car {
  private $plate;
  private $driver;

  public function getDriver() : string {
    return $this->driver;
  }

  public function getPlate() : string {
    return $this->plate;
  }
}
```

Final

The *final* keyword prevents child classes from overriding a method.

If the class itself is *final*, it cannot be extended.

```php
final class RaceCar implements Car {
  private $plate;
  private $driver;

  public function getDriver() : string {
    return $this->driver;
  }

  final public function getPlate() : string {
    return $this->plate;
  }
}
```

Exceptions

Exceptions

Exceptions are events that disrupt the normal flow of instructions.

As in other programming languages, exceptions can be **thrown** and **caught**.

To throw an exception we use the throw keyword:

```
if ($db == null)
  throw new Exception('Database not initialized')
```

Exceptions

Exception is a class with the following public methods:

```
final public string getMessage ();
final public Exception getPrevious ();
final public mixed getCode ();
final public string getFile ();
final public int getLine ();
final public array getTrace ();
final public string getTraceAsString ();
```

User-defined exceptions can be defined by extending the built-in *Exception* class.

Try and Catch

The **try-catch** statement consists of a try block followed by one or more catch clauses, which specify handlers for different exceptions.

```
try {
  $car = getCar($id);
} catch (DatabaseException $e) {
  echo 'Database error: ' . $e->getMessage();
} catch (Exception $e) {
  echo 'Unknown error: ' . $e->getMessage();
}
```

Databases

PDO

The PHP Data Objects (PDO) extension defines a lightweight, consistent interface for accessing databases in PHP.

Connecting

To connect to a database, we use a PDO object.

The connection string is database-dependent.

```
$dbh = new PDO('mysql:host=localhost;dbname=test'
```

```
$dbh = new PDO('pgsql:host=localhost;port=5432;db
               $user, $pass);
```

```
$dbh = new PDO('sqlite:database.db');
```

Prepared Statements

Prepared statements (first prepare, then execute) are the recommended way of executing queries as they prevent **SQL injection** attacks (more on this later):

```
$stmt = $dbh->prepare('INSERT INTO person (name,
                       VALUES (:name, :address)')

$stmt->bindParam(':name', $name);
$stmt->bindParam(':address', $address);

$stmt->execute();
```

Prepared Statements

Another form of prepared statements:

```php
$stmt = $dbh->prepare('INSERT INTO person (name,
                       VALUES (?, ?)');

$stmt->execute(array($name, $address));
```

Values are bound to each question mark by their order.

Getting Results

To get the query results, we use the fetch function.

This function fetches one row at a time and returns *false* if there are no more rows.

```
$stmt = $dbh->prepare('SELECT * FROM person WHERE
$stmt->execute(array($name));

while ($row = $stmt->fetch()) {
  echo $row['address'];
}
```

Getting Results

The fetchAll function returns the complete result as an array of rows.

```
$stmt = $dbh->prepare('SELECT * FROM person WHERE
$stmt->execute(array($name));

$result = $stmt->fetchAll()

foreach ($result as $row) {
  echo $row['address'];
}
```

Using *fetchAll* might take too much memory if the result size is substantial.

Fetch Mode

Query results can return results in several different modes. Some of them:

- PDO::FETCH_ASSOC: returns an array indexed by column name.
- PDO::FETCH_NUM: returns an array indexed by column number.
- PDO::FETCH_BOTH (default): returns an array indexed by both column name and 0-indexed column number.

Changing the default fetch mode (has to be done every time a connection is created):

```
$dbh->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE,
```

Transactions

Unfortunately, not every database supports transactions, so PDO needs to run in what is known as "auto-commit" mode when you first open the connection.

If you need a transaction, you must use the beginTransaction method to initiate one.

```
$dbh->beginTransaction();

// queries go here

$dbh->commit; // or $dbh->rollBack();
```

To close a transaction we can either use commit or rollBack.

Error Handling

PDO offers you a choice of 3 different error handling strategies:

- **PDO::ERRMODE_SILENT** The default mode. No error is shown. You can use the errorCode() and errorInfo() on both database and statement objects to inspect the error.
- **PDO::ERRMODE_WARNING** Similar to previous one but a warning is shown.
- **PDO::ERRMODE_EXCEPTION** In addition to setting the error code, PDO will throw a PDOException and set its properties to reflect the error code and error information.

Error Handling

Setting the default error handling strategy:

```
$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMOD
```

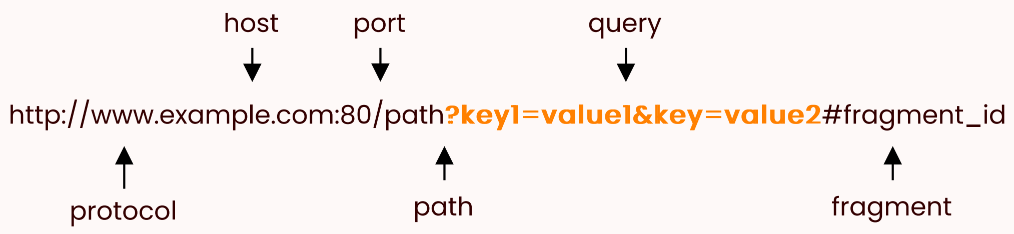
Using PDO exceptions:

```
try {
  $stmt = $dbh->prepare("SELECT * FROM person WHE
  $stmt->execute(array($name));

  $result = $stmt->fetchAll()
} catch (PDOException $e) {
  // Do something about it...
  echo $e->getMessage();
}
```

HTTP Parameters

Query String

The **query string** allows extra information to be sent to a webserver when requesting a resource.


host
port
query
http://www.example.com:80/path?key1=value1&key=value2#fragment_id
protocol
path
fragment

```
<a href="newsitem.php?id=10">
```

```
<form action="search.php" method="get"> <!-- sear
  <input type="search" name="q">
  <button type="submit">
</form>
```

For a form with *method="post"*, it works the same way but the information is sent separately from the URL (more on this later).

HTTP Parameters

Extra information sent to a resource, can be accessed in a PHP script using two different arrays, **$_GET** and **$_POST**, depending on the way the information was sent.

```
<a href="newsitem.php?id=10">
```

```
$id = $_GET['id'];  // On newsitem.php
```

```
<form action="search.php" method="post">
  <input type="search" name="q">
  <button type="submit">
</form>
```

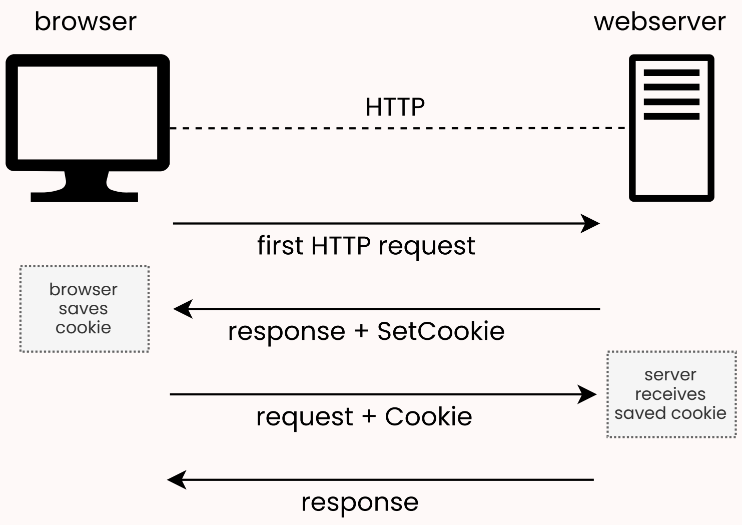
```
$q = $_POST['q'];  // On search.php
```

These arrays are **superglobal**, or **automatically global**, variables. There is no need to do *"global $variable;"* to access them within functions or methods.

Sessions

Cookies

- The HTTP protocol is a **stateless** protocol.
- No state information is stored on the server.
  Every request must be understood in isolation.
- Cookies are a mechanism for storing data in the browser.
  That is sent to the server in every request.



Cookies

Cookies can be set using the **setcookie** function:

```
bool setcookie (string $name, string $value, int
                string $path, string $domain, boo
                bool $httponly = false)
```

- All parameters are optional except the *cookie name*.
- Cookies must be sent before any output from your script.
  This is an HTTP protocol restriction.
- This requires that you place calls to this function prior to any output, including any **whitespace**.

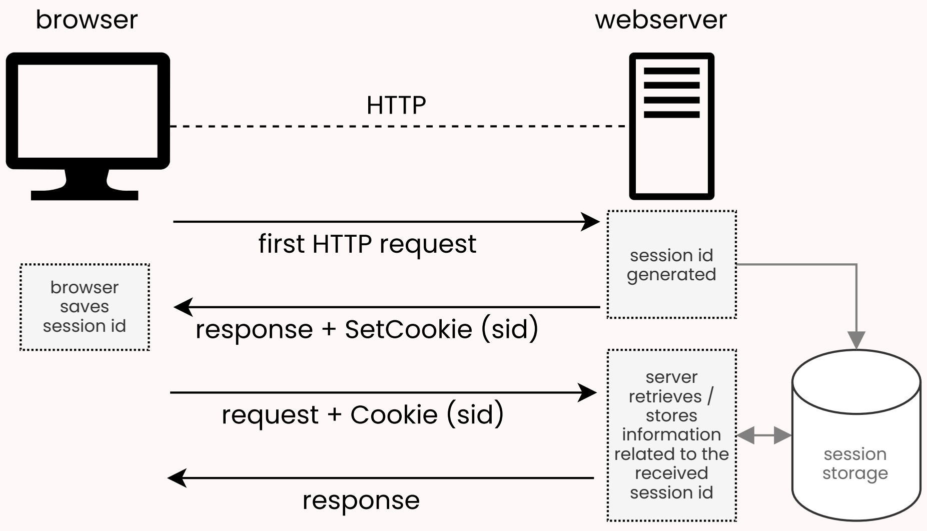
You can access the cookies sent by the browser using the special **$_COOKIE** array.

Sessions

As cookies are stored in the browser, they **cannot be used** as a secure mechanism for storing sensitive information (*e.g.*, the current user).

Sessions are a mechanism that can be used to persist state information between page requests in the server:

- A **unique session identifier** is stored on the **client** (*e.g.*, using a cookie).
- The **server** keeps any **session information** associated with that **session id**.



Sessions in PHP

PHP automatically manages sessions. When a session is started:

1. Retrieve session information:
   - if a *session id* is received (usually from a cookie), **retrieve** any state information for that id.
   - if no *session id* is received, **generate** a new *session id* and **send** it to the client (usually to a cookie).
2. Populate the **$_SESSION** superglobal array with session information associated with the *session id*.
3. When the script ends, serialize the **$_SESSION** contents and store them.

Session Start

Sessions can be started using the **session_start** function:

```
bool session_start (void)
```

- Like other header functions, sessions must be started before any output from your script.
  Because we are using cookies.
- Normally called in every page to ensure session variables are always accessible.

```
session_start();
var_dump($_SESSION);              // inspect ses
$_SESSION['username'] = $username; // modify sess
```

Session Destroy

The function **session_destroy** destroys all of the data associated with the current session.

```
bool session_destroy (void)
```

It must be called after calling **session_start()**.

Session Parameters

The parameters of the cookie used for the session cookie can be changed using the **session_set_cookie_params** function.

```
void session_set_cookie_params
  (int $lifetime, string $path, string $domain,
   bool $secure = false, bool $httponly = false)
```

All parameters are optional except *lifetime*.

- **lifetime** of the session cookie, defined in seconds. The value 0 means "until the browser is closed.
- **path** on the domain where the cookie will work. Use a single slash ('/') for all paths on the domain.
- Cookie **domain**, for example 'www.fe.up.pt'. To make cookies visible on all subdomains, then the domain must be prefixed with a dot, *e.g.*, '.fe.up.pt'.

We will talk about the *secure* and *httponly* parameters when we talk about security.

Storing Passwords

Hash Functions

Password should never be stored in plain text. Instead you should use a one-way hashing function.

```
echo md5('apple');
// 1f3870be274f6c49b3e31a0c6728957f
echo sha1('apple');
// d0be2dc421be4fcd0172e5afceea3970e2f3d940
echo hash('sha256', 'apple');
// 3a7bd3e2360a3d29eea436fcfb7e44c735d117c42d1c18
```

We will talk about better ways of storing passwords when we talk about security.

HTTP Headers

Header

The header function sends a raw HTTP header to the browser.

- This can be used, for example, to redirect the browser to another page:

```php
header('Location: another_page.php');
```

- Headers must be sent before any output from your script.
  This is a protocol restriction.
- Do not forget that this does not stop the execution of the script.
- If you want to stop execution you must follow this instruction with **die()** or **exit()**.

We will talk more about headers when we study the HTTP protocol.

Includes

Includes

- The **include** statement includes and evaluates the specified file.
- The **require** statement is identical to include, except, upon failure, it will also produce a fatal **E_COMPILE_ERROR** level error.
- The **include_once** statement is identical to include, except PHP will check if the file has already been included.
- The **require_once** statement is identical to require, except PHP will check if the file has already been included.

```
include_once('other_file.php');
```

Relative Includes

In PHP, includes are relative to the **file requested** by the browser, not the file that contains the 'include'. This means that:

```
b/Y.php // file requested by the browser
b/Z.php // file included by Y.php

//Y.php only needs to do: include('Z.php')
```

But:

```
a/X.php // file requested by the browser
b/Y.php // file included by X.php
b/Z.php // file included by Y.php

//X.php needs to do: include('../b/Y.php')
//Y.php needs to do: include('../b/Z.php')
```

Magic Constants

To make including files in PHP more manageable, we can use the following magic constants:

```
__FILE__ // The full path and filename of the cur
__DIR__  // The folder of the current file.
```

And the following function that returns the folder of a file:

```
string dirname ( string $path [, int $levels = 1
```

For example:

```
dirname(__FILE__) // same as __DIR__
dirname(__DIR__)  // returns the parent folder of
```

ℹ Magic constants change value depending on where they are used!

JSON

JSON

- JSON (**J**ava**S**cript **O**bject **N**otation) is a *lightweight data-interchange format.* Some alternatives are YAML and TOML.
- It is easy for **humans** to read and write.
- It is easy for **machines** to parse and generate.

```
[
  {
   "id":"1",
   "title":"Mauris...",
   "introduction":"Sed eu...",
   "fulltext":"Donec feugiat..."
  }, {
   "id":"2",
   "title":"Etiam efficitur...",
   "introduction":"Cum sociis ...",
   "fulltext":"Donec feugiat..."
  }
]
```

JSON

The **json_encode** and **json_decode** functions can be used to encode from and to JSON easily.

```
$encoded = json_encode($posts);
$decoded = json_decode($encoded); //$decoded ===
```

Don't forget to tell the client your are sending JSON data:

```
$data = getSomeData();
header('Content-Type: application/json; charset=u
echo json_encode($data);
```

Best Practices

Validate your input

Never trust the user:

```
if (empty($_GET['username']) || length($_GET['u
  // Do something about it
```

Always verify if the data you are receiving is in the expected format.

Separate your PHP and HTML code

Always start by calculating/querying all your data, and only after that output HTML.

```
<?php
  $stmt = $dbh->prepare('SELECT * FROM car WHERE
  $stmt->execute(array($make));

  $cars = $stmt->fetchAll();
?>
<body>
<?php foreach ($cars as $car) { ?>
  <ul>
    <li><strong>Model:</strong> <?=$car['model']?
    <li><strong>Price:</strong> <?=$car['price']?
  </ul>
<?php } ?>
</body>
```

You can use the short *echo* version to make your code look nicer.

Tip: PHP delimiters can break in the middle of a block and pick up later.

Don't Repeat Yourself (DRY)

Use include and/or functions to avoid code repetitions:

```php
// inside database/cars.php
function getAllCars(PDO $dbh) : array {
  $stmt = $dbh->prepare('SELECT * FROM car WHERE
  $stmt->execute(array($make));

  $cars = $stmt->fetchAll();
}
```

```php
include ('database/connection.php');
include ('database/cars.php');
$cars = getAllCars($dbh);
```

Don't Repeat Yourself (DRY)

Use include and/or functions to avoid code repetitions:

```html
<html> <!-- inside templates/header.html -->
  <head>
    <title>My Site</title>
    <meta charset="utf-8">
  </head>
  <body>
```

```html
  </body> <!-- inside templates/footer.html -->
</html>
```

Don't Repeat Yourself (DRY)

Use include and/or functions to avoid code repetitions:

```
<?php
  include ('database/connection.php');
  include ('database/cars.php');
  $cars = getAllCars($dbh);

  include ('templates/header.html');

  foreach ($cars as $car) { ?>

    <ul>
      <li><strong>Model:</strong> <?=$car['model'
      <li><strong>Price:</strong> <?=$car['price'
    </ul>

<? }
  include ('templates/footer.html');
?>
```

Templates

You can also create and reuse **parameterized** functions that output HTML code:

```
<?php function drawCarList(array $cars) { ?>
  <?php foreach ($cars as $car) { ?>
  <ul>
    <li><strong>Model:</strong> <?=$car['model']?
    <li><strong>Price:</strong> <?=$car['price']?
  </ul>
  <?php } ?>
<?php } ?>
```

Templates

And in the end, you will get clean PHP code:

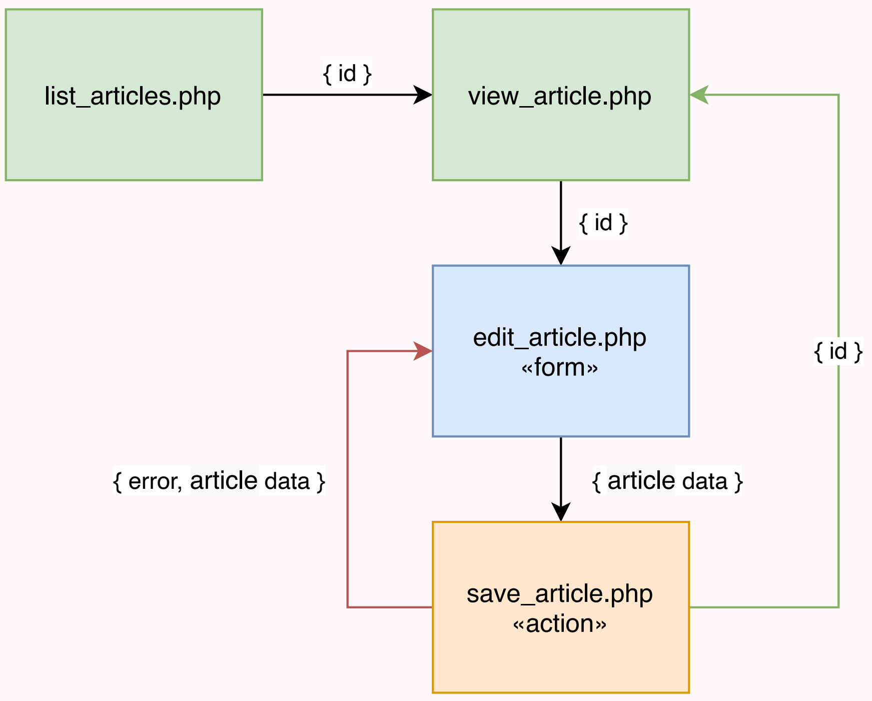
```php
<?php
  include ('database/connection.php');
  include ('database/cars.php');

  include ('templates/common.php');
  include ('templates/cars.php');

  $cars = getAllCars($dbh);

  drawHeader();       // from templates/common.ph
  drawCarList($cars); // from templates/cars.php
  drawFooter();       // from templates/common.ph
?>
```

Separate Actions from Views

Never mix scripts that **return** data with scripts that **change** data:

- **list_articles.php**
  - Shows all news.
  - Has links to each one of the news articles **view_article.php**.
- **view_article.php**
  - Shows one news article and its comments.
  - Receives the id of the article.
  - Link to **edit_article.php**.
- **edit_article.php**
  - Shows a form that allows the user to edit a news article.
  - Submits to **save_article.php**.
- **save_article.php**
  - Receives the new data for the news article.
  - Saves it in the database and redirects to **view_item.php** (on success).

Separate Actions from Views



Extra Stuff

- Functions: Dates, Image Processing
- Charts: jpGraph, pChart, phpChart, Charts 4 PHP
- Standard Library: SPL
- Dependency Manager: Composer
- Template Engines: Twig, Blade, Smarty
- Frameworks: CodeIgniter, CakePHP, Symfony, Zend, Laravel, ...