PassQueen
a.k.a dumpsqueen.com

Zend Certified PHP Engineer

Zend 200-550

Version Demo

Total Demo Questions: 27

Total Premium Questions: 277

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, PHP Basics	24
Topic 2, Functions	25
Topic 3, Data Format & Types	25
Topic 4, Web Features	65
Topic 5, Object Oriented Programming	50
Topic 6, Security	20
Topic 7, Error Handling	3
Topic 8, Strings and Patterns	22
Topic 9, Arrays	26
Topic 10, Databases and SQL	16
Topic 11, Mix Questions	1
Total	277

QUESTION NO: 1

The following form is loaded in a recent browser and submitted, with the second select option selected:

In the server-side PHP code to deal with the form data, what is the value of $_POST['list'] ?

A. 1

B. 2

C. two

D. null

E. It is not set (the `list` field is not submitted).

ANSWER: E

Explanation:

`It is not set (the list field is not submitted)`. is correct because the `<select>` element is explicitly closed before any of the `<option>` elements occur. Consequently, the options are not children of that select control in the parsed document. The named select itself has no option available to be selected, so it does not contribute a name/value pair when the form is submitted. Although the prompt says that the second displayed option is selected, that text does not repair the document structure or place the option within the named select element. Form submission constructs its data set from successful controls; a select contributes its name and the value of a selected option only when it has an appropriate selected option. Since no `list` entry is sent in the request body, PHP does not create `$_POST['list']`. Accessing that missing array key directly can produce an undefined-array-key warning in current PHP versions, so robust code should first test with `isset($_POST['list'])` or use a null-coalescing fallback. The HTML form-data construction rules are specified by the WHATWG HTML Standard; PHP's handling of externally supplied request variables is described in the PHP $_POST documentation.

QUESTION NO: 2

What is the output of the following code? class A { public $a = 1;

public function __construct($a) { $this->a = $a; }

public function mul() { return function($x) {

return $this->a*$x;

};

}

}

$a = new A(2); $a->mul = function($x) {

return $x*$x;

};

$m = $a->mul(); echo $m(3);

A. 9

B. 6

C. 0

D. 3

ANSWER: B

Explanation:

The output is **6**. Constructing the object with `new A(2)` invokes the constructor and sets the public instance property `$a` to `2`. The later assignment to `$a->mul` creates an object property containing a closure, but it does not replace the class's declared `mul()` method. The syntax `$a->mul()` is a method call, so it invokes the declared method and returns the anonymous function defined within that method.

That returned anonymous function uses `$this->a * $x`. Since it was created in a non-static object-method context, the closure has access to the current object through `$this`. Its call as `$m(3)` therefore multiplies the object's stored value, `2`, by `3`, producing `6`. PHP documents that anonymous functions declared in a class context are automatically bound to the object instance when appropriate, allowing use of `$this`. See the PHP manual on anonymous functions and its reference for object methods and member access.

QUESTION NO: 3

Given a DateTime object that is set to the first second of the year 2014, which of the following samples will correctly return a date in the format '2014-01-01 00:00:01'?

A. $datetime->format('%Y-%m-%d %h:%i:%s')

B. $datetime->format('%Y-%m-%d %h:%i:%s', array('year', 'month', 'day', 'hour', 'minute', 'second'))

C. $datetime->format('Y-m-d H:i:s')

D. $date = date('Y-m-d H:i:s', $datetime);

ANSWER: C

Explanation:

`$datetime->format('Y-m-d H:i:s')` correctly formats a PHP `DateTime` object as `2014-01-01 00:00:01`. The `DateTime::format()` method accepts a format string whose characters are PHP date-format tokens. In this string, `Y` produces the four-digit year, `m` the zero-padded numeric month, and `d` the zero-padded day of the month. The space is emitted literally. For the time portion, `H` produces a zero-padded 24-hour value, `i` produces zero-padded minutes, and `s` produces zero-padded seconds. Therefore, a DateTime value representing January 1, 2014 at one second after midnight is rendered exactly in the requested year-month-day and 24-hour time layout. PHP uses these unprefixed format characters for both the procedural `date()` function and the object-oriented `DateTimeInterface::format()` API; percent-prefixed directives belong to `strftime()`-style formatting rather than this method. The PHP manual documents `DateTimeInterface::format()` and its supported date/time format characters at PHP: DateTimeInterface::format and PHP: format character reference.

QUESTION NO: 4

Which of the following statements about property visibility in PHP are correct? (Choose 2)

A. A private property may be accessed only by the class that declares it.

B. A protected property may be accessed by the declaring class and its child classes.

C. A private property may be accessed by every child class.

D. A protected property may be accessed from any script.

E. A public property may be accessed only by methods of its class.

ANSWER: A B

Explanation:

A `private` property may be accessed only from within the class that declares it. A child class does not receive direct access to a private property declared by its parent, even though an instance of the child class contains the inherited object state.

A `protected` property may be accessed from the declaring class and from classes that extend it. Protected visibility is therefore useful where a base class needs to expose implementation state to subclasses without making that state publicly accessible to unrelated calling code. See the PHP documentation on visibility.

QUESTION NO: 5

Which of the following rules must every correct XML document adhere to? (Choose 2)

A. It has to be well-formed.

B. It has to be valid.

C. It has to be associated to a DTD.

D. It may only contain UTF-8 encoded characters.

ANSWER: A B

Explanation:

A correct XML document must be well-formed. Well-formedness is the fundamental XML syntax requirement: the document must have a single document element, properly nested elements, matching start and end tags, quoted attribute values, and correctly escaped reserved markup characters. These rules allow an XML parser to read the document unambiguously.

A correct XML document must also be valid when the document is being evaluated against a declared document model. Validity builds on well-formedness by requiring the XML content and attributes to conform to the constraints defined by the applicable DTD. In practice, validity is used where an application needs a predictable element structure and permitted attribute/content combinations. The XML specification distinguishes the baseline well-formedness requirements from the additional constraints checked for validity, and PHP's DOM extension exposes validation through `DOMDocument::validate()` for documents using a DTD. See the W3C XML 1.0 Recommendation and the PHP DOMDocument::validate documentation.

QUESTION NO: 6

Type hinting in PHP allows the identification of the following variable types: (Choose 2)

A. String

B. Integer

C. Array

D. Any class or interface type

E. All of the above

ANSWER: C D

Explanation:

For the PHP version covered by the Zend 200-550 examination, type hints could be declared for an `array` parameter and for an object that is an instance of a specified class or implements a specified interface. Thus, `function process(array`

`$items)` requires the argument to be an array, while `function save(PDO $connection)` requires an instance of `PDO` or a compatible subclass. An interface can likewise be used as the declared type, allowing any object that implements that interface to satisfy the requirement. These declarations let PHP validate values passed to functions and methods at the call boundary, making APIs clearer and helping detect invalid arguments early. PHP's historical type-hinting documentation specifically describes class/interface names and `array` as supported hints; scalar declarations such as `string` and `int` were introduced later and are not part of the type-hinting set tested by this exam version. See the PHP manual on type hinting and the PHP manual on type declarations.

QUESTION NO: 7

When a query that is supposed to affect rows is executed as part of a transaction, and reports no affected rows, it could mean that: (Choose 2)

A. The transaction failed

B. The transaction affected no lines

C. The transaction was rolled back

D. The transaction was committed without error

ANSWER: A B

Explanation:

The transaction failed and **The transaction affected no lines** are the applicable conclusions. An affected-row result describes the outcome of the data-modification statement that was executed, rather than guaranteeing that the transaction has produced durable changes. A failure while executing the statement or processing the transactional work can leave no changes applied by that work; applications should therefore inspect the database API's error status or exception in addition to examining the affected-row value.

A zero affected-row count can also be a legitimate, successful result: the statement may have found no rows matching its condition, or an update may have produced no changed rows according to the driver/database semantics. In that situation, the transaction can remain active, but this particular query has not modified any rows. Code that requires a row to exist or a state transition to occur should treat an unexpected zero count as a business-condition check and decide whether to retry, report a conflict, or roll back.

PHP's database APIs document that affected-row information must be interpreted together with execution and error results; see the mysqli affected_rows documentation and the PDOStatement::rowCount documentation.

QUESTION NO: 8

Which of these databases is NOT supported by a PDO driver?

A. Microsoft SQL Server

B. SQLite

C. Microsoft Access

D. Berkeley DB

ANSWER: D

Explanation:

Berkeley DB is the correct answer because PHP's PDO extension does not include an official PDO driver for Berkeley DB. PDO is a database-access abstraction layer, but its usable connections depend on a driver implemented for a particular database system. The PHP documentation's PDO driver list identifies the supported drivers, such as drivers for MySQL, PostgreSQL, SQLite, Oracle, ODBC, IBM databases, Firebird, Informix, and SQL Server variants; Berkeley DB is not

included in that driver set. Berkeley DB is also primarily an embedded key-value storage library rather than a conventional SQL relational database server, so it does not fit the normal PDO driver model used for SQL statements, prepared statements, and result sets. Therefore, an application cannot create a standard PDO Berkeley DB connection using a built-in or officially documented PDO Berkeley DB driver. The available PDO drivers can be checked through `PDO::getAvailableDrivers()` in an installed PHP environment, which returns only drivers compiled into that PHP build. See the PHP PDO drivers documentation and the PDO::getAvailableDrivers() documentation.

QUESTION NO: 9

Which of the following is NOT true about PHP traits? (Choose 2)

A. Multiple traits can be used by a single class.

B. A trait can implement an interface.

C. A trait can declare a private variable.

D. Traits are able to be auto-loaded.

E. Traits automatically resolve conflicts based on definition order.

ANSWER: E

Explanation:

Traits automatically resolve conflicts based on definition order is not true. When a class imports multiple traits that define a method with the same name, PHP does not choose one merely because that trait appears first or last in the `use` statement. Instead, PHP reports a method-collision error unless the class explicitly states how the conflict is to be resolved. The class uses the `insteadof` operator to select the method from one trait in preference to another, and may use `as` to assign an alias to a trait method. This explicit syntax makes the selected implementation clear and prevents a change in trait declaration order from silently changing application behavior. PHP's trait documentation describes conflict-resolution rules and provides examples using `insteadof` and `as`: PHP Manual: Traits. The formal language specification likewise defines trait method conflicts as requiring an explicit resolution rather than an order-based automatic decision: PHP Manual: Conflict Resolution.

QUESTION NO: 10

Given the following DateTime objects, what can you use to compare the two dates and indicate that $date2 is the later of the two dates?

$date1 = new DateTime('2014-02-03'); $date2 = new DateTime('2014-03-02');

A. $date2 > $date1

B. $date2 < $date1

C. $date1->diff($date2) < 0

D. $date1->diff($date2) > 0

ANSWER: A

Explanation:

`$date2 > $date1` is the appropriate comparison because PHP supports comparison operators for `DateTime` objects. When two date/time objects are compared, PHP evaluates their represented instants chronologically. Here, `$date1` represents February 3, 2014, while `$date2` represents March 2, 2014. Since March 2 occurs after February 3 in the same year, the greater-than expression evaluates to `true`, directly indicating that `$date2` is later.

This is concise and suitable when the required result is simply a boolean chronological comparison. The objects were created from ISO-style date strings containing no explicit time, so each represents the beginning of its specified calendar date in the applicable default time zone. PHP's Date and Time documentation describes the supported comparison behavior for date/time objects, including chronological object comparisons. See the PHP DateTime comparison documentation and the PHP DateTime class reference.

QUESTION NO: 11

Which one of the following XML declarations is NOT valid?

A. <?xml version="1.0" ?>
B. <?xml version="1.1" encoding="UTF-8" ?>
C. <?xml standalone="no" ?>
D. <?xml standalone="1" ?>

A. Option A

B. Option B

C. Option C

D.

ANSWER: D

Explanation:

`<?xml version="1.0" standalone="yes" encoding="UTF-8"?>` is not a valid XML declaration because XML requires declaration pseudo-attributes to appear in a defined order. The `version` pseudo-attribute, when present, must be first. It may be followed by `encoding`, and `standalone`, if supplied, must occur after the encoding declaration. Consequently, an XML processor cannot accept a declaration that places `standalone` before `encoding`. This is a syntactic requirement of XML, rather than a matter of application preference or character-set compatibility. Correctly ordered declarations allow parsers to establish the XML version and character encoding before processing the remainder of the document, while the standalone value specifies whether the document relies on external markup declarations. The XML specification defines the declaration grammar as `version`, followed optionally by `encoding`, followed optionally by `standalone`. See the W3C's XML declaration grammar and its discussion of the standalone document declaration.

QUESTION NO: 12

What is the output of the following code?

echo "1" + 2 * "0x02";

A. 1

B. 3

C. 5

D. 20

E. 7

ANSWER: C

Explanation:

The output is `5`. This question targets the PHP behavior applicable to the Zend 200-550 era, when a hexadecimal-formatted string such as `"0x02"` was converted to its hexadecimal numeric value, `2`, when used in an arithmetic expression. PHP

evaluates multiplication before addition, so it first evaluates `2 * "0x02"` as `2 * 2`, producing `4`. It then evaluates `"1" + 4`. Because the addition operator requires numeric operands, the string `"1"` is converted to the integer `1`, and the final result is therefore `5`. Finally, `echo` writes that numeric result directly to the output. This behavior is version-sensitive: PHP later changed the handling of hexadecimal strings in numeric contexts, so modern PHP versions should not be assumed to produce the same result. The PHP migration documentation records that hexadecimal strings stopped being considered numeric in PHP 7.0, confirming the older behavior relevant to this certification question. See the PHP 7.0 backward-incompatible changes documentation and the PHP operator-precedence documentation.

QUESTION NO: 13

Which of the following functions will allow identifying unique values inside an array?

A. array_unique_values

B. array_distinct

C. array_count_values

D. array_intersect

E. array_values

ANSWER: C

Explanation:

`array_count_values` is correct because it examines an array's values and returns an associative array whose keys are the encountered values and whose values are the number of times each value occurs. This makes it possible to identify values that occur uniquely: after calling the function, entries with a count of `1` represent values present only once in the original array. For example, applying `array_count_values()` to `['red', 'blue', 'red', 'green']` produces counts equivalent to `['red' => 2, 'blue' => 1, 'green' => 1]`. The values `blue` and `green` can therefore be identified as unique by checking for a frequency of one.

This function is especially useful where "unique" means values that have no duplicate occurrence, rather than merely obtaining a de-duplicated list. The function accepts an array of integers and strings, records each value's frequency, and supplies the information needed to filter or otherwise process uniquely occurring values. PHP's official function reference documents that it counts all values in an array and returns an array using the values as keys and their frequencies as values: PHP: array_count_values. The broader PHP array-function reference is available at PHP Array Functions.

QUESTION NO: 14

Which options do you have in PHP to set the expiry date of a session?

A. Set the session.duration directive in php.ini

B. Set session cookie expiry date locally via session_set_cookie_params()

C. Set session expiry date locally via session_cache_expire()

D. None of the above

ANSWER: B

Explanation:

`Set session cookie expiry date locally via session_set_cookie_params()` is correct because PHP sessions commonly identify the browser session through a cookie, and `session_set_cookie_params()` configures the parameters used when PHP sends that session cookie. Its `lifetime` parameter specifies how many seconds the cookie remains valid; a positive value causes PHP to issue a persistent cookie with an expiration time, while a lifetime of `0` produces a cookie that expires when the browser session ends. The function must be called before `session_start()`, because

session startup sends or resumes the cookie and its associated HTTP headers. For example, calling `session_set_cookie_params(3600)` before starting the session requests a cookie lifetime of one hour. This is the appropriate local, per-session configuration mechanism when the requirement is to control the expiry date presented to the client browser. PHP documents the lifetime parameter as the number of seconds the cookie is valid and notes that these parameters are used by `session_start()`. See the PHP manual for session_set_cookie_params() and the session.cookie_lifetime configuration documentation.

QUESTION NO: 15

Which of the following items in the $_SERVER superglobal are important for authenticating the client when using HTTP Basic authentication? (Choose 2)

A. PHP_AUTH_TYPE

B. PHP_AUTH_PASSWORD

C. PHP_AUTH_DIGEST

D. PHP_AUTH_PW

E. PHP_AUTH_USER

ANSWER: D E

Explanation:

For HTTP Basic authentication handled by PHP, the credentials supplied by the client are exposed through the `$_SERVER` superglobal as `PHP_AUTH_USER` and `PHP_AUTH_PW`. `PHP_AUTH_USER` contains the username submitted in the HTTP Authorization header, while `PHP_AUTH_PW` contains the corresponding password. An application uses these two values together to identify the claimed user and validate the supplied secret against its own trusted identity store, such as a database or directory service. PHP makes these variables available when it is running as an Apache module and HTTP Basic authentication is in use; application code commonly checks whether the username is present, then compares the submitted credentials using secure password-verification practices. These are therefore the key request values needed to authenticate a Basic-auth client. PHP also documents a separate server variable for HTTP Digest authentication, but Basic authentication specifically relies on the user and password variables. See the PHP manual's documentation for HTTP authentication with PHP and its reference for $_SERVER reserved variables.

QUESTION NO: 16 - (SIMULATION)

Which PHP function is used to validate whether the contents of $_FILES['name']['tmp_name'] have really been uploaded via HTTP?

ANSWER: See the explanation for the answer

Explanation:

Use `is_uploaded_file()`.

```
if (is_uploaded_file($_FILES['name']['tmp_name'])) {
    // The temporary file was uploaded through HTTP POST
}
```

This function verifies that the specified file was genuinely uploaded through PHP's HTTP POST upload mechanism.

QUESTION NO: 17

Transactions should be used to: (Choose 2)

A. Prevent errors in case of a power outage or a failure in the SQL connection

B. Ensure that the data is properly formatted

C. Ensure that either all statements are performed properly, or that none of them are.

D. Recover from user errors

ANSWER: A C

Explanation:

Transactions should be used to prevent errors in case of a power outage or a failure in the SQL connection, provided that the database engine supports transactions and the changes have not yet been committed. A transaction groups related database operations into a single unit of work. If an interruption occurs before a successful commit, the database can roll back the uncommitted work, preventing partially completed changes from being left visible. This behavior is central to maintaining consistent data when an application loses its database connection or a system failure interrupts processing.

Transactions should also be used to ensure that either all statements are performed properly, or that none of them are. This is the atomicity property: a group of dependent changes is committed together only after the full operation succeeds. For example, when transferring funds, debiting one account and crediting another should be treated as one operation; if either statement cannot complete, a rollback preserves the prior consistent state. In PHP, PDO exposes transaction control through `beginTransaction()`, `commit()`, and `rollBack()`. See the PHP PDO Transactions documentation and the MySQL COMMIT documentation.

QUESTION NO: 18

What does the __FILE__ constant contain?

A. The filename of the current script.

B. The full path to the current script.

C. The URL of the request made.

D. The path to the main script.

ANSWER: B

Explanation:

`__FILE__` contains the full filesystem path and filename of the PHP file in which that constant occurs. It is a PHP magic constant: PHP resolves its value while parsing the source file, rather than deriving it from the incoming HTTP request. Thus, when the code is in a file such as `/var/www/app/includes/config.php`, `__FILE__` evaluates to that complete path (subject to PHP's path resolution behavior). This makes it especially useful for building reliable paths relative to the current source file, for example with `dirname(__FILE__)` or the related `__DIR__` magic constant. If a file is included by another PHP script, the value still identifies the included file containing the expression, not merely the entry script handling the request. PHP documents `__FILE__` as "the full path and filename of the file," and notes that resolved symbolic links are reflected in the value. See the official PHP documentation on magic constants and dirname().

QUESTION NO: 19

What is the output of the following code?

```
$a = array('a' => 1, 'b' => 2);
$b = array('b' => 3, 'c' => 4);
print_r($a + $b);
```

A. Array ([a] => 1 [b] => 2 [c] => 4)

B. Array ([a] => 1 [b] => 3 [c] => 4)

C. Array ([a] => 1 [b] => 5 [c] => 4)

D. PHP Fatal error

ANSWER: A

Explanation:

The result is an array equivalent to `Array ( [a] => 1 [b] => 2 [c] => 4 )`. The `+` operator performs an array union when both operands are arrays. It adds entries from the right-hand array only when their keys do not already exist in the left-hand array.

The key `b` already exists in `$a`, so its value of `2` is retained and the value `3` from `$b` is ignored. The key `c` does not exist in `$a`, so the entry `'c' => 4` is added. PHP documents this behavior in its reference for array operators.

QUESTION NO: 20

Which function can NOT help prevent cross-site scripting? (Choose 2)

A. addslashes()

B. htmlentities()

C. htmlspecialchars()

D. strip_tags()

E. quotemeta()

ANSWER: A E

Explanation:

`addslashes()` and `quotemeta()` cannot be relied on to prevent cross-site scripting because neither function performs HTML output encoding. Cross-site scripting defenses must ensure that data is encoded for the specific output context before it is inserted into an HTML document, such as element content or an attribute value. PHP documents `addslashes()` as adding backslashes before single quotes, double quotes, backslashes, and NUL bytes. That transformation is not equivalent to converting HTML-significant characters such as `<`, `>`, and `&` into safe entities, so browser-parsed markup can remain dangerous. `quotemeta()` is intended to quote regular-expression metacharacters, including characters such as periods, brackets, parentheses, and asterisks. Its purpose is regex construction rather than safe HTML rendering, and it likewise does not provide context-appropriate HTML escaping. In PHP applications, output encoding with functions designed for HTML contexts, especially `htmlspecialchars()` with suitable flags and character encoding, is the appropriate baseline control for untrusted text rendered into HTML. See the PHP references for addslashes() and quotemeta().

QUESTION NO: 21

An unbuffered database query will: (Choose 2)

A. Return the first data faster

B. Return all data faster

C. Free connection faster for others scripts to use

D. Use less memory

ANSWER: A D

Explanation:

An unbuffered database query returns rows to PHP as they are received from the database server, rather than first retrieving and storing the complete result set in the client-side PHP process. Consequently, it can **return the first data faster**: application code may begin fetching and processing an initial row without waiting for every row in a potentially large result set to be transferred. This is particularly useful for streaming exports, reports, or other sequential processing of large query results.

It will also **use less memory** in PHP because the entire result set is not retained in a client-side buffer. Memory consumption is therefore largely limited to the current row and the application's own processing requirements, which helps avoid exhausting memory when a query produces many rows. PHP documents this behavior for MySQLi's unbuffered result mode, where rows are fetched individually from the server; PDO MySQL similarly provides an unbuffered-query setting. See the PHP MySQLi query documentation and the PHP documentation on buffered and unbuffered queries.

QUESTION NO: 22

When would you use classes and when would you use namespaces?

A. Use classes to encapsulate code and represent objects, and namespaces to avoid symbol name collisions

B. Use classes for performance-sensitive code, and namespaces when readability matters more

C. Use namespaces for performance-sensitive code, and classes when readability matters more

D. Always use them; namespaces are always superior to classes

ANSWER: A

Explanation:

Use classes to encapsulate code and represent objects, and namespaces to avoid symbol name collisions. In PHP, a class provides a blueprint for objects and can group related state and behavior through properties, methods, constants, visibility, inheritance, and interfaces. Classes are therefore appropriate when code models an entity or service and benefits from object-oriented encapsulation. A namespace is instead a mechanism for organizing identifiers: it qualifies the names of classes, interfaces, traits, functions, constants, and enumerations. Its core purpose is to prevent conflicts when distinct libraries or parts of an application define symbols with the same short name. For example, two packages can each expose a class named `Logger` when those classes reside in distinct namespaces. Namespaces and classes solve complementary problems rather than being alternatives; a class can be declared within a namespace, such as `namespace App\\Service;` followed by `class Logger`. PHP's manual describes namespaces as a way of encapsulating items and preventing name collisions, while its object-oriented programming documentation covers classes as the basis for defining objects and their behavior. See the PHP namespaces documentation and PHP classes and objects documentation.

QUESTION NO: 23

Which of the following PHP session configuration options help protect a session cookie from unnecessary exposure? (Choose 2)

A. session.cookie_httponly

B. session.cookie_secure

C. session.use_trans_sid

D. session.cache_limiter

E. session.gc_divisor

ANSWER: A B

Explanation:

`session.cookie_httponly` instructs PHP to mark the session cookie with the HttpOnly attribute. Browsers that support this attribute do not expose the cookie to client-side scripts such as JavaScript, helping reduce the risk that script injection can directly read a session identifier.

`session.cookie_secure` instructs PHP to mark the session cookie with the Secure attribute. A Secure cookie is sent only over HTTPS connections, protecting the session identifier from being transmitted over an unencrypted HTTP request. This setting should be enabled for applications served over HTTPS. See the PHP documentation for session.cookie_httponly and session.cookie_secure.

QUESTION NO: 24 - (SIMULATION)

Which SPL class implements fixed-size storage?

ANSWER: See the explanation for the answer

Explanation:

The SPL class that implements fixed-size storage is `SplFixedArray`.

Example:

```
$array = new SplFixedArray(10); // exactly 10 slots
$array[0] = 'value';
```

Unlike a normal PHP array, its size is explicitly set and can only be changed through `setSize()`.

QUESTION NO: 25

Your application uses PHP to accept and process file uploads. It fails to upload a file that is 5 MB in size, although upload_max_filesize is set to "10M". Which of the following configurations could be responsible for this outcome? (Choose 2)

A. The PHP configuration option post_max_size is set to a value that is too small

B. The web server is using an incorrect encoding as part of the HTTP response sent to the client

C. The browser uses an incorrect encoding as part of the HTTP request sent to the server

D. The hidden form field MAX_FILE_SIZE was set to a value that is too small

E. PHP cannot process file uploads larger than 4 MB

ANSWER: A D

Explanation:

`post_max_size` is a separate PHP limit that can prevent an upload even when `upload_max_filesize` permits the individual file. A multipart/form-data request contains the file plus multipart boundaries, the names and values of other form controls, and request headers. Consequently, `post_max_size` must be larger than the intended uploaded file size; PHP documents that it must generally be larger than `upload_max_filesize`. If the complete request exceeds `post_max_size`, PHP does not populate the normal POST and uploaded-file data, so application upload handling can observe a failed or missing upload. See PHP: post_max_size.

The hidden `MAX_FILE_SIZE` form field can also cause this result when its value is below 5 MB. When this field appears before the file input, PHP uses it as the maximum accepted file size for that form submission. A file exceeding that declared value produces the `UPLOAD_ERR_FORM_SIZE` upload error, regardless of the higher server-side `upload_max_filesize` setting. PHP emphasizes that this field is not a security control because it is supplied by the client, but it remains an upload constraint that can affect behavior. See PHP: File Upload Common Pitfalls.

QUESTION NO: 26

Which of the following statements about anonymous functions in PHP are NOT true? (Choose 2)

A. Anonymous functions can be bound to objects

B. Anonymous functions created within object context are always bound to that object

C. Assigning closure to a property of an object binds it to that object

D. Methods bind() and bindTo() of the Closure object provide means to create closures with different binding and scope

E. Binding defines the value of $this and the scope for a closure

ANSWER: B C

Explanation:

Anonymous functions created within object context are always bound to that object is not universally true. PHP supports `static` anonymous functions, and declaring a closure as static prevents automatic binding of `$this`. Consequently, the object context alone does not guarantee that every anonymous function created there has an object binding. The relevant PHP behavior is documented in the PHP manual's anonymous-functions documentation, which notes that static anonymous functions cannot have `$this` bound automatically.

Assigning closure to a property of an object binds it to that object is also not true. Storing a Closure instance in an object property is ordinary assignment: the property holds the same closure object, and that action does not establish or change its bound object or class scope. A closure's binding must instead be established at creation where applicable, or deliberately changed through Closure binding facilities. PHP documents this distinction in its Closure::bindTo() reference: creating a closure with a new bound object and scope is an explicit operation, separate from assigning the closure to a property.

QUESTION NO: 27

When retrieving data from URLs, what are valid ways to make sure all file_get_contents calls send a certain user agent string? (Choose 2)

A. $default_opts = array('http'=>array('user_agent'=>"My Cool Browser"));
$default = stream_context_set_default($default_opts);

B. stream_context_set_option("user_agent", "My Cool Browser");

C. ini_set('user_agent', "My Cool Browser");

D. stream_context_set_option($context, "http", "user_agent", "My Cool Browser");

ANSWER: A C

Explanation:

Both `stream_context_set_default()` with an HTTP-context `user_agent` setting and `ini_set('user_agent', ...)` establish defaults used by PHP's HTTP stream wrapper. A default stream context is applied to stream operations that do not explicitly supply their own context, so configuring the `http` wrapper's `user_agent` option makes ordinary URL-based `file_get_contents()` calls send the selected value. PHP documents that the default context is used whenever no stream context is supplied to a stream-creation function. See stream_context_set_default().

The `user_agent` INI directive is specifically the default User-Agent header used by PHP's HTTP wrapper. Since it is runtime configurable, `ini_set('user_agent', "My Cool Browser")` changes that wrapper default for the remainder of the current script execution. Consequently, URL retrievals performed through `file_get_contents()`, when they use the HTTP wrapper and do not override the header through a supplied context, inherit this configured agent string. The directive and its HTTP-wrapper behavior are documented in the PHP filesystem and streams configuration reference.