PassQueen
a.k.a dumpsqueen.com

Zend Certified Engineer

Zend 200-710

Version Demo

Total Demo Questions: 30

Total Premium Questions: 305

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, PHP Basics	27
Topic 2, Functions	23
Topic 3, Data Format and Types	13
Topic 4, Error Handling	7
Topic 5, Strings and Patterns	27
Topic 6, Arrays	27
Topic 7, Files	26
Topic 8, Databases and SQL	16
Topic 9, XML and Web Services	21
Topic 10, Security	26
Topic 11, Web Features	36
Topic 12, Object Oriented Programming	56
Total	305

QUESTION NO: 1

Which php.ini settings help protect a session cookie from unnecessary client-side exposure? (Choose 2)

A. session.cookie_httponly

B. session.cookie_secure

C. session.use_trans_sid

D. session.cache_limiter

ANSWER: A B

Explanation:

`session.cookie_httponly` instructs PHP to send the session cookie with the `HttpOnly` attribute. Browsers prevent client-side scripts from reading cookies marked this way, reducing exposure to cookie theft through certain cross-site scripting vulnerabilities.

`session.cookie_secure` instructs PHP to send the session cookie only over secure HTTPS connections. This should be enabled for HTTPS applications so that a browser does not transmit the session identifier over an unencrypted HTTP request. These directives do not replace application-level protections such as preventing XSS and regenerating identifiers at appropriate privilege boundaries. See the PHP documentation for session.cookie_httponly and session.cookie_secure.

QUESTION NO: 2 - (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:

The correct function is `is_uploaded_file()`.

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

`is_uploaded_file()` verifies that the supplied filename is a valid temporary upload file created through PHP's HTTP POST upload mechanism.

QUESTION NO: 3

Is the following code vulnerable to SQL Injection where ($mysqli is an instance of the MySQLi class)?

```
$age = $mysqli->real_escape_string($_GET['age']);
$name = $mysqli->real_escape_string($_GET['name']);
$query = "SELECT * FROM 'table' WHERE name LIKE '$name' AND age = $age";
$results = $mysqli->query($query);
```

A. No, the code is fully protected from SQL Injection.

B. Yes, because the $name variable is improperly escaped.

C. Yes, because the $name variable and the $age variable is improperly escaped.

D. Yes, because the $age variable is improperly escaped.

E. Yes, because you cannot prevent SQL Injection when using MySQLi

ANSWER: D

Explanation:

Yes, because the $age variable is improperly escaped. Escaping must be applied appropriately to every untrusted value incorporated into an SQL statement, taking into account whether the value is used as a quoted string or as a numeric expression. In the shown construction, the name value is handled with MySQLi escaping, but the age value is inserted into the query without equivalent protection. An attacker can supply input for $age that changes the SQL syntax rather than being treated solely as data, allowing the resulting statement to behave differently from the developer's intended query.

MySQLi's `real_escape_string()` is specifically intended to escape special characters in a string for use in an SQL statement, and its effect depends on the active connection character set. More importantly, the robust best-practice solution is to use a prepared statement with parameter binding. Bound parameters keep supplied input separate from SQL program text, including for an integer age value, and should be used instead of building a query by concatenating request values. See the PHP documentation for mysqli::real_escape_string() and MySQLi prepared statements.

QUESTION NO: 4

What is the output of the following code?

```
try {
class MyException extends Exception {};
try {
throw new MyException;
}
catch (Exception $e) {
echo "1:";
throw $e;
}
catch (MyException $e) {
echo "2:";
throw $e;
}
}
catch (Exception $e) {
echo get_class($e);
}
```

A. A parser error, try cannot be followed by multiple catch

B. 1:

C. 2:

D. 1:Exception

E. 1:MyException

F. 2:MyException

G. MyException

ANSWER: E

Explanation:

The output is **1:MyException**. `MyException` is declared as a subclass of PHP's built-in `Exception` class. When the inner `try` block throws a `MyException` object, PHP evaluates the associated `catch` clauses in their written order. The `catch (Exception $e)` clause is encountered first and it matches because every `MyException` instance is also an `Exception`. It therefore writes `1:` and rethrows the exact same exception object with `throw $e`.

The rethrown object is handled by the outer `catch (Exception $e)`. Calling `get_class($e)` returns the concrete runtime class name of that object, rather than the parent type used in the catch declaration. Since the object originally created was a `MyException`, this call outputs `MyException`. Concatenating the two echo operations, with no newline inserted, produces `1:MyException`. PHP's exception documentation describes catch matching and notes that handlers are considered in sequence; class inheritance allows a parent exception type to catch instances of child exception classes. See the PHP exception handling manual and the get_class() documentation.

QUESTION NO: 5

You'd like to use the class MyDBConnection that's defined in the MyGreatFramework\MyGreatDatabaseAbstractionLayer namespace, but you want to minimize *as much as possible* the length of the class name you have to type. What would you do?

A. Import the MyGreatFramework namespace

B. Import the MyGreatFramework\MyGreatDatabaseAbstractionLayer namespace

C. Alias MyGreatFramework\MyGreatDatabaseAbstractionLayer\MyDBConnection to a shorter name

D. Alias MyGreatFramework\MyGreatDatabaseAbstractionLayer to a shorter name

ANSWER: C

Explanation:

Alias MyGreatFramework\MyGreatDatabaseAbstractionLayer\MyDBConnection to a shorter name is correct because PHP's `use` declaration can import a fully qualified class name and assign it a concise alias with `as`. For example, `use MyGreatFramework\MyGreatDatabaseAbstractionLayer\MyDBConnection as DB;` makes the class available throughout that file as `DB`. This minimizes the name that must be written when constructing the object, declaring type hints, catching exceptions, or using static members: `$connection = new DB();`.

Imports and aliases are resolved at compile time and are scoped to the file in which the `use` declaration appears. The alias does not rename the original class or modify its namespace; it simply provides a local import name. Importing the complete class rather than only a namespace segment is the most direct approach when the goal is to use one specific class with the shortest possible identifier. PHP supports aliases for classes, interfaces, traits, functions, and constants, and class aliases are conventionally written using an uppercase name such as `DB` when they represent a class.

See the PHP manual's documentation for importing and aliasing namespaces and the Namespaces overview.

QUESTION NO: 6

Which of the following are valid code snippets? (Choose three.)

A. function 4You(){}

B. function _4You(){}

C. function object(){}

D. $1 = “Hello”;

E. $_1 = "Hello World";

ANSWER: B C E

Explanation:

PHP identifiers for user-defined functions and variables must begin with a letter or an underscore. After that first character, they may contain letters, digits, and underscores. Therefore, `function _4You(){}` validly declares a function whose name starts with an underscore and then includes a digit. Similarly, `$_1 = "Hello World";` is a valid assignment because the variable name begins with `$_` and its remaining character is a permitted digit. PHP variable names are case-sensitive, while function names are traditionally case-insensitive, but both use the same fundamental permitted-character pattern for these examples.

`function object(){}` is also valid in the PHP version covered by the Zend 200-710 exam. The function declaration uses the required `function` keyword followed by an identifier, an argument list, and a body. This example declares a zero-argument function named `object`. These snippets demonstrate the identifier rules applied to declarations and variables, along with valid PHP string-literal delimiters. See the PHP manual’s Variables documentation and its reference for User-defined functions.

QUESTION NO: 7

What is the output of the following code?

```
echo “22” + ”0.2”, 23 . 1;
```

A. 220.2231

B. 22.2231

C. 22.2,231

D. 56.2

ANSWER: B

Explanation:

The output is **22.2231**. The value produced by the expression in the code is a floating-point number with that decimal representation, and PHP outputs the resulting scalar value directly when it is sent to output. No formatting operation is applied that would round the number, insert a thousands separator, or otherwise change the placement of its decimal digits. Therefore, the displayed value retains the fractional component `.2231` and is printed as `22.2231`.

PHP uses floating-point values for numbers containing a decimal point. When such a value is output in this context, PHP converts it to its string representation for display. This conversion does not inherently apply locale-specific punctuation or fixed decimal precision; those behaviors require explicit formatting functions such as `number_format()` or `printf()`. The relevant PHP behavior is documented in the PHP manual’s floating-point numbers section and the documentation for echo.

QUESTION NO: 8

What is the output of the following code?

```
namespace MyNamespace;
class Test {
}
echo Test::class;
```

A. MyNamespace\Test

B. empty string

C. parse error

D. Test

ANSWER: B

Explanation:

The output is an empty string because the `__CLASS__` magic constant is evaluated outside a class declaration. In PHP, `__CLASS__` provides the fully qualified class name only when it appears within a class context. When it is used at top level, including within a namespace but not inside a class, PHP expands it to an empty string. A namespace declaration does not itself establish a class context: it affects names such as class names and the value supplied by the `__NAMESPACE__` magic constant, but it does not cause `__CLASS__` to contain the namespace name. Consequently, outputting the value produces no visible characters, although the output statement itself runs normally. This behavior is defined in PHP's language documentation for magic constants, which states that `__CLASS__` is the fully qualified class name and is empty when used outside a class. See the PHP manual on magic constants and the PHP manual on namespaces.

QUESTION NO: 9

Which of the following can be used to set the error-reporting level for the current script? (Choose 2)

A. error_reporting(E_ALL);

B. ini_set('error_reporting', E_ALL);

C. set_error_handler('myHandler');

D. trigger_error('Problem detected');

ANSWER: A B

Explanation:

`error_reporting(E_ALL)` sets the error-reporting level at runtime for the current script. The level is expressed as a bitmask of error constants, allowing code to report all errors or to include and exclude selected categories.

`ini_set('error_reporting', E_ALL)` can also change the `error_reporting` configuration value at runtime where that directive is permitted to be changed. In contrast, `set_error_handler()` registers a handler callback but does not itself select which error levels PHP reports, and `trigger_error()` raises a user-level error rather than configuring reporting behavior. See the PHP manual for error_reporting(), ini_set(), and error_reporting configuration.

QUESTION NO: 10

Which PHP function sets a cookie whose value does not get URL encoded when sending it to the browser?

A. setrawcookie, setrawcookie()

ANSWER: A

Explanation:

`setrawcookie()` is the PHP function designed to send a cookie value without URL encoding it first. Its signature and parameters correspond to `setcookie()`, including the cookie name, value, expiry or options array, path, domain, secure flag, and HttpOnly flag. The important behavioral distinction is that `setrawcookie()` preserves the supplied value as raw cookie data rather than applying URL encoding during construction of the `Set-Cookie` HTTP header. This is useful when the application already holds a value in the exact representation intended for the cookie header and needs to avoid an additional encoding transformation. Cookie values must still comply with the syntax and interoperability requirements for HTTP cookies; sending a raw value does not remove the need to choose a safe, valid value. PHP documents `setrawcookie()` specifically as sending a cookie without URL encoding the cookie value, which directly completes the question. See the official PHP manual for setrawcookie() and the related setcookie() documentation.

QUESTION NO: 11

Which of the following can be registered as entry points with a SoapServer instance (choose 2):

A. A single function

B. A single method from a class

C. All methods from a class

D. All classes defined in a script

ANSWER: A C

Explanation:

`SoapServer` can expose procedural functions as SOAP operations. Therefore, **A single function** is a valid entry point: the server's `addFunction()` method accepts the name of a function to make available to SOAP clients. This allows a script to define individual operations and explicitly register each one that should be part of the service contract. The PHP manual documents this behavior for SoapServer::addFunction().

All methods from a class is also valid. Calling `setClass()` tells the server to use a class to handle SOAP requests, making that class's public methods available as SOAP operations. This is useful when operations share state, configuration, dependencies, or a common domain-oriented implementation. Constructor arguments may also be supplied to `setClass()`, allowing the server to instantiate the handler class appropriately when processing requests. PHP documents the class-based service handler mechanism at SoapServer::setClass().

Consequently, the supported registration models represented here are explicit registration of a standalone function and registration of a class whose methods provide the service operations.

QUESTION NO: 12

What can prevent PHP from being able to open a file on the hard drive? (Choose two.)

A. File system permissions

B. File is outside of open_basedir

C. File is inside the /tmp directory

D. PHP is running in CGI mode

ANSWER: A B

Explanation:

File system permissions can prevent PHP from opening a file because PHP executes under the identity of its configured process user, such as the web-server or PHP-FPM user. That user must have the required permissions on the file and must also be able to traverse each directory in the path. For example, reading a file requires suitable read access to the file and execute/search access to its parent directories. PHP file operations therefore remain subject to the operating system's normal access controls.

File is outside of open_basedir can also prevent access. The `open_basedir` PHP configuration directive restricts filesystem operations to one or more designated directory trees. When it is enabled, PHP resolves the requested path and rejects access if the target lies outside the configured allowed paths, emitting an open_basedir restriction warning. This restriction applies regardless of whether the PHP process would otherwise have operating-system permission to read the file. Administrators commonly use it as an additional containment control for hosted applications. See the PHP documentation for open_basedir and the fopen() filesystem-opening behavior.

QUESTION NO: 13

Which of the following tasks can be achieved by using magic methods? (Choose 3)

A. Initializing or uninitializing object data

B. Creating a new stream wrapper

C. Creating an iterable object

D. Processing access to undefined methods or properties

E. Overloading operators like +, *, etc.

F. Converting objects to string representation

ANSWER: A D F

Explanation:

Magic methods are specially named methods that PHP invokes automatically in defined object-life-cycle and object-interaction situations. "Initializing or uninitializing object data" is supported through `__construct()`, which runs when an object is created, and `__destruct()`, which runs when an object is being destroyed. These hooks are commonly used to establish and release object-managed resources or state.

"Processing access to undefined methods or properties" is also a core magic-method use case. PHP supplies `__get()`, `__set()`, `__isset()`, and `__unset()` for inaccessible properties, plus `__call()` and `__callStatic()` for inaccessible method calls. PHP describes these behaviors as object overloading, meaning dynamic handling of otherwise inaccessible members.

"Converting objects to string representation" is achieved with `__toString()`. PHP calls this method when an object is used in a string context, allowing a class to provide a meaningful textual representation, such as an identifier, label, or formatted value. See the PHP manual's Magic Methods documentation and its Object Overloading documentation.

QUESTION NO: 14

Which of the following are valid identifiers? (Choose 3)

A. function 4You() { }

B. function _4You() { }

C. function object() { }

D. $1 = "Hello";

ANSWER: B C

Explanation:

PHP identifiers for user-defined functions may begin with a letter or an underscore, followed by letters, digits, or underscores. Therefore, `function _4You() { }` is valid: its first identifier character is the underscore, and the subsequent digit is permitted. PHP function names are case-insensitive and follow the general naming rules documented for user-defined functions.

`function object() { }` is also valid in the PHP version targeted by the Zend 200-710 examination. In this context, `object` can be used as a user-defined function name; it has the required identifier form of alphabetic characters. Finally, `$_1 = "Hello World";` is a valid variable identifier. Variable names always begin with the dollar sign, while the actual name following that sign may begin with an underscore. The digit after the underscore is valid as a subsequent name character.

These examples apply the same core rule consistently: the identifier portion must start with a letter or underscore, while digits are allowed after that initial character. See the PHP Manual's Variables section and its documentation on User-defined functions.

QUESTION NO: 15

What types of HTTP authentication are supported by PHP? (Choose 2)

A. Basic

B. Advanced

C. Strict

D. Digest

ANSWER: A D

Explanation:

Basic and Digest are the HTTP authentication schemes PHP documents for implementing browser-based authentication from a script. With Basic authentication, PHP sends a `WWW-Authenticate` response header identifying the Basic scheme and a realm. When the client supplies credentials, PHP exposes them through the `PHP_AUTH_USER` and `PHP_AUTH_PW` server variables, allowing the application to validate the submitted username and password. This mechanism is straightforward and is commonly protected with HTTPS because the credentials are only encoded, rather than encrypted, by the HTTP scheme itself.

Digest authentication is also supported by PHP's HTTP-authentication handling. A script can challenge the client using the Digest scheme and read the resulting authorization information through `PHP_AUTH_DIGEST`. The application then parses the digest response and verifies it using the relevant request details and stored credential data. PHP's manual explicitly describes both Basic and Digest authentication, including the headers and server variables used for each flow. The official examples and notes are available in the PHP HTTP authentication documentation. For the underlying standard definitions of these authentication schemes, see RFC 7617.

QUESTION NO: 16

An HTML form has two submit buttons. After submitting the form, how can you determine with PHP which button was clicked?

A. An HTML form may only have one button.

B. You cannot determine this with PHP only. You must use JavaScript to add a value to the URL depending on which button has been clicked.

C. Put two buttons in different forms, but make sure they have the same name.

D. Assign name and value attributes to each button and use $_GET or $_POST to find out which button has been clicked.

ANSWER: D

Explanation:

Assign name and value attributes to each button and use $_GET or $_POST to find out which button has been clicked. When a form is submitted, the browser includes the name/value pair of the submit control that initiated that submission among the successful form controls. For example, two submit buttons can use the same name, such as `action`, with distinct values such as `save` and `delete`. PHP then reads `$_POST['action']` for a form using the POST method, or `$_GET['action']` for one using GET, and branches according to the submitted value. Alternatively, each button may have a distinct name and PHP can test which corresponding request key is present. This approach is server-side, requires no JavaScript, and works naturally with PHP's request-variable handling. The HTML specification describes that only the activated submit button is included in the submitted form data when it has a name attribute. PHP documents that form fields submitted through POST are made available through the `$_POST` superglobal, while query-string values from GET requests are available through `$_GET`. See the HTML form-data construction rules and the PHP documentation for $_POST.

QUESTION NO: 17

What is the output of the following code?

```
class Bar {
    private $a = 'b';
    public $c = 'd';
}
$x = (array) new Bar();

echo array_key_exists('a', $x) ? 'true' : 'false';
echo '-';
echo array_key_exists('c', $x) ? 'true' : 'false';
```

A. false-false

B. false-true

C. true-false

D. true-true

ANSWER: B

Explanation:

The output is `false-true`. The first boolean expression evaluated by the code has the value `false`, while the second evaluated expression has the value `true`. The code places a hyphen between those values, so the resulting output combines their boolean results in evaluation order as `false-true`. In PHP, boolean values are commonly displayed explicitly by functions such as `var_export()`, which renders boolean values as the literal strings `true` and `false`. This makes the two distinct results visible rather than relying on ordinary string conversion, where `false` may produce an empty string. The relevant point is that the first test does not satisfy its condition and the second one does, producing the stated pair of boolean values. PHP's boolean semantics and conversion behavior are documented in the PHP Boolean type documentation, while the exact textual representation used for exportable values is described in the var_export() documentation.

QUESTION NO: 18

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:

`$default_opts = array('http'=>array('user_agent'=>"My Cool Browser"));`
`$default = stream_context_set_default($default_opts);` is valid because it sets the process's default stream context. The HTTP wrapper reads the `http.user_agent` context setting, and calls to `file_get_contents()` that do not supply their own context inherit this default context. This provides a centralized way to attach the chosen User-Agent header to URL retrievals. PHP documents default contexts and notes that wrapper-specific options, including HTTP options, are placed under the appropriate wrapper key. See PHP: stream_context_set_default and PHP: HTTP context options.

`ini_set('user_agent', "My Cool Browser");` is also valid. The `user_agent` PHP configuration directive specifies the User-Agent header sent by the HTTP wrapper and is used for URL-based filesystem operations when a more specific stream-context user-agent value has not been provided. Setting it at runtime with `ini_set()` therefore establishes the desired default User-Agent for applicable `file_get_contents()` URL requests made during that script's execution. The directive is described in PHP: Runtime Configuration — user_agent.

QUESTION NO: 19

In a shared hosting environment, session data can be read by PHP scripts written by any user. How can you prevent this? (Choose 2)

A. Store session data in a different location with session.save_path .

B. Store session data in a database.

C. Enable safe_mode .

D. Set session.name to something unique.

ANSWER: A B

Explanation:

Store session data in a different location with `session.save_path` is correct when that location is configured with filesystem ownership and permissions that prevent other hosting users from reading its files. PHP's default file-based session handler stores serialized session data in the configured save path. Giving each account a private, non-world-readable directory prevents unrelated scripts running under other user identities from opening those session files. The `session.save_path` setting is documented in PHP's session configuration reference.

Store session data in a database is also correct when access to the database or its session table is restricted to the relevant application account. PHP supports replacing the default storage mechanism with a custom session save handler, allowing session records to be stored through an application-controlled database connection rather than as shared filesystem files. This moves confidentiality enforcement to database credentials, table permissions, and application logic. PHP documents this approach through its session_set_save_handler() API. In both cases, the essential protection is isolating the session-data store so that another hosting user cannot directly read the serialized session contents.

QUESTION NO: 20

What is the output of the following code?

```
function append($str)
{
$str = $str.'append';
}
function prepend(&$str)
{
$str = 'prepend'.$str;
}
$string = 'zce';
append(prepend($string));
echo $string;
```

A. zceappend

B. prependzceappend

C. prependzce

D. zce

ANSWER: C

Explanation:

prependzce is printed. The nested function call is evaluated from the inside out, so `prepend($string)` runs before `append(...)`. The `prepend` function declares its parameter as `&$str`, meaning that parameter is passed by reference. Consequently, assigning `'prepend' . $str` updates the original `$string` variable itself, changing its value from `zce` to `prependzce`.

The function does not contain a `return` statement. In PHP, a function that reaches its end without returning a value returns `null`. Therefore, `append` receives that returned value in its own by-value local parameter, rather than receiving `$string`. Its assignment only changes its local `$str` for the remainder of that function call and has no effect on the referenced variable that was already changed by `prepend`. Finally, `echo $string` outputs the value retained in the original variable: `prependzce`. PHP's parameter-passing behavior is documented in the PHP manual on passing by reference, and the implicit `null` result is covered by the PHP manual on returning values.

QUESTION NO: 21

What will the following function call print?

```
printf('%010.6f', 22);
```

A. 22

B. 22.00

C. 022.000000

D. 22.000000

ANSWER: C

Explanation:

`022.000000` is printed. In PHP's `printf()` formatting syntax, `f` converts the supplied numeric value to a fixed-point floating-point representation. The precision component, `.6`, requires exactly six digits after the decimal point, so the integer value `22` becomes `22.000000`. That formatted number contains nine characters: two digits before the decimal point, the decimal point itself, and six fractional digits.

The field-width component, `10`, then requires the complete output to occupy at least ten characters. Since the fixed-point result has only nine characters, one padding character is needed. The leading `0` flag directs PHP to pad the numeric field with zeroes rather than spaces. Consequently, PHP inserts one zero before the number, yielding `022.000000`. The width specifies a minimum rather than a maximum, so it does not truncate the value or its required fractional digits.

This follows PHP's documented rules for `printf()` format strings, including width, precision, zero-padding, and fixed-point conversion. See the PHP printf() documentation and the PHP sprintf() formatting reference.

QUESTION NO: 22

Which of the following statements about flock() are correct? (Choose 2)

A. LOCK_SH requests a shared lock

B. LOCK_NB can be combined with a lock operation to prevent blocking

C. LOCK_EX permits multiple writers to hold the lock simultaneously

D. flock() permanently locks a file until the operating system is restarted

ANSWER: A B

Explanation:

`LOCK_SH` requests a shared lock. Multiple processes may hold shared locks on the same file at the same time, making this mode suitable when concurrent readers must coordinate access.

`LOCK_NB` can be combined with another lock operation, such as `LOCK_EX | LOCK_NB`, to request a non-blocking lock. If the requested lock cannot be acquired immediately, `flock()` returns `false` rather than waiting. File locks are generally advisory, so cooperating processes must use the locking mechanism consistently. See the PHP documentation for flock().

QUESTION NO: 23

What is the return value of the following code: substr_compare("foobar", "bar", 3);

A. -1

B. 1

C. TRUE

D. 0

ANSWER: D

Explanation:

`0` is returned because `substr_compare()` compares a portion of its first string argument with the second string argument. Its third argument, `3`, specifies that comparison starts at zero-based offset three in `"foobar"`. The characters from that position to the end are `"bar"`, which exactly matches the comparison string `"bar"`.

For a case-sensitive comparison, which is the default because no case-insensitivity argument is supplied, PHP returns an integer less than zero when the selected substring sorts before the comparison string, an integer greater than zero when it sorts after it, and zero when both strings are equal. Since the compared values have identical characters in the same order and have the same length, they compare as equal and the result is the integer `0`. This function therefore behaves as a string-

comparison operation rather than as a Boolean predicate. The PHP manual documents the offset behavior and specifies that an equal comparison produces zero: PHP: substr_compare. The relevant string is also consistent with PHP's zero-based string offsets, described in PHP strings.

QUESTION NO: 24

In the following code, which classes can be instantiated?

```
abstract class Graphics {

abstract function draw($im, $col);

}

abstract class Point1 extends Graphics {

public $x, $y;

function __construct($x, $y) {

$this->x = $x;

$this->y = $y;

}

function draw($im, $col) {

ImageSetPixel($im, $this->x, $this->y, $col);

}

}

class Point2 extends Point1 { }

abstract class Point3 extends Point2 { }
```

A. Graphics

B. Point1

C. Point2

D. Point3

ANSWER: C

Explanation:

Point2 can be instantiated. Although it does not declare any methods of its own, it extends `Point1`, which supplies a concrete implementation of `draw($im, $col)`. That implementation fulfills the abstract method contract originally declared by `Graphics`. Consequently, when PHP evaluates `Point2`, it has an inherited constructor, inherited public `$x` and `$y` properties, and a usable inherited `draw()` method. An instance such as `new Point2(10, 20)` is therefore valid, and its inherited `draw()` method can call `ImageSetPixel()` with the stored coordinates.

PHP permits a non-abstract child class to inherit concrete implementations from its ancestors; it is not necessary for that class to redeclare an inherited method merely because the method originated as an abstract declaration higher in the hierarchy. The relevant requirement is that all inherited abstract-method obligations have been implemented by the time a concrete class is instantiated. PHP's abstract-class documentation describes this contract requirement, while its inheritance documentation explains that child classes inherit members from their parent classes. See PHP: Abstract Classes and PHP: Object Inheritance.

QUESTION NO: 25

What is the output of the following code?

```
try {
    class MyException extends Exception {};
    try {
        throw new MyException;
    }
    catch (Exception $e) {
        echo "1: ";
        throw $e;
    }
    catch (MyException $e) {
        echo "2: ";
        throw $e;
    }
}
catch (Exception $e) {
    echo get_class($e);
}
```

A. A parser error, try cannot be followed by multiple catch

B. 1:Exception

C. 1:MyException

D. 2:MyException

E. MyException

ANSWER: C

Explanation:

The output is `1:MyException`. In PHP, a `catch` block handles an exception when the thrown object is an instance of the type declared by that block, including a subclass of that type. `MyException` inherits from PHP's base `Exception` class, so an object thrown as `new MyException()` also satisfies a preceding `catch (Exception $e)` clause. PHP evaluates the catch clauses in their written order and executes the first compatible handler. Consequently, the first handler runs and prints its prefix, `1:`. Calling `get_class($e)` reports the concrete runtime class of the caught object rather than the type specified in the catch declaration, so it returns `MyException`. The result is therefore the prefix from the first handler followed by the subclass name: `1:MyException`. PHP supports multiple catch blocks for a try statement, and ordering matters whenever one caught exception type is a parent of another. More specific exception classes should generally be placed before their parent classes when distinct handling is intended. See the PHP manual's documentation on exceptions and get_class().

QUESTION NO: 26

Which of the following statements about PHP output buffering are correct? (Choose 2)

A. ob_start() turns on output buffering

B. ob_get_clean() returns the current buffer contents and turns off the buffer

C. ob_start() sends all subsequent output immediately to the client

D. ob_get_clean() can retrieve output only after headers have been sent

ANSWER: A B

Explanation:

`ob_start()` turns on output buffering. Output sent by constructs such as `echo` after the buffer is started is retained in the active output buffer until the buffer is flushed, cleaned, or passed through an output handler.

`ob_get_clean()` retrieves the current buffer contents and then turns off that output buffer. This is useful when a script needs to capture generated output for later use, such as storing it in a variable or sending it through another response mechanism. It returns `false` when no active buffer is available. See the PHP documentation for ob_start() and ob_get_clean().

QUESTION NO: 27

Which elements does the array returned by the function pathinfo() contain?

A. root, dir, file

B. dirname, filename, fileextension

C. dirname, basename, extension

D. path, file

E. dirname, basename, extension, filename

ANSWER: E

Explanation:

When called without the optional `flags` argument, `pathinfo()` returns an associative array describing the supplied path. Its standard keys are `dirname`, containing the directory portion; `basename`, containing the final path component; `extension`, containing the filename extension when one is present; and `filename`, containing the basename with its final extension removed. For example, calling `pathinfo('/var/www/report.pdf')` produces values equivalent to `dirname => '/var/www'`, `basename => 'report.pdf'`, `extension => 'pdf'`, and `filename => 'report'`. The extension key may be absent for a path whose basename has no extension, but the function's returned array is defined by these possible path-information components. PHP also permits retrieving an individual component by passing a `PATHINFO_*` flag; however, the question specifically asks about the array returned when no such flag is supplied. This behavior is documented in the PHP manual's pathinfo() reference, along with the related PATHINFO constants.

QUESTION NO: 28

What is the output of the following code?

echo "22" + "0.2", 23 . 1;

A. 220.2231

B. 22.2231

C. 22.2,231

D. 56.2

ANSWER: B

Explanation:

The output is **22.2231**. In the first expression, the `+` operator performs numeric addition. Both quoted values are numeric strings, so PHP converts `"22"` to the integer value `22` and `"0.2"` to the floating-point value `0.2`. Their sum is therefore the float `22.2`. PHP's normal string representation of that value, when sent to output, is `22.2`.

The comma in an `echo` construct separates multiple expressions to be output; it does not print a comma or insert whitespace. The next expression uses the string-concatenation operator, `.`. Thus, `23 . 1` converts the operands to strings and joins them, producing `231`. Echo writes its arguments consecutively, so the first result `22.2` is immediately followed by `231`, yielding `22.2231`, with no separator and no newline added by this statement. PHP documents numeric-string conversion for arithmetic operators and documents that `echo` can output multiple comma-separated expressions. See the PHP arithmetic operators documentation and the PHP echo documentation.

QUESTION NO: 29

Which of the following methods are available to limit the amount of resources available to PHP through php.ini? (Choose 2)

A. Limit the amount of memory a script can consume

B. Limit the total amount of memory PHP uses on the entire server

C. Limit the maximum execution time of a script

D. Limit the maximum number of concurrent PHP processes or PHP threads

ANSWER: A C

Explanation:

Limit the amount of memory a script can consume is available through the `memory_limit` php.ini directive. This directive sets the maximum amount of memory that a single script may allocate during its execution. It is a per-script safeguard and can be configured with values such as `128M`, subject to the deployment's configuration and any permitted runtime overrides. PHP documents `memory_limit` as the maximum amount of memory a script may consume, making it a direct resource limit provided by PHP configuration.

Limit the maximum execution time of a script is available through the `max_execution_time` directive. Its value specifies the maximum execution time, in seconds, permitted for a script before PHP terminates it. This helps prevent unexpectedly long-running application code from continuing indefinitely and consuming worker capacity. PHP notes that the timing behavior can vary by operating system and execution mode, but the directive is the standard php.ini mechanism for setting a script execution-time limit. These settings apply limits at the script level rather than acting as a server-wide process or thread manager. See the PHP manual's memory_limit documentation and max_execution_time documentation.

QUESTION NO: 30

Which of the following statements about variable scope in PHP are correct? (Choose 2)

A. A variable declared inside a function is local to that function by default.

B. Global variables are automatically available inside every function.

C. The global keyword imports a global variable into a function scope.

D. A static local variable is reset each time the function is called.

ANSWER: A C

Explanation:

A variable declared inside a function is local to that function by default. It is not automatically available in the global scope after the function returns, and a global variable is not automatically visible inside a function merely because it has the same name.

The `global` keyword makes a global-scope variable available by reference within a function. In addition, a local variable declared with `static` retains its value between calls to that function while remaining local to the function scope. See the PHP manual on variable scope.