Dynamic webpages with php

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

Preface

PHP is a server-side scripting language commonly used to create dynamic web pages and applications.

A user's browser requests a PHP page from a server (e.g., clicking a link to a .php file).

The server (like Apache or Nginx) identifies the requested file as PHP, passing it to the PHP interpreter.

The PHP interpreter runs the PHP code within the file, executing commands like database queries, logic operations, or form handling.

PHP code can dynamically generate HTML, CSS, JavaScript, or other types of data based on inputs, conditions, or other dynamic factors.

PHP generates output (often HTML) based on the code's logic. This output is usually the web page the user sees.

PHP can also return other types of data, like JSON or XML, often used in APIs and AJAX calls.

The server sends the output back to the client, which the user's browser renders as a regular webpage.

PHP itself is not visible to the end-user, as it runs on the server, unlike client-side languages like JavaScript.

A web server is a computer system or software application that serves content over the internet to users. It hosts websites, processes requests, and delivers webpages or other resources to clients (like browsers). Here's a basic breakdown of how it functions:

- **Hosting Content:** A web server stores files like HTML, CSS, JavaScript, images, videos, and sometimes scripts in languages like PHP or Python.
- **Listening for Requests:** It listens for HTTP (or HTTPS) requests from clients. When you enter a URL in your browser, you're sending a request to a web server to access that specific resource.
- **Processing Requests:** Once the server receives a request, it processes it:

  If it's a static file (like an image or HTML), the server quickly retrieves and sends it back.

  If it's a dynamic file (like PHP or ASP), the server may involve additional processing, such as fetching data from a database or running scripts to generate the content.

  Sending Responses: After processing, the server sends back the content as an HTTP response, which your browser displays.
- **Handling Additional Functions:** Web servers also handle tasks like load balancing (distributing requests to multiple servers for efficiency), security (managing SSL certificates for HTTPS), and caching to speed up response times.

Popular web server software includes Apache, Nginx, and Microsoft's IIS. Each is designed to manage requests, handle different types of files, and ensure reliable delivery of content across the internet.

Setting up apache on arch linux

In order to view code written in .php files, we must set up a web server. On arch linux, we can setup apache. First, install the required packages

```
pacman -Sy apache php php-apache
```

2.1 Configure PHP with Apache

To make Apache process PHP files, you need to edit the Apache configuration.

```
sudo vi /etc/httpd/conf/httpd.conf
```

Add the following lines to the end of the file to load the PHP module and configure PHP handling:

```
# Load PHP module
LoadModule php_module modules/libphp.so
AddHandler php-script .php
Include conf/extra/php_module.conf
```

2.2 Enable PHP in Apache

The file php_module.conf should have been installed with the php-apache package. If it's missing, you may need to create it manually at /etc/httpd/conf/extra/php_module.conf.

```
/etc/httpd/conf/extra/php\_module.conf
```

```
DirectoryIndex index.php index.html
```

This line ensures that Apache will serve index.php as the default file if it's available.

2.3 Start and Enable Apache

```
sudo systemctl start httpd
sudo systemctl enable httpd
```

2.4 Test the PHP Configuration

```
echo "<?php phpinfo(); ?>" | sudo tee /srv/http/index.php
```

Open your web browser and go to http://localhost/index.php. You should see the PHP info page, indicating that PHP is correctly configured.

2.5 Adjust Permissions

If you plan to edit files in /srv/http frequently, you might want to adjust the permissions:

```
sudo chown -R $USER:http /srv/http
sudo chmod -R 755 /srv/http
```

2.6 Allow http traffic

If you have a firewall enabled, allow HTTP traffic:

```
sudo iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

Or with ufw,

```
sudo ufw allow http
```

Once you've set up Apache and PHP as described, you can view .php files by placing them in the web root directory and accessing them via a web browser.

2.7 Test apache config

Sometimes, a small syntax error in the Apache configuration file can cause the server to fail to start. Run the following command to check the configuration:

```
sudo apachectl configtest
```

If there's an error, this command will display a message that can help pinpoint the issue.

2.8 Pages not loading, change default php module

switch to the mpm_prefork module, which works well with the default PHP module on Arch Linux.

Edit the Apache configuration file /etc/httpd/conf/httpd.conf and comment out or remove the line loading the mpm_event module:

```
# LoadModule mpm_event_module modules/mod_mpm_event.so      # Comemnt
LoadModule mpm_prefork_module modules/mod_mpm_prefork.so    # add
```

Then restart apache

```
sudo systemctl restart httpd
```

2.9 Apache document root

If you'd prefer to store your PHP files outside of /srv/http, you can change the Apache DocumentRoot to point to a different directory.

First, make the directory you want as root

```
mkdir -p ~/mywebsite
```

Open the Apache configuration file:

```
sudo vi /etc/httpd/conf/httpd.conf
```

Locate the DocumentRoot directive, which should look like this:

```
DocumentRoot "/srv/http"
```

Change this to the path of your new directory, such as:

```
DocumentRoot "/home/yourusername/mywebsite"
```

Also, update the <Directory> block for /srv/http to match your new directory:

```
<Directory "/home/yourusername/mywebsite">
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>
```

Ensure that Apache has permission to read files in your chosen directory:

```
sudo chown -R $USER:http ~/mywebsite
sudo chmod -R 755 ~/mywebsite
```

After making these changes, restart Apache to apply the new configuration:

```
sudo systemctl restart httpd
```

Place your PHP files in the new directory (/mywebsite). You can then access them in your browser as before:

```
http://localhost/test.php
```

This way, you can keep your PHP files in a custom directory without needing to store everything in /srv/http.

2.10 Permission error: 403

For Apache to access files in this directory, it needs execute (x) permissions on each parent directory in the path.

Ensure that each directory in the path (/home and /home/*username*) has the appropriate execute permissions for Apache to access subdirectories. This doesn't mean Apache will have access to all files, but it will allow it to "traverse" the directories.

```
sudo chmod o+x /home
sudo chmod o+x /home/username
```

This will allow "other" users (including the Apache user) to traverse the /home and /home/*username* directories.

You should also change permissions on the documentroot directory, and possible the parent directories. Then restart httpd

```
sudo chown -R $USER:http ~/documentroot
sudo chmod -R 755 ~/documentroot
sudo systemctl restart httpd
```

- **$USER:** This is an environment variable that represents the currently logged-in user's username. By using $USER, you're setting yourself as the owner of the directory and its contents, allowing you to manage the files easily.
- **http:** This is the group associated with the Apache server on many Linux distributions (including Arch Linux). Assigning the http group to the directory allows Apache to access the files and directories within, provided the group has the necessary permissions.

Setting up mariadb on arch linux

First, update your package database and install MariaDB:

```
sudo pacman -Syu
sudo pacman -S mariadb
```

Before starting MariaDB, initialize the database. This will set up the system databases and files.

```
sudo mariadb-install-db --user=mysql --basedir=/usr
↪  --datadir=/var/lib/mysql
```

Once initialized, start the MariaDB service and enable it to start on boot:

```
sudo systemctl start mariadb
sudo systemctl enable mariadb
```

Run the secure installation script to set up security options, such as the root password and disabling remote root login:

```
sudo mysql_secure_installation
```

To verify that the setup works, you can log into MariaDB as the root user:

```
sudo mysql -u root -p
```

Configuration files for MariaDB are in /etc/my.cnf.d/ and /etc/mysql/my.cnf. You can adjust settings here based on your requirements.

To confirm that MariaDB is running correctly, use:

```
systemctl status mariadb
```

In the MariaDB prompt, create a new user with the following syntax:

```
CREATE USER 'username'@'localhost' IDENTIFIED BY 'password';
```

You can assign specific privileges to the new user. For example, to give full access to a particular database, use:

```
GRANT ALL PRIVILEGES ON database_name.* TO
↪  'username'@'localhost';

FLUSH PRIVILEGES;
```

Then that user can login with

```
mysql -u username -p
```

php.ini

php.ini is the main configuration file for PHP. It controls PHP's behavior, settings, and options on the server where PHP is installed. The file allows you to customize PHP's performance, security, error handling, file handling, and much more, making it essential for PHP's runtime environment.

The php.ini file is typically located in the PHP installation directory.

The exact path can vary depending on the server and operating system. Common paths include /etc/php.ini (Linux), /etc/php/[version]/cli/php.ini, or C:\xampp\php\php.ini (Windows with XAMPP).

To find the exact location, you can run phpinfo() in a PHP script, which will display the Loaded Configuration File path.

4.1 Error Reporting

Controls which types of errors are reported and logged.

```
error_reporting = E_ALL
display_errors = On
display_startup_errors = On;
log_errors = On
error_log = "/path/to/php-error.log"
```

4.2 Memory Limit

Sets the maximum memory PHP scripts are allowed to use. Useful for preventing memory leaks and controlling resource usage.

```
memory_limit = 128M
```

4.3 File Uploads

Controls file upload settings for handling user-uploaded files.

```
file_uploads = On
upload_max_filesize = 2M
post_max_size = 8M
```

4.4 Max Execution Time

Sets the maximum time (in seconds) that a script is allowed to run.

Prevents poorly written scripts from running indefinitely and consuming server resources.

```
max_execution_time = 30
```

4.5 Session Settings

Manages session behavior and storage options.

```
session.save_path = "/path/to/sessions"
session.gc_maxlifetime = 1440
```

4.6 Short Open Tag

Enables or disables the short <? ... ?> tag syntax in PHP.

```
short_open_tag = Off
```

4.7 Timezone

Sets the default timezone for PHP.

```
date.timezone = "America/New_York"
```

Showing errors

If you don't want to enable errors in php.ini, you can set them for a specific php file. Simply add the following at the very top of a php file

```php
ini_set('display_errors', 1);
ini_set('display_startup_errors', 1);
error_reporting(E_ALL);
```

php in .php

6.1 php blocks

php code should be contained with php blocks

```
example.php

<?php #no space between ? and php
    ... php code
?>
```

We can have as many blocks as we want, or have everything in one block. Anything not in these blocks are sent to the server as plaintext (or html if its html code).

6.2 Short tags

<?= ... ?> is shorthand for <?php echo ... ?>, commonly used to output data directly in HTML.

Enabled by default in modern PHP versions.

Useful for quickly embedding PHP output in HTML.

```
<p>Hello, <?= $name; ?>!</p>
```

6.3 Short open tags

There is an additional shorthand php tag, called the short open tag

```
<? ... ?>
```

This is the shorthand version of <?php, without the php keyword.

Often discouraged because it may not be enabled on all servers (it depends on the short_open_tag directive in php.ini).

Not recommended for compatibility reasons; using <?php ... ?> is generally safer.

Note that the short echo tag is enabled by default since PHP 5.4

Outputting data

7.1 Echo

echo is one of the most commonly used output functions in PHP.

It can output one or more strings separated by commas and does not return a value.

Faster than print in most cases because it does not return a value.

```
1  echo "Hello, World!";
2  echo "This ", "is ", "PHP ", "output.";
```

7.2 Print

Print is another commonly used output function.

Unlike echo, print returns 1, making it slightly slower than echo.

Only supports a single argument (cannot separate strings by commas like echo).

```
1  print "Hello, World!";
```

7.3 Print_r

print_r is primarily used for debugging because it outputs human-readable information about a variable.

It can output arrays and objects in a way that's easy to understand.

Useful for printing array structures, often used in debugging.

```
1  $array = array("apple", "banana", "cherry");
2  print_r($array);
```

7.4 var_dump

var_dump provides detailed information about a variable, including data type and length.

Commonly used in debugging, especially to inspect complex data structures like arrays and objects.

It outputs the value along with type information and nested data.

```
$variable = 42;
var_dump($variable);

$array = array("apple", "banana", "cherry");
var_dump($array);
```

7.5 var_export

var_export outputs a parsable string representation of a variable, making it suitable for creating PHP code.

Often used when you want to output PHP code that represents a value.

Similar to var_dump, but the output is valid PHP code.

```
$array = array("apple", "banana", "cherry");
var_export($array);
```

7.6 sprintf

sprintf formats a string and returns it without printing it immediately.

Useful for formatting strings with variables in complex ways before outputting them with echo or print.

```
$name = "Alice";
$age = 30;
$formatted = sprintf("Name: %s, Age: %d", $name, $age);
echo $formatted;
```

7.7 die and exit

die and exit both output a message (if provided) and stop the script.

Often used in error handling to output an error message and terminate the script.

```
die("An error occurred.");
exit("Stopping execution.");
```

Lexical structure

8.1 Case Sensitivity

The names of user-defined classes and functions, as well as built-in constructs and keywords such as echo, while, class, etc., are case-insensitive.

Variables are case sensitive.

8.2 Statements and Semicolons

Use semicolons to separate statements, lines must end with a semicolon.

8.3 Whitespace and Line Breaks

In general, whitespace doesn't matter in a PHP program. You can spread a statement across any number of lines, or lump a bunch of statements together on a single line. For example, this statement:

```
raisePrices($inventory, $inflation, $costOfLiving, $greed);
```

Could just as well be written with more whitespace

```
raisePrices (
    $inventory ,
    $inflation ,
    $costOfLiving ,
    $greed
);
```

8.4 Comments

Comments can be made with #, // or */.

8.5 Variables

Variable names always begin with a dollar sign ($) and are case-sensitive

8.5.1 Variable Variables

You can reference the value of a variable whose name is stored in another variable by prefacing the variable reference with an additional dollar sign ($)

```
$foo = "bar";
$$foo = "baz";
```

After the second statement executes, the variable $bar has the value ”baz”.

8.5.2 Variable References

In PHP, references are how you create variable aliases. To make $black an alias for the variable $white, use

```
$black = &$white;
```

The old value of $black, if any, is lost. Instead, $black is now another name for the value that is stored in $white:

Unsetting a variable that is aliased does not affect other names for that variable's value

```
$white = "snow";
$black =& $white;
unset($white);
print $black;

snow
```

8.5.3 Returning by reference

Functions can return values by reference

```
function &retRef() // note the &
{
    $var = "PHP";
    return $var;
}
$v =& retRef(); // note the &
```

8.5.4 Static variables

A static variable retains its value between calls to a function but is visible only within that function. You declare a variable static with the static keyword

```
function updateCounter()
{
    static $counter = 0;
    $counter++;
    echo "Static counter is now {$counter}\n";
}

$counter = 10;
updateCounter();
updateCounter();
```

8.6 Const variables

We can declare variables constant in php with the *const* qualifier. This lets us drop the $.

```
const a = 12;
echo a; // Notice no dollar sign
```

Note that we cannot pass const variables by reference to functions.

8.7 Defined Constants

A constant is an identifier for a simple value; only scalar values—Boolean, integer, double, and string—can be constants. Once set, the value of a constant cannot change. Constants are referred to by their identifiers and are set using the define() function:

```
define('PUBLISHER', "O'Reilly & Associates");
echo PUBLISHER;
```

Reserved words

9.1 Magic constants

A keyword (or reserved word) is a word set aside by the language for its core functionality—you cannot give a variable, function, class, or constant the same name as a keyword. Here we shall list and describe the keywords in PHP, which are case-insensitive.

9.1.1 ___class___

Returns the name of the current class as a string. Inside a class method, it returns the name of the class.

9.1.2 ___dir___

Returns the directory of the file. Useful for file path operations.

9.1.3 ___file___

Returns the full path and filename of the file.

9.1.4 ___function___

Returns the name of the current function as a string

9.1.5 ___line___

Returns the line number in the file.

9.1.6 ___method___

Returns the name of the current method in the format ClassName::methodName.

9.1.7 ___namespace___

Returns the name of the current namespace as a string.

9.1.8 ___trait___

Returns the name of the current trait.

9.1.9 ___halt_compiler()

Stops PHP from parsing the rest of the file. Useful in embedded scripts or for providing data after PHP code.

9.2 Object-Oriented Programming Keywords

9.2.1 abstract

Used to declare an abstract class or abstract method. Abstract classes cannot be instantiated, and abstract methods must be defined in child classes.

9.2.2 extends

Used to indicate that a class is inheriting from a parent class.

9.2.3 final

When applied to a class, it prevents that class from being extended. When applied to a method, it prevents the method from being overridden in child classes.

9.2.4 interface

Defines a contract for classes. Interfaces contain method declarations but no implementations. Classes that implement an interface must define all its methods.

9.2.5 insteadof

Used in conjunction with traits to resolve conflicts when two traits have methods with the same name.

9.2.6 new

instantiates a new object of a class.

9.2.7 private

Declares properties or methods that are only accessible within the class.

9.2.8 protected

Declares properties or methods that are accessible within the class and its subclasses.

9.2.9 public

Declares properties or methods that are accessible from anywhere.

9.3 Control Structures

9.3.1 and

Logical operator used to combine conditions. Similar to &&, but with lower precedence.

9.3.2 as

Used in foreach loops to assign array keys and values to variables.

9.3.3 break

Exits a loop or switch statement prematurely.

9.3.4 continue

Used within loop structures to skip the current iteration and move directly to the next iteration of the loop. It's typically used in for, foreach, while, and do-while loops.

9.3.5 else

Specifies a block of code to execute if a condition is false.

9.3.6 elseif

Specifies a new condition to check if the previous condition was false.

9.3.7 enddeclare

Ends a declare block.

9.3.8 endfor

Ends a for loop.

9.3.9 endforeach

Ends a foreach loop.

9.3.10 endif

Ends an if block.

9.3.11 endswitch

Ends a switch block.

9.3.12 endwhile

Ends a while loop.

9.3.13 exit()

Terminates script execution immediately. It can take an optional status code as an argument.

9.4 Built-in Functions and Constructs

9.4.1 array()

Creates an array. PHP's array type can hold values of any data type.

9.4.2 echo

Outputs one or more strings. It's a language construct, so parentheses are optional.

9.4.3 empty()

Checks if a variable is empty (e.g., null, 0, "", or an unset variable).

9.4.4 eval()

Parses and executes a string of PHP code. Generally discouraged due to potential security risks.

9.4.5 isset()

Checks if a variable is set (i.e., it's been declared and is not null).

9.4.6 list()

Assigns values to a list of variables. Often used to unpack arrays.

9.4.7 print()

Outputs a string, similar to echo, but has a return value of 1 (which means it can be used in expressions).

9.4.8 require

Includes and evaluates a specified file. If the file is not found, it causes a fatal error, stopping script execution.

9.4.9 require_once

Similar to require, but it only includes the file if it hasn't been included before. Prevents multiple inclusions.

9.4.10 return

Exits a function and optionally returns a value.

9.5 Namespace management

9.5.1 namespace

Defines a namespace, allowing classes, functions, and constants to be organized into distinct groups to prevent naming conflicts.

9.6 Function and Class Handling

9.6.1 callable

Indicates that a value can be called as a function. Often used to type-hint function parameters that can accept functions or callable objects.

9.6.2 class

Defines a class in PHP, which is a blueprint for creating objects with properties and methods.

9.6.3 const

Defines a constant within a class or globally. Constants are immutable once set.

9.6.4 function

Defines a function in PHP, which is a block of reusable code that can accept parameters and return values.

9.6.5 implements

Used with classes to specify that they implement certain interfaces. The class must define all methods in the interface.

9.6.6 instanceof

Checks if an object is an instance of a specific class or interface.

9.6.7 static

Declares properties or methods that belong to the class rather than an instance. Static properties and methods can be accessed without creating an instance of the class.

9.6.8 trait

A mechanism for code reuse in PHP. Traits allow classes to include methods without inheriting from a parent class.

9.6.9 use

Imports namespaces or traits into a class or file, allowing them to be used without fully qualifying their names.

9.6.10 var

An alias for `public` in early PHP versions. It is now deprecated; use `public` instead.

9.7 Control Structures

9.7.1 case

Used within a `switch` statement to define branches based on specific values.

9.7.2 catch

Defines a block of code to execute when an exception is thrown in a `try` block.

9.7.3 declare

Defines directives for code execution, such as enabling strict types.

9.7.4 default

Specifies the fallback branch in a `switch` statement if no `case` matches.

9.7.5 do

Used with `while` to create a `do-while` loop that executes at least once.

9.7.6 for

Initiates a `for` loop, which iterates a specified number of times based on initialization, condition, and increment expressions.

9.7.7 foreach

Loops through each element of an array, assigning the current element's value to a specified variable.

9.7.8 goto

Jumps to a specified label within the code. Generally discouraged due to readability concerns.

9.7.9 if

Begins a conditional statement that executes code only if a specified condition is true.

9.7.10 switch

Evaluates a variable against multiple cases and executes the matching block of code.

9.7.11 throw

Used to throw an exception within a `try-catch` structure.

9.7.12 try

Defines a block of code to attempt, where any thrown exceptions are caught in associated `catch` blocks.

9.7.13 while

Initiates a `while` loop that continues as long as the specified condition is true.

9.8 Built-in Functions and Logical Operators

9.8.1 die()

An alias for `exit()`, immediately terminating script execution and optionally outputting a message.

9.8.2 include

Includes and evaluates a specified file in the current script. Throws a warning if the file is not found.

9.8.3 include_once

Includes a specified file only once, preventing re-inclusion errors.

9.8.4 unset()

Destroys a specified variable, freeing up its memory.

9.8.5 xor

Logical operator representing exclusive OR. Returns true if only one of the operands is true.

Data types

- **String:** A sequence of characters used to store and manipulate text.
- **Integer:** A whole number, positive or negative, without a decimal point.
- **Float:** A number with a decimal point or in exponential form, used to represent fractional values.
- **Boolean:** Represents two possible states: true or false.
- **Array:** A collection of values, indexed by integers (numeric array) or strings (associative array). Arrays can store multiple types of data within them.
- **Object:** An instance of a class, which can hold both data (properties) and functions (methods) related to a specific concept or model.
- **Null:** A special data type representing a variable with no value. Any variable can be explicitly set to NULL.
- **Resource:** A special data type that holds references to external resources, such as database connections or file handles. It is used to interact with these resources rather than storing values directly.
- **Callable**: Represents a function that can be called, which could be a function name, a method, or an anonymous function.

```php
function myFunction() { return "Hello"; }
$callableFunction = 'myFunction';
echo $callableFunction();
```

When you assign the function name as a string to a variable (like myFunction in your example), PHP treats it as a callable. If the function is defined and exists in the current scope, you can call it by using the variable as a function.

- **Iterable**: Represents a function that can be called, which could be a function name, a method, or an anonymous function.

```php
function printIterable(iterable $myIterable) {
    foreach ($myIterable as $item) {
        echo $item;
    }
}
```

10.1 Strings

10.1.1 Variable Interpolation

When you define a string literal using double quotes or a heredoc, the string is subject to variable interpolation. Interpolation is the process of replacing variable names in the string with the values of those variables. There are two ways to interpolate variables into strings.

The simpler of the two ways is to put the variable name in a double-quoted string or heredoc:

```
$who = 'Kilroy';
$where = 'here';
echo "$who was $where";
Kilroy was here
```

The other way is to surround the variable being interpolated with curly braces. Using this syntax ensures the correct variable is interpolated. The classic use of curly braces is to disambiguate the variable name from surrounding text:

```
$n = 12;
echo "You are the {$n}th person";
You are the 12th person
```

Unlike in some shell environments, in PHP strings are not repeatedly processed for interpolation. Instead, any interpolations in a double-quoted string are processed first and the result is used as the value of the string:

10.1.2 Single-Quoted Strings

Single-quoted strings do not interpolate variables. Thus, the variable name in the following string is not expanded because the string literal in which it occurs is singlequoted:

The only escape sequences that work in single-quoted strings are \', which puts a single quote in a single-quoted string, and \\, which puts a backslash in a single-quoted string. Any other occurrence of a backslash is interpreted simply as a backslash

10.1.3 Double-Quoted Strings

Double-quoted strings interpolate variables and expand the many PHP escape sequences.

10.1.4 Here Documents

You can easily put multiline strings into your program with a heredoc, as follows:

```
$s = <<< END
    Hello
    World
    This
    Is a
    HereDoc
    END;

echo $s; // Hello World This is a HereDoc
```

The identifier *END* is arbitrary, but it needs to be the same at the start and end.

As a special case, you can put a semicolon after the terminating identifier to end the statement, as shown in the previous code. If you are using a heredoc in a more complex expression, you need to continue the expression on the next line, as shown here

```
printf(<<< Template
%s is %d years old.
Template
, "Fred", 35);
```

Single and double quotes in a heredoc are passed through:

```
$dialogue = <<< NoMore
"It's not going to happen!" she fumed.
He raised an eyebrow. "Want to bet?"
NoMore;
echo $dialogue;
"It's not going to happen!" she fumed.
He raised an eyebrow. "Want to bet?"
```

Whitespace in a heredoc is also preserved. The newline before the trailing terminator is removed. If you want a newline to end your heredoc-quoted string, you'll need to add an extra one yourself

```
$s <<< END
Hello
World!

END;
```

10.1.5 printf specifiers

Specifier	Meaning
%	Displays the % character.
b	The argument is an integer and is displayed as a binary number.
c	The argument is an integer and is displayed as the character with that value.
d	The argument is an integer and is displayed as a decimal number.
e	The argument is a double and is displayed in scientific notation.
E	The argument is a double and is displayed in scientific notation using uppercase letters.
f	The argument is a floating-point number and is displayed as such in the current locale's format.
F	The argument is a floating-point number and is displayed as such.
g	The argument is a double and is displayed either in scientific notation (as with the %e type specifier) or as a floating-point number (as with the %f type specifier), whichever is shorter.
G	The argument is a double and is displayed either in scientific notation (as with the %E type specifier) or as a floating-point number (as with the %f type specifier), whichever is shorter.
o	The argument is an integer and is displayed as an octal (base-8) number.
s	The argument is a string and is displayed as such.
u	The argument is an unsigned integer and is displayed as a decimal number.
x	The argument is an integer and is displayed as a hexadecimal (base-16) number; lowercase letters are used.
X	The argument is an integer and is displayed as a hexadecimal (base-16) number; uppercase letters are used.

Each substitution marker in the template consists of a percent sign (%), possibly followed by modifiers from the following list, and ends with a type specifier. (Use %% to get a single percent character in the output.) The modifiers must appear in the order in which they are listed here:

A padding specifier denoting the character to use to pad the results to the appropriate string size. Specify 0, a space, or any character prefixed with a single quote. Padding with spaces is the default.

A sign. This has a different effect on strings than on numbers. For strings, a minus (-) here forces the string to be left-justified (the default is to right-justify). For numbers, a plus (+) here forces positive numbers to be printed with a leading plus sign (e.g., 35 will be printed as +35).

The minimum number of characters that this element should contain. If the result would be less than this number of characters, the sign and padding specifier govern how to pad to this length.

For floating-point numbers, a precision specifier consisting of a period and a number; this dictates how many decimal digits will be displayed. For types other than double, this specifier is ignored.

10.1.6 Basic string functions

- **strlen():** Get the length of a string.
- **str_word_count():** Count the number of words in a string.
- **strrev():** Reverse a string.
- **strpos():** Find the position of the first occurrence of a substring.
- **stripos():** Find the position of the first occurrence of a substring (case-insensitive).
- **strrpos():** Find the position of the last occurrence of a substring.
- **strripos():** Find the position of the last occurrence of a substring (case-insensitive).
- **substr():** Return a part of a string.
- **substr_count():** Count the number of substring occurrences.
- **strstr()**: he strstr() function finds the first occurrence of a substring within a string and returns the portion of the string starting from that substring.
- **strrchr()**: find last occurence of substr
- **strspn()**: The strspn() function finds the length of the initial segment of a string that consists entirely of characters from a specified mask (set of characters). This is useful for checking the length of a prefix that only contains certain characters.
- **strcspn()**: The strcspn() function finds the length of the initial segment of a string that does not contain any characters from a specified mask (set of characters). It stops counting when it encounters a character from the mask.

10.1.7 String Manipulation

- **str_replace():** Replace all occurrences of a substring.
- **str_ireplace():** Replace all occurrences of a substring (case-insensitive).
- **trim():** Strip whitespace (or other characters) from the beginning and end of a string.
- **ltrim():** Strip whitespace (or other characters) from the beginning of a string.
- **rtrim():** Strip whitespace (or other characters) from the end of a string.
- **strtoupper():** Convert a string to uppercase.
- **strtolower():** Convert a string to lowercase.
- **ucfirst():** Make the first character uppercase.
- **lcfirst():** Make the first character lowercase.
- **ucwords():** Uppercase the first character of each word.

10.1.8 String comparison

- **strcmp():** Binary safe string comparison.
- **strcasecmp():** Binary safe case-insensitive string comparison.
- **strnatcmp():** Natural order string comparison.
- **strnatcasecmp():** Case-insensitive natural order string comparison.
- **strcoll():** Locale-based string comparison.
- **similar_text()**: The similar_text() function returns the number of characters that its two string arguments have in common. The third argument, if present, is a variable in which to store the commonality as a percentage

10.1.9 Formatting and Conversion

- **number_format():** Format a number with grouped thousands.
- **sprintf():** Return a formatted string.
- **printf():** Output a formatted string.
- **vprintf():** Output a formatted string with an array of values.
- **htmlspecialchars():** Convert special characters to HTML entities.
- **strip_tags()**: used to remove HTML and PHP tags from a string
- **get_meta_tags()**: used to extract the meta tags from an HTML document and return them as an associative array. It's particularly useful for fetching metadata information from a webpage, such as the description, keywords, or any custom meta tags used in SEO
- **html_entity_decode():** Convert HTML entities back to characters.
- **str_pad():** Pad a string to a new length.
- **addslashes()**: The addslashes() function adds backslashes to certain characters in a string. Escapes single quotes, double quotes, backslashes, and null characters \0
- **stripslashes()**: The stripslashes() function is the inverse of addslashes(). It removes any backslashes added by addslashes(), making the string readable again by removing the escape sequences.
- **addcslashes()**: The addcslashes() function adds backslashes to characters in a string based on a specified character list or range. This is useful for escaping special characters or creating custom escape sequences for characters you want to protect in output.
- **stripcslashes()**: The stripcslashes() function is the inverse of addcslashes(). It removes backslashes added by addcslashes(), interpreting escape sequences in the style of C language escape sequences.
- **sscanf()**: The sscanf() function reads formatted data from a string, parsing it according to specified format specifiers. It's similar to scanf() in C and is used to extract values from a string based on a format pattern.
- **parse_url()**: parse_url() function returns an array of components of a URL

10.1.10 Encoding and Decoding

- **md5():** Calculate the MD5 hash of a string.
- **sha1():** Calculate the SHA-1 hash of a string.
- **base64_encode():** Encode data with Base64.
- **base64_decode():** Decode data encoded with Base64.
- **urlencode():** Encode a URL.
- **urldecode():** Decode a URL-encoded string

10.1.11 Additional Functions

- **explode():** Split a string by a string.
- **implode() (alias join()):** Join array elements with a string.
- **str_split():** Convert a string to an array.
- **addslashes():** Quote a string with slashes.
- **str_repeat():** Repeat a string a specified number of times.

10.2 Arrays

In PHP, arrays are versatile data structures that allow you to store multiple values in a single variable. Arrays can hold values of different data types and can be indexed numerically, associatively (by string keys), or both.

10.2.1 Indexed (numeric) arrays

Indexed arrays use numbers as keys (starting from 0 by default).

```php
$colors = ["red", "green", "blue"];
echo $colors[0];  // Outputs: red
```

You can also create indexed arrays using the array() function:

```php
$colors = array("red", "green", "blue");
```

10.2.2 Associative Arrays

Associative arrays use strings as keys, allowing you to assign meaningful names to values.

```php
$person = [
    "name" => "John",
    "age" => 30,
    "city" => "New York"
];
echo $person["name"];  // Outputs: John
```

10.2.3 Multidimensional Arrays

A multidimensional array is an array that contains other arrays. It allows for more complex data structures, such as a matrix or a table of data.

```php
$employees = [
    ["name" => "Alice", "age" => 28, "department" => "Sales"],
    ["name" => "Bob", "age" => 34, "department" => "Marketing"]
];
echo $employees[0]["name"];  // Outputs: Alice
```

10.2.4 Array functions

PHP provides many built-in functions for working with arrays:

- **count($array):** Returns the number of elements in an array.
- **array_push($array, $value):** Adds an element to the end of an array.
- **array_pop($array):** Removes the last element of an array.
- **array_merge($array1, $array2):** Combines two or more arrays.
- **array_keys($array):** Returns an array of all the keys in an array.
- **array_values($array):** Returns an array of all the values in an array.
- **in_array($value, $array):** Checks if a value exists in an array.
- **array_key_exists(key, array)**: To see if an element exists in the array
- **array_slice($array, $offset, $length):** Extracts a portion of an array.
- **sort($array):** Sorts an array in ascending order.
- **rsort()**: See the sorting section below
- **usort()**: See the sorting section below
- **arsort()**: See the sorting section below
- **krsort()**: See the sorting section below
- **uasort()**: See the sorting section below
- **uksort()**: See the sorting section below
- **ksort($array):** Sorts an associative array by key.
- **asort($array):** Sorts an associative array by value.
- **natsort()**: See the sorting section below
- **natcasesort()**: See the sorting section below
- **arary_multisort()**: Sort parallel arrays

- **range()**: The range() function creates an array of consecutive integer or character values between and including the two values you pass to it as arguments. Only the first letter of a string argument is used to build the range:
- **array_pad()**: The array_pad() function in PHP is used to pad an array to a specified length with a given value. It can be used to extend an array to a larger size, filling additional slots with a specified value, or even to shrink it (though it won't remove elements if the specified length is smaller than the array's current size).
- **array_chunk(array, size, [, preserve_keys])**: To divide an array into smaller, evenly sized arrays, use the array_chunk() function:
- **array_splice(array, start, [length], [replacement])**: The array_splice() function can remove or insert elements in an array and optionally create another array from the removed elements:
- **extract()**: Create locals from keys in an array
- **compact()**: The compact() function is the reverse of extract(). Pass it the variable names as strings to compact either as separate parameters or in an array. The compact() function creates an associative array whose keys are the variable names and whose values are the variable's values. Any names in the array that do not correspond to actual variables are skipped. If a variable created by the extraction has the same name as an existing one, the variable's value is overwritten with that from the array.
- **array_reverse()**: The array_reverse() function reverses the internal order of elements in an array:
- **array_flip()**: The array_flip() function returns an array that reverses the order of each original element's key-value pair:
- **shuffle()**: Randomize array.
- **array_sum()**: Accumulate elements in an array.
- **array_diff()**: The array_diff() function identifies values from one array that are not present in others:
- **array_filter()**: To identify a subset of an array based on its values, use the array_filter() function:
- **array_unique()**: Remove duplicates from an array
- **array_intersect()**: Make an array based on the intersection of n-arrays

10.2.5 Iterating Over Arrays

Arrays are commonly used with loops to access or modify each element.

The foreach loop is ideal for iterating over arrays. It works with both indexed and associative arrays.

```php
// Indexed array
$colors = ["red", "green", "blue"];
foreach ($colors as $color) {
    echo $color . " ";
}

// Associative array
$person = ["name" => "John", "age" => 30, "city" => "New York"];
foreach ($person as $key => $value) {
    echo "$key: $value\n";
}
```

For indexed arrays, you can also use a for loop, especially if you need the index.

10.2.6 Adding and Removing Elements

Use array_push() to add to the end, or simply assign a value to a new index or key.

```php
$colors = ["red", "green"];
$colors[] = "blue";         // Adds "blue" to the end
$colors["favorite"] = "red"; // Adds an associative element
```

Use array_pop() to remove the last element, unset() to remove a specific element by key or index, or array_shift() to remove the first element.

```php
unset($colors[0]); // Removes the first element
array_shift($colors); // Removes the first element, reindexing
↪ the array
```

10.2.7 Array Operators

PHP provides operators to work with arrays:

- +: Union of two arrays. Combines arrays, but only keeps the first value for duplicate keys.
- ==: Checks if two arrays have the same key-value pairs.
- ===: Checks if two arrays are identical in both key-value pairs and order.

10.2.8 Array Casting

In PHP, you can cast a variable to an array type using (array). This is useful when you need to ensure a variable is treated as an array.

```
$number = 5;
$array = (array)$number;
```

10.2.9 Array unpacking

To copy all of an array's values into variables, use the list() construct:

```
list ($variable, ...) = $array;
```

The array's values are copied into the listed variables in the array's internal order. By default that's the order in which they were inserted, but the sort functions described later let you change that

```
$person = array("Fred", 35, "Betty");
list($name, $age, $wife) = $person;

```

If you have more values in the array than in the list(), the extra values are ignored

If you have more values in the list() than in the array, the extra values are set to NULL:

Two or more consecutive commas in the list() skip values in the array:

Combine array_slice() with list() to extract only some values to variables:

10.2.10 Extract and compact

The extract() function automatically creates local variables from an array. The indices of the array elements become the variable names

```
$person = array('name' => "Fred", 'age' => 35, 'wife' =>
↪  "Betty");
extract($person);
```

If a variable created by the extraction has the same name as an existing one, the variable's value is overwritten with that from the array.

You can modify extract()'s behavior by passing a second argument. The most useful value is EXTR_PREFIX_ALL, which indicates that the third argument to extract() is a prefix for the variable names that are created. This helps ensure that you create unique variable names when you use extract(). It is good PHP style to always use EXTR_PREFIX_ALL, as shown here:

```
$shape = "round";
$array = array('cover' => "bird", 'shape' => "rectangular");
extract($array, EXTR_PREFIX_ALL, "book");
echo "Cover: {$book_cover}, Book Shape: {$book_shape}, Shape:
↪  {$shape}";
Cover: bird, Book Shape: rectangular, Shape: round
```

The compact() function is the reverse of extract(). Pass it the variable names to compact either as separate parameters or in an array. The compact() function creates an associative array whose keys are the variable names and whose values are the variable's values. Any names in the array that do not correspond to actual variables are skipped. Here's an example of compact() in action:

```
$color = "indigo";
$shape = "curvy";
$floppy = "none";

$a = compact("color", "shape", "floppy");
// Or
// or
$names = array("color", "shape", "floppy");
$a = compact($names);
```

10.2.11 array_walk()

PHP provides a mechanism, array_walk(), for calling a user-defined function once per element in an array:

```php
array_walk(array, callable);
```

The function you define takes in two or, optionally, three arguments: the first is the element's value, the second is the element's key, and the third is a value supplied to array_walk() when it is called.

```php
$a = array("key1" => "value1", "key2" => "value2", "key3" =>
↪  "value3");

$callable = fn($value, $key) => print("$value"." ");

array_walk($a, $callable);
```

10.2.12 array_reduce

array_reduce() applies a function to each element of the array in turn, to build a single value:

```php
$result = array_reduce(array, callable [, default ]);
```

The function takes two arguments: the running total, and the current value being processed. It should return the new running total. For instance, to add up the squares of the values of an array, use

```php
$a = array(2,2,2,2);

$callable = fn($total, $n) => $total+=$n*$n;

echo array_reduce($a, $callable); // 16
```

If the array is empty, array_reduce() returns the default value. If no default value is given and the array is empty, array_reduce() returns NULL

10.3 Sorting arrays

The functions provided by PHP to sort an array are shown in the table below

Effect	Ascending	Descending	User-defined order
Sort array by values, then reassign indices starting with 0	sort()	rsort()	usort()
Sort array by values	asort()	arsort()	uasort()
Sort array by keys	ksort()	krsort()	uksort()

10.3.1 Natural order sorting

PHP's built-in sort functions correctly sort strings and numbers, but they don't correctly sort strings that contain numbers. For example, if you have the filenames ex10.php, ex5.php, and ex1.php, the normal sort functions will rearrange them in this order: ex1.php, ex10.php, ex5.php. To correctly sort strings that contain numbers, use the natsort() and natcasesort() functions:

```
$output = natsort(input);
$output = natcasesort(input);
```

10.3.2 Sorting Multiple Arrays at Once

The array_multisort() function sorts multiple indexed arrays at once

```
array_multisort(array1 [, array2, ... ]);
```

Pass it a series of arrays and sorting orders (identified by the SORT_ASC or SORT_DESC constants), and it reorders the elements of all the arrays, assigning new indices. It is similar to a join operation on a relational database

Imagine that you have a lot of people, and several pieces of data on each person

```
$names = array("Tom", "Dick", "Harriet", "Brenda", "Joe");
$ages = array(25, 35, 29, 35, 35);
$zips = array(80522, '02140', 90210, 64141, 80522);
```

The first element of each array represents a single record—all the information known about Tom. Similarly, the second element constitutes another record—all the information known about Dick. The array_multisort() function reorders the elements of the arrays, preserving the records. That is, if "Dick" ends up first in the $names array after the sort, the rest of Dick's information will be first in the other arrays too. (Note that we needed to quote Dick's zip code to prevent it from being interpreted as an octal constant.)

Here's how to sort the records first ascending by age, then descending by zip code:

```
array_multisort($ages, SORT_ASC, $zips, SORT_DESC, $names,
↪  SORT_ASC);
```

We need to include $names in the function call to ensure that Dick's name stays with his age and zip code. Printing out the data shows the result of the sort:

```
for ($i = 0; $i < count($names); $i++) {
    echo "{$names[$i]}, {$ages[$i]}, {$zips[$i]}\n";
}
Tom, 25, 80522
Harriet, 29, 90210
Joe, 35, 80522
Brenda, 35, 64141
Dick, 35, 02140
```

10.3.3 Reversing arrays

The array_reverse() function reverses the internal order of elements in an array:

```
$reversed = array_reverse(array);
```

Numeric keys are renumbered starting at 0, while string indices are unaffected.

The array_flip() function returns an array that reverses the order of each original element's key-value pair:

```
$flipped = array_flip(array);
```

That is, for each element of the array whose value is a valid key, the element's value becomes its key and the element's key becomes its value

Elements whose original values are neither strings nor integers are left alone in the resulting array. The new array lets you discover the key in the original array given its value, but this technique works effectively only when the original array has unique values.

10.3.4 Randomizing Order

To traverse the elements in an array in random order, use the shuffle() function. It replaces all existing keys—string or numeric—with consecutive integers starting at 0

Obviously, the order after your shuffle() may not be the same as the sample output here due to the random nature of the function. Unless you are interested in getting multiple random elements from an array without repeating any specific item, using the rand() function to pick an index is more efficient

10.3.5 Calculating the Sum of an Array

The array_sum() function adds up the values in an indexed or associative array:

10.3.6 Calculating the Difference Between Two Arrays

The array_diff() function identifies values from one array that are not present in others:

Values are compared using the strict comparison operator ===, so 1 and "1" are considered different. The keys of the first array are preserved

10.3.7 Filtering Elements from an Array

To identify a subset of an array based on its values, use the array_filter() function:

```
$filtered = array_filter(array, callback);
```

Each value of array is passed to the function named in callback. The returned array contains only those elements of the original array for which the function returns a true value.

10.4 Objects

PHP also supports object-oriented programming (OOP). OOP promotes clean modular design, simplifies debugging and maintenance, and assists with code reuse

10.4.1 New and the arrow operator

Once a class is defined, any number of objects can be made from it with the new keyword, and the object's properties and methods can be accessed with the -> construct:

```
$ed = new Person;
$ed->name('Edison');
echo "Hello, {$ed->name}\n";
$tc = new Person;
$tc->name('Crapper');
echo "Look out below {$tc->name}\n";
```

10.4.2 is_object()

Use the is_object() function to test whether a value is an object:

```
if (is_object($x)) {

}
```

10.5 Callbacks

Callbacks are functions or object methods used by some functions, such as call_user_func(). Callbacks can also be created by the create_function() method and through closures

```php
$callback = function myCallbackFunction()
{
    echo "callback achieved";
}
call_user_func($callback);
```

10.6 Type testing

PHP provides a variety of type-checking functions to check for specific types or conditions. Here's a list of commonly used type-checking functions:

- **is_null()**: Checks if a variable is NULL.
- **is_bool()**: Checks if a variable is a boolean.
- **is_int()**:r is_integer() or is_long() - Checks if a variable is an integer.
- **is_float()**:r is_double() or is_real() - Checks if a variable is a float.
- **is_string()**: Checks if a variable is a string.
- **is_array()**: Checks if a variable is an array.
- **is_object()**: Checks if a variable is an object.
- **is_resource()**: Checks if a variable is a resource.
- **is_scalar()**: Checks if a variable is a scalar (int, float, string, or bool).
- **is_callable()**: Checks if a variable is callable (e.g., function or method).
- **is_iterable()**: Checks if a variable is iterable (array or object implementing Traversable).
- **is_numeric()**: Checks if a variable is numeric (int, float, or a numeric string).
- **is_countable()**: Checks if a variable is countable (array or an object implementing Countable).

10.7 gettype()

The gettype() function in PHP is used to retrieve the data type of a given variable. It returns a string describing the variable's type, which can be helpful when debugging or dynamically handling different types of data.

Returns

- **”boolean”**: for boolean values (true or false)
- **”integer”**: for integer numbers
- **”double”**: for floating-point numbers (also referred to as ”float” or ”real number” in other languages)
- **”string”**: for string values
- **”array”**: for arrays
- **”object”**: for objects
- **”resource”**: for resources (such as file or database connections)
- **”NULL”**: for null values
- **”unknown type”**: if the type is unknown or not recognized

Typing

In PHP, typing is traditionally dynamic and weak, allowing variables to hold any type of data and automatically converting between types when necessary. However, modern versions of PHP (especially PHP 7 and beyond) offer features to enforce types more strictly, including type declarations, type hints, and the declare(strict_types=1); directive. Here's how typing in PHP works and how to enforce types:

11.1 Dynamic and Weak Typing in PHP

PHP's default behavior allows you to assign different types of values to the same variable without declaring its type:

```php
$var = "Hello";

$var = 123;
```

PHP also performs type coercion automatically, converting types when necessary. For example:

```php
$result = "10" + 5;  // "10" is coerced to integer 10, result is
↪  15
```

While this flexibility is convenient, it can lead to unexpected results and errors in complex codebases.

11.2 Enforcing Types in PHP

To gain more control over types, PHP provides several features for enforcing type constraints. Here's a breakdown of the main methods:

Since PHP 7, you can specify the expected type for function arguments and return types. This helps prevent incorrect types from being passed to functions.

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

echo add(5, 10);      // Works, outputs: 15
echo add("5", "10");  // Also works due to type coercion,
↪  outputs: 15
echo add("hello", 5); // Causes a TypeError in strict mode
```

11.2.1 Nullable types

By prefixing a type with ?, you allow the value to be either of that type or NULL. For example, ?int allows an int or NULL.

11.2.2 Strict typing

To strictly enforce types and disable type coercion, you can enable strict mode with declare(strict_types=1); at the top of a PHP file. This ensures PHP will throw a TypeError if a function receives a type it doesn't explicitly allow.

```php
declare(strict_types=1);

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

echo add(5, 10);          // Works
echo add("5", "10");      // Causes a TypeError in strict mode
```

Note: declare(strict_types=1); only applies to scalar type declarations for function arguments and return types. It does not enforce strict typing on variable assignments within the same file.

11.2.3 Union types

PHP 8.0 introduced union types, which allow you to specify that a variable can accept multiple types.

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

11.2.4 Mixed and Void Types

Mixed is a special type introduced in PHP 8 that indicates a variable or return value can be of any type.

```php
function example(mixed $input): mixed {
    return $input;
}
```

Void is used as a return type to indicate a function does not return a value.

```php
function logMessage(string $message): void {
    echo $message;
}
```

11.3 Type casting

PHP supports the following C-style type casts:

- **(int) or (integer):** Converts a value to an integer.
- **(bool) or (boolean):** Converts a value to a boolean.
- **(float) or (double) or (real):** Converts a value to a float.
- **(string):** Converts a value to a string.
- **(array):** Converts a value to an array.
- **(object):** Converts a value to an object.
- **(unset):** Converts a variable to NULL.

11.4 Implicit casting

Many operators have expectations of their operands—for instance, binary math operators typically require both operands to be of the same type. PHP's variables can store integers, floating-point numbers, strings, and more, and to keep as much of the type details away from the programmer as possible, PHP converts values from one type to another as necessary.

The conversion of a value from one type to another is called casting. This kind of implicit casting is called type juggling in PHP. The rules for the type juggling done by arithmetic operators are shown below

Type of first operand	Type of second operand	Conversion performed
int	float	The int is converted to a float.
int	String	The string is converted to a number; if the value after conversion is a floating-point number, the integer is converted to a floating-point number.
float	String	The string is converted to a floating-point number.

Some other operators have different expectations of their operands, and thus have different rules. For example, the string concatenation operator converts both operands to strings before concatenating them:

```
1  3 . 2.74 // gives the string 32.74
```

You can use a string anywhere PHP expects a number. The string is presumed to start with an integer or floating-point number. If no number is found at the start of the string, the numeric value of that string is 0. If the string contains a period (.) or upper- or lowercase e, evaluating it numerically produces a floating-point number.

```
"9 Lives" - 1; // 8 (int)
"3.14 Pies" * 2; // 6.28 (float)
"9 Lives." - 1; // 8 (float)
"1E3 Points of Light" + 1; // 1001 (float)
```

Garbage collection

PHP uses reference counting and copy-on-write to manage memory. Copy-on-write ensures that memory isn't wasted when you copy values between variables, and reference counting ensures that memory is returned to the operating system when it is no longer needed.

To understand memory management in PHP, you must first understand the idea of a symbol table. There are two parts to a variable—its name (e.g., $name), and its value (e.g., "Fred"). A symbol table is an array that maps variable names to the positions of their values in memory

When you copy a value from one variable to another, PHP doesn't get more memory for a copy of the value. Instead, it updates the symbol table to indicate that "both of these variables are names for the same chunk of memory." So the following code doesn't actually create a new array:

```
1 $worker = array("Fred", 35, "Wilma");
2 $other = $worker; // array isn't copied
```

If you subsequently modify either copy, PHP allocates the required memory and makes the copy:

```
1 $worker[1] = 36; // array is copied, value changed
```

By delaying the allocation and copying, PHP saves time and memory in a lot of situations. This is copy-on-write.

Each value pointed to by a symbol table has a reference count, a number that represents the number of ways there are to get to that piece of memory. After the initial assignment of the array to $worker and $worker to $other, the array pointed to by the symbol table entries for $worker and $other has a reference count of 2.1 In other words, that memory can be reached two ways: through $worker or $other. But after $worker[1] is changed, PHP creates a new array for $worker, and the reference count of each of the arrays is only 1.

When a variable goes out of scope, such as function parameters and local variables do at the end of a function, the reference count of its value is decreased by one. When a variable is assigned a value in a different area of memory, the reference count of the old value is decreased by one. When the reference count of a value reaches 0, its memory is released. This is reference counting.

12.1 isset(), unset()

Reference counting is the preferred way to manage memory. Keep variables local to functions, pass in values that the functions need to work on, and let reference counting take care of the memory management. If you do insist on trying to get a little more information or control over freeing a variable's value, use the isset() and unset() functions

To see if a variable has been set to something—even the empty string—use isset():

```php
$s1 = isset($name);

$name = "Fred";
$s2 = isset($name);
```

Use unset() to remove a variable's value:

```php
$name = "Fred";
unset($name);
```

Operators, precedence, associativity

The table below summarizes the operators in PHP, many of which were borrowed from C and Perl. The column labeled "P" gives the operator's precedence; the operators are listed in precedence order, from highest to lowest. The column labeled "A" gives the operator's associativity, which can be L (left-to-right), R (right-to-left), or N (nonassociative).

P	A	Operator	Operation
21	N	`clone, new`	Create new object
20	L	`[`	Array subscript
19	R	`~`	Bitwise NOT
19	R	`++, --`	Increment, Decrement
19	R	`(int), (bool), (float),`	Cast
19	R	`(string), (array), (object)`	Cast
19	R	`(unset)`	Cast
19	R	`@`	Inhibit errors
18	N	`instanceof`	Type testing
17	R	`!`	Logical NOT
16	L	`*, /, %`	Multiplication, Division, Modulus
15	L	`+, -, .`	Addition, Subtraction, String concatenation
14	L	`<<, >>`	Bitwise shift left, Bitwise shift right
13	N	`<, <=, >, >=`	Comparisons
12	N	`==, !=, <>`	Value equality, Inequality
12	N	`===, !==`	Type and value equality/inequality
11	L	`&`	Bitwise AND
10	L	`^`	Bitwise XOR
9	L	`|`	Bitwise OR
8	L	`&&`	Logical AND
7	L	`||`	Logical OR
6	L	`?:`	Conditional operator
5	L	`=, +=, -=`	Assignment with operation
5	L	`*=, /=, .=`	Assignment with operation
5	L	`%=, &=, |=`	Assignment with operation
5	L	`^=, ~=, <<=, >>=`	Assignment with operation
4	L	`and`	Logical AND
3	L	`xor`	Logical XOR
2	L	or	Logical OR
1	L	,	List separator

13.1 String concatenation operator

Manipulating strings is such a core part of PHP applications that PHP has a separate string concatenation operator (.). The concatenation operator appends the righthand operand to the lefthand operand and returns the resulting string. Operands are first converted to strings, if necessary. For example:

```
$n = 5;
$s = 'There were ' . $n . ' ducks.';

```

The concatenation operator is highly efficient, because so much of PHP boils down to string concatenation.

13.2 Auto-increment and Auto-decrement Operators

In programming, one of the most common operations is to increase or decrease the value of a variable by one. The unary auto-increment (++) and auto-decrement (--) operators provide shortcuts for these common operations. These operators are unique in that they work only on variables; the operators change their operands' values and return a value.

Operator	Name	Value returned	Effect on $var
$var++	Post-increment	$var	Incremented
++$var	Pre-increment	$var + 1	Incremented
$var--	Post-decrement	$var	Decremented
--$var	Pre-decrement	$var - 1	Decremented

These operators can be applied to strings as well as numbers. Incrementing an alphabetic character turns it into the next letter in the alphabet. As illustrated in Table 2-6, incrementing "z" or "Z" wraps it back to "a" or "A" and increments the previous character by one (or inserts a new "a" or "A" if at the first character of the string), as though the characters were in a base-26 number system.

Incrementing this	Gives this
"a"	"b"
"z"	"aa"
"spaz"	"spba"
"K9"	"L0"
"42"	"43"

13.3 Comparison Operators

Operands to the comparison operators can be both numeric, both string, or one numeric and one string. The operators check for truthfulness in slightly different ways based on the types and values of the operands, either using strictly numeric comparisons or using lexicographic (textual) comparisons. Table 2-7 outlines when each type of check is used.

First operand	Second operand	Comparison
Number	Number	Numeric
String that is entirely numeric	String that is entirely numeric	Numeric
String that is entirely numeric	Number	Numeric
String that is entirely numeric	String that is not entirely numeric	Numeric
String that is not entirely numeric	Number	Lexicographic
String that is not entirely numeric	String that is not entirely numeric	Lexicographic

One important thing to note is that two numeric strings are compared as if they were numbers. If you have two strings that consist entirely of numeric characters and you need to compare them lexicographically, use the strcmp() function.

The comparision operators are:

- **Equality (==):** If both operands are equal, this operator returns true; otherwise, it returns false.
- **Identity (===):** If both operands are equal and are of the same type, this operator returns true; otherwise, it returns false. Note that this operator does not do implicit type casting. This operator is useful when you don't know if the values you're comparing are of the same type. Simple comparison may involve value conversion. For instance, the strings "0.0" and "0" are not equal. The == operator says they are, but === says they are not.
- **Inequality (!= or <>):** If both operands are not equal, this operator returns true; otherwise, it returns false.
- **Not identical (!==):** If both operands are not equal, or they are not of the same type, this operator returns true; otherwise, it returns false.
- **Greater than (>):** If the lefthand operand is greater than the righthand operand, this operator returns true; otherwise, it returns false.
- **Greater than or equal to (>=):** If the lefthand operand is greater than or equal to the righthand operand, this operator returns true; otherwise, it returns false.
- **Less than (<):** If the lefthand operand is less than the righthand operand, this operator returns true; otherwise, it returns false.
- **Less than or equal to (<=):** If the lefthand operand is less than or equal to the righthand operand, this operator returns true; otherwise, it returns false.

13.4 Logical Operators

- **Logical AND (&&, and)**: The result of the logical AND operation is true if and only if both operands are true; otherwise, it is false. If the value of the first operand is false, the logical AND operator knows that the resulting value must also be false, so the righthand operand is never evaluated. This process is called short-circuiting, and a common PHP idiom uses it to ensure that a piece of code is evaluated only if something is true. For example, you might connect to a database only if some flag is not false:

  The && and and operators differ only in their precedence.

- **Logical OR (||, or)**: The result of the logical OR operation is true if either operand is true; otherwise, the result is false. Like the logical AND operator, the logical OR operator is shortcircuited. If the lefthand operator is true, the result of the operator must be true, so the righthand operator is never evaluated. A common PHP idiom uses this to trigger an error condition if something goes wrong

  The || and or operators differ only in their precedence.

- **Logical XOR (xor)**: The result of the logical XOR operation is true if either operand, but not both, is true; otherwise, it is false.

- **Logical negation (!)**: The logical negation operator returns the Boolean value true if the operand evaluates to false, and false if the operand evaluates to true.

13.5 More on casting operators

Although PHP is a weakly typed language, there are occasions when it's useful to consider a value as a specific type. The casting operators, (int), (float), (string), (bool), (array), (object), and (unset), allow you to force a value into a particular type

The table below lists the casting operators, synonymous operands, and the type to which the operator changes the value

Operator	Synonymous operators	Changes type to
(int)	(integer)	Integer
(bool)	(boolean)	Boolean
(float)	(double),	(real) Floating point
(string)		String
(array)		Array
(object)		Object
(unset)		NULL

Casting affects the way other operators interpret a value rather than changing the value in a variable

Not every cast is useful. Casting an array to a numeric type gives 1, and casting an array to a string gives "Array" (seeing this in your output is a sure sign that you've printed a variable that contains an array).

Casting an object to an array builds an array of the properties, thus mapping property names to values:

```
class Person
{
    var $name = "Fred";
    var $age = 35;
}
$o = new Person;
$a = (array) $o;
print_r($a);

Array (
    [name] => Fred
    [age] => 35
)
```

You can cast an array to an object to build an object whose properties correspond to the array's keys and values. For example:

```
$a = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");
$o = (object) $a;
echo $o->name;
Fred
```

Keys that are not valid identifiers are invalid property names and are inaccessible when an array is cast to an object, but are restored when the object is cast back to an array.

13.6 Notes about assignment in php

Because all operators are required to return a value, the assignment operator returns the value assigned to the variable. For example, the expression $a = 5 not only assigns 5 to $a, but also behaves as the value 5 if used in a larger expression. Consider the following expressions

```
$a = 5;
$b = 10;
$c = ($a = $b);
```

The expression $a = $b is evaluated first, because of the parentheses. Now, both $a and $b have the same value, 10. Finally, $c is assigned the result of the expression $a = $b, which is the value assigned to the lefthand operand (in this case, $a). When the full expression is done evaluating, all three variables contain the same value: 10.

13.7 Miscellaneous Operators

The remaining PHP operators are for error suppression, executing an external command, and selecting values:

- **Error suppression (@)**: Some operators or functions can generate error messages. The error suppression operator is used to prevent these messages from being created.
- **Execution (‘..‘)**: The backtick operator executes the string contained between the backticks as a shell command and returns the output. For example:

```
$listing = `ls -ls /tmp`;
echo $listing;
```

- **Conditional (? :)**: The conditional operator is, depending on the code you look at, either the most overused or most underused operator. It is the only ternary (three-operand) operator and is therefore sometimes just called the ternary operator. The conditional operator evaluates the expression before the ?. If the expression is true, the operator returns the value of the expression between the ? and :; otherwise, the operator returns the value of the expression after the :. For instance:

```
<a href="<?= $url; ?>"><?= $linktext ? $linktext : $url;
↪  ?></a>
```

  If text for the link $url is present in the variable $linktext, it is used as the text for the link; otherwise, the URL itself is displayed.

- **Type (instanceof)**: The instanceof operator tests whether a variable is an instantiated object of a given class or implements an interface

```php
$a = new Foo;
$isAFoo = $a instanceof Foo; // true
$isABar = $a instanceof Bar; // false
```

Flow control

14.1 if

```
if (expression) {
    ...
} else if (expression) {
    ...
} else {
    ...
}
```

14.1.1 Shorthand ifs

Php supports shorthand if statements for single line blocks

```
if (expression)
    ...
else if (expression)
    ...
else
    ...
```

14.1.2 endif

We can also use colons to start the blocks, then end with *endif*

```
if (expression):
    ...
else:
    ...
endif;
```

```
<? if ($user_validated) :?>
 <table>
 <tr>
 <td>First Name:</td><td>Sophia</td>
 </tr>
 <tr>
 <td>Last Name:</td><td>Lee</td>
 </tr>
 </table>
<? else: ?>
 Please log in.
<? endif ?>
```

14.2 switch

A switch statement is given an expression and compares its value to all cases in the switch; all statements in a matching case are executed, up to the first break keyword it finds. If none match, and a default is given, all statements following the default keyword are executed, up to the first break keyword encountered

```
switch (expression) {

    case value:
        ...
        break;
    case value:
        ...
        break;
    default:
        ...
        break;

}
```

14.3 Fall throughs

Because statements are executed from the matching case label to the next break keyword, you can combine several cases in a fall-through. In the following example, “yes” is printed when $name is equal to sylvie or bruno

```
switch ($name) {
 case 'sylvie': // fall-through
 case 'bruno':
     print("yes");
     break;
 default:
     print("no");
     break;
}
```

14.4 While loops

The simplest form of loop is the while statement:

```
while (expression) statement;

while (expression) {
    ...
}
```

If the expression evaluates to true, the statement is executed and then the expression is re-evaluated (if it is still true, the body of the loop is executed again, and so on). The loop exits when the expression is no longer true, i.e., evaluates to false.

14.4.1 endwhile

The alternative syntax for while has this structure:

```
while (expr):
    statement;
    more statements;
endwhile;
```

14.4.2 break n

you can put a number after the break keyword indicating how many levels of loop structures to break out of. In this way, a statement buried deep in nested loops can break out of the outermost loop. F

14.5 do while

We can also create do while loops in php

```
do {

} while (expression);
```

14.6 for loops

```
for (start; condition; increment) { statement(s); }
```

14.7 endfor

The alternative syntax of a for statement is:

```
for (expr1; expr2; expr3):
 statement;
 ...
endfor;
```

You can also leave an expression empty, signaling that nothing should be done for that phase. In the most degenerate form, the for statement becomes an infinite loop. You probably don't want to run this example, as it never stops printing:

```
for (;;) {
    ...
}
```

14.8 foreach

The foreach statement allows you to iterate over elements in an array

```
foreach ($array as $current) {
 // ...
}
```

14.8.1 endforeach

The alternate syntax is:

```
foreach ($array as $current):
 // ...
endforeach;
```

To loop over an array, accessing both key and value, use:

```
foreach ($array as $key => $value) {
 // ...
}
```

The alternate syntax is:

```
foreach ($array as $key => $value):
 // ...
endforeach;
```

14.9 try...catch

The try...catch construct is not so much a flow-control structure as it is a more graceful way to handle system errors. For example, if you want to ensure that your web application has a valid connection to a database before continuing, you could write code like this:

```
try {
    $dbhandle = new PDO('mysql:host=localhost; dbname=library',
    ↪  $username, $pwd);
    doDB_Work($dbhandle); // call function on gaining a
    ↪  connection
    $dbhandle = null; // release handle when done
}
catch (PDOException $error) {
    print "Error!: " . $error->getMessage() . "<br/>";
    die();
}
```

Here the connection is attempted with the try portion of the construct and if there are any errors with it, the flow of the code automatically falls into the catch portion, where the PDOException class is instantiated into the $error variable. It can then be displayed on the screen and the code can "gracefully" fail, rather than making an abrupt end. You can even redirect to another connection attempt to an alternate database, or respond to the error any other way you wish within the catch portion.

14.10 declare

The declare statement allows you to specify execution directives for a block of code. The structure of a declare statement is:

Currently, there are only two declare forms: the ticks and encoding directives. You can specify how frequently (measured roughly in number of code statements) a tick function registered when register_tick_function() is called using the ticks directive. For example:

```
register_tick_function("someFunction");
declare(ticks = 3) {
    for($i = 0; $i < 10; $i++) {
        // do something
    }
}
```

In this code, someFunction() is called after every third statement within the block is executed.

You can specify a PHP script's output encoding using the encoding directive. For example:

```
declare(encoding = "UTF-8");
```

This form of the declare statement is ignored unless you compile PHP with the --enable-zend-multibyte option.

14.11 exit

The exit statement takes an optional value. If this is a number, it is the exit status of the process. If it is a string, the value is printed before the process terminates. The function die() is an alias for this form of the exit statement

14.12 goto

The goto statement allows execution to "jump" to another place in the program. You specify execution points by adding a label, which is an identifier followed by a colon (:). You then jump to the label from another location in the script via the goto statemen

```
for ($i = 0; $i < $count; $i++) {
    // oops, found an error
    if ($error) {
        goto cleanup;
    }
}
cleanup:
// do some cleanup
```

You can only goto a label within the same scope as the goto statement itself, and you can't jump into a loop or switch. Generally, anywhere you might use a goto (or multilevel break statement, for that matter), you can rewrite the code to be cleaner without it

14.13 Including Code

PHP provides two constructs to load code and HTML from another module: require and include. Both load a file as the PHP script runs, work in conditionals and loops, and complain if the file being loaded cannot be found. The main difference is that attempting to require a nonexistent file is a fatal error, while attempting to include such a file produces a warning but does not stop script execution.

A common use of include is to separate page-specific content from general site design. Common elements such as headers and footers go in separate HTML files, and each page then looks like:

```
<?php include "header.html"; ?>
content
<?php include "footer.html"; ?>
```

We use include because it allows PHP to continue to process the page even if there's an error in the site design file(s). The require construct is less forgiving and is more suited to loading code libraries, where the page cannot be displayed if the libraries do not load. For example:

```
require "codelib.php";
mysub(); // defined in codelib.php
```

A marginally more efficient way to handle headers and footers is to load a single file and then call functions to generate the standardized site elements

```
<?php require "design.php";
header(); ?>
content
<?php footer();
```

If PHP cannot parse some part of a file added by include or require, a warning is printed and execution continues. You can silence the warning by prepending the call with the silence operator (@)—for example, @include.

If the allow_url_fopen option is enabled through PHP's configuration file, php.ini, you can include files from a remote site by providing a URL instead of a simple local path:

```
include "http://www.example.com/codelib.php";
```

If the filename begins with http:// or ftp://, the file is retrieved from a remote site and loaded.

Files included with include and require can be arbitrarily named. Common extensions are .php, .php5, and .html. Note that remotely fetching a file that ends in .php from a web server that has PHP enabled fetches the output of that PHP script—it executes the PHP code in that file

If a program uses include or require to include the same file twice (mistakenly done in a loop, for example), the file is loaded and the code is run, or the HTML is printed twice. This can result in errors about the redefinition of functions, or multiple copies of headers or HTML being sent. To prevent these errors from occurring, use the include_once and require_once constructs. They behave the same as include and require the first time a file is loaded, but quietly ignore subsequent attempts to load the same file. For example, many page elements, each stored in separate files, need to know the current user's preferences. The element libraries should load the user preferences library with require_once. The page designer can then include a page element without worrying about whether the user preference code has already been loaded.

Code in an included file is imported at the scope that is in effect where the include statement is found, so the included code can see and alter your code's variables. This can be useful—for instance, a user-tracking library might store the current user's name in the global $user variable:

```
// main page
include "userprefs.php";
echo "Hello, {$user}.";
```

The ability of libraries to see and change your variables can also be a problem. You have to know every global variable used by a library to ensure that you don't accidentally try to use one of them for your own purposes, thereby overwriting the library's value and disrupting how it works.

If the include or require construct is in a function, the variables in the included file become function-scope variables for that function.

Because include and require are keywords, not real statements, you must always enclose them in curly braces in conditional and loop statements

```
for ($i = 0; $i < 10; $i++) {
    include "repeated_element.html";
}
```

Use the get_included_files() function to learn which files your script has included or required. It returns an array containing the full system path filenames of each included or required file. Files that did not parse are not included in this array.

Functions

To define a function, use the following syntax:

```
function [&] function_name([parameter[, ...]])
{
    statement list
}
```

For example,

```
function strcat($left, $right)
{
    return $left . $right;
}
```

Once the function is defined, you can use it anywhere on the page

15.1 Nesting functions

You can nest function declarations, but with limited effect. Nested declarations do not limit the visibility of the inner-defined function, which may be called from anywhere in your program. The inner function does not automatically get the outer function's arguments. And, finally, the inner function cannot be called until the outer function has been called, and also cannot be called from code parsed after the outer function:

```
function outer ($a)
{
    function inner ($b)
    {
        echo "there $b";
    }
    echo "$a, hello ";
}
// outputs "well, hello there reader"
outer("well");
inner("reader");
```

15.2 Variable scope

If you don't use functions, any variable you create can be used anywhere in a page. With functions, this is not always true. Functions keep their own sets of variables that are distinct from those of the page and of other functions.

The variables defined in a function, including its parameters, are not accessible outside the function, and, by default, variables defined outside a function are not accessible inside the function

Only functions can provide local scope. Unlike in other languages, in PHP you can't create a variable whose scope is a loop, conditional branch, or other type of block.

15.3 Global variables

Variables declared outside a function are global. That is, they can be accessed from any part of the program. However, by default, they are not available inside functions. To allow a function to access a global variable, you can use the global keyword inside the function to declare the variable within the function. Here's how we can rewrite the updateCounter() function to allow it to access the global $counter variable

```
function updateCounter()
{
    global $counter;
    $counter++;
}
$counter = 10;
updateCounter();
echo $counter;
```

15.4 const variables in the global scope

PHP doesn't allow const constants to be declared in the global scope and then referenced with global. Only variables (declared with $) can be declared in the global scope and referenced with global inside functions. Constants declared with const are fixed and cannot be changed, nor do they need the global keyword within functions.

15.5 Const function parameter

PHP does not support making function parameters const. Parameters are inherently mutable within the function scope, and there is no direct way to enforce immutability (constancy) on them in PHP.

15.6 More on static variables

Like C, PHP supports declaring function variables static. A static variable retains its value between all calls to the function and is initialized during a script's execution only the first time the function is called. Use the static keyword at the variable's first use to declare a function variable static. Typically, the first use of a static variable is to assign an initial value:

```
static var [= value][, ... ];
```

15.7 Default arguments

To specify a default parameter, assign the parameter value in the function declaration. The value assigned to a parameter as a default value cannot be a complex expression; it can only be a scalar value:

```
function getPreferences($whichPreference = 'all')
{


    // otherwise, get the specific preference requested...
}
```

A function may have any number of parameters with default values. However, they must be listed after all parameters that do not have default values.

15.8 Variadic functions

To declare a function with a variable number of arguments, leave out the parameter block entirely

PHP provides three functions you can use in the function to retrieve the parameters passed to it. func_get_args() returns an array of all parameters provided to the function; func_num_args() returns the number of parameters provided to the function; and func_get_arg() returns a specific argument from the parameters

```
function bprint() {
    foreach($args = func_get_args() as $item) {
        print($item);
    }
    echo "<br/>";
    echo "Printed " . func_num_args() . " items";
    echo "<br/>";
    echo "Second argument was " . func_get_arg(1);
}
```

15.9 Variadic functions with elipsis

PHP also supports a ”variable-length argument list” using the ... (splat operator) syntax, allowing functions to accept a flexible number of arguments.

```
function f($a, $b, ...$c) {
    echo $a . $b;
    foreach($c as $arg) {
        echo $arg . " ";
    }
}
f(1,2,3,4,5,6)
```

In this example, passing more than two arguments to f stores them in the array c.

15.10 Returning references

By default, values are copied out of the function. To return a value by reference, both declare the function with an & before its name and when assigning the returned value to a variable:

```
$names = array("Fred", "Barney", "Wilma", "Betty");
function &findOne($n) {
    global $names;
    return $names[$n];
}
$person =& findOne(1); // Barney
$person = "Barnetta";
```

n this code, the findOne() function returns an alias for $names[1], instead of a copy of its value. Because we assign by reference, $person is an alias for $names[1], and the second assignment changes the value in $names[1].

This technique is sometimes used to return large string or array values efficiently from a function. However, PHP implements copy-on-write for variable values, meaning that returning a reference from a function is typically unnecessary. Returning a reference to a value is slower than returning the value itself.

15.11 Variable function calling

Consider the following code

```
function f($a) {
    echo $a . " ";
}
$a="f";
$a();
```

The line $a = ”f”; and the subsequent call $a(1,2,3,4); demonstrate variable function calling. By assigning the function name ”f” to the variable $a, you can invoke the function f through $a as if it were the function’s name. In PHP, variables containing function names can be used to call those functions.

15.12 Anonymous Functions (Closures)

Some PHP functions use a function you provide them with to do part of their work. For example, the usort() function uses a function you create and pass to it as a parameter to determine the sort order of the items in an array.

Although you can define a function for such purposes, as shown previously, these functions tend to be localized and temporary. To reflect the transient nature of the callback, create and use an anonymous function (also known as a closure).

You can create an anonymous function using the normal function definition syntax, but assign it to a variable or pass it directly

```
$array = array("really long string here, boy", "this", "middling
↪  length", "larger");
usort($array, function($a, $b) {
    return strlen($a) - strlen($b);
});
print_r($array);
```

The array is sorted by usort() using the anonymous function, in order of string length.

Anonymous functions can use the variables defined in their enclosing scope using the use syntax. For example

```
$array = array("really long string here, boy", "this", "middling
↪  length", "larger");
$sortOption = 'random';
usort($array, function($a, $b) use ($sortOption)
{
    if ($sortOption == 'random') {
        // sort randomly by returning (-1, 0, 1) at random
        return rand(0, 2) - 1;
    }
    else {
        return strlen($a) - strlen($b);
    }
});
print_r($array);
```

Note that incorporating variables from the enclosing scope is not the same as using global variables—global variables are always in the global scope, while incorporating variables allows a closure to use the variables defined in the enclosing scope. Also note that this is not necessarily the same as the scope in which the closure is called.

15.13 Arrow functions

In PHP 7.4 and later, you can also use arrow functions, which are shorter lambdas with implicit return statements and single-line syntax

```
$add = fn($a, $b) => $a + $b;

echo $add(3, 4);  // Outputs: 7
```

Arrow functions automatically capture variables from the parent scope, so you don't need use

Iterators

Every PHP array keeps track of the current element you're working with; the pointer to the current element is known as the iterator. PHP has functions to set, move, and reset this iterator. The iterator functions are:

- **current()**: Returns the element currently pointed at by the iterator
- **reset()**: Moves the iterator to the first element in the array and returns it
- **next()**: Moves the iterator to the next element in the array and returns it
- **prev()**: Moves the iterator to the previous element in the array and returns it
- **end()**: Moves the iterator to the last element in the array and returns it
- **key()**: Returns the key of the current element

```
1  reset($a); // Move a's iterator to start and put value in first
2
3  echo current($a); // Get current value (still at the start)
4  echo next($a); // One past the start
5  echo prev($a); // Back to start
6
7  $last = end($a); // Move a's iterator to end dand put in last
```

Sets

Arrays let you implement the basic operations of set theory: union, intersection, and difference. Each set is represented by an array, and various PHP functions implement the set operations. The values in the set are the values in the array—the keys are not used, but they are generally preserved by the operations.

The union of two sets is all the elements from both sets with duplicates removed. The array_merge() and array_unique() functions let you calculate the union. Here's how to find the union of two arrays:

```
function arrayUnion($a, $b)
{
 $union = array_merge($a, $b); // duplicates may still exist
 $union = array_unique($union);
 return $union;
}
$first = array(1, "two", 3);
$second = array("two", "three", "four");
$union = arrayUnion($first, $second);
print_r($union);
```

The intersection of two sets is the set of elements they have in common. PHP's built-in array_intersect() function takes any number of arrays as arguments and returns an array of those values that exist in each. If multiple keys have the same value, the first key with that value is preserved.

Stacks and queues

Although not as common in PHP programs as in other programs, one fairly common data type is the last-in first-out (LIFO) stack. We can create stacks using a pair of PHP functions, array_push() and array_pop(). The array_push() function is identical to an assignment to $array[]. We use array_push() because it accentuates the fact that we're working with stacks, and the parallelism with array_pop() makes our code easier to read. There are also array_shift() and array_unshift() functions for treating an array like a queue

Iterator Interface

Using the foreach construct, you can iterate not only over arrays, but also over instances of classes that implement the Iterator interface. To implement the Iterator interface, you must implement five methods on your class:

- **current()**: Returns the element currently pointed at by the iterator
- **key()**: Returns the key for the element currently pointed at by the iterator
- **next()**: Moves the iterator to the next element in the object and returns it
- **rewind()**: Moves the iterator to the first element in the array
- **valid()**: Returns true if the iterator currently points at a valid element, false otherwise

Object-oriented programming

20.1 Creating an object

We can create objects with the keyword *new*. Assuming that a Person class has been defined, here's how to create a Person object:

```
$rasmus = new Person;
```

Do not quote the class name, or you'll get a compilation error:

Some classes permit you to pass arguments to the new call. The class's documentation should say whether it accepts arguments. If it does, you'll create objects like this:

```
$object = new Person("Fred", 35);
```

The class name does not have to be hardcoded into your program. You can supply the class name through a variable

```
$class = "Person";
$object = new $class;
// is equivalent to
$object = new Person;
```

Specifying a class that doesn't exist causes a runtime error.

Once created, objects are passed by reference—that is, instead of copying around the entire object itself (a time- and memory-consuming endeavor), a reference to the object is passed around instead

20.1.1 Clone

If you want to create a true copy of an object, you use the clone operator:

```
$f = new Person("Fred", 35);
$b = clone $f; // make a copy
$b->setName("Barney");// change the copy
printf("%s and %s are best friends.\n", $b->getName(),
 ↪  $f->getName());
Fred and Barney are best friends.
```

When you use the clone operator to create a copy of an object and that class declares the ___clone() method, that method is called on the new object immediately after it's cloned. You might use this in cases where an object holds external resources (such as file handles) to create new resources, rather than copying the existing ones.

20.2 Accessing Properties and Methods

Once you have an object, you can use the -> notation to access methods and properties of the object

```
$object->propertyname $object->methodname([arg, ... ])
```

A static method is one that is called on a class, not on an object. Such methods cannot access properties. The name of a static method is the class name followed by two colons and the function name. For instance, this calls the p() static method in the HTML class

```
HTML::p("Hello, world");
```

20.3 Declaring a Class

To design your program or code library in an object-oriented fashion, you'll need to define your own classes, using the class keyword. A class definition includes the class name and the properties and methods of the class. Class names are case-insensitive and must conform to the rules for PHP identifiers. The class name stdClass is reserved. Here's the syntax for a class definition:

```
class classname [ extends baseclass ] [ implements interfacename
↪  ,
[interfacename, ... ] ]
{
    [ use traitname, [ traitname, ... ]; ]
    [ visibility $property [ = value ]; ... ]
    [ function functionname (args) {
        // code
    }
    ...
    ]
}
```

20.3.1 Declaring Methods

A method is a function defined inside a class. Although PHP imposes no special restrictions, most methods act only on data within the object in which the method resides. Method names beginning with two underscores (___) may be used in the future by PHP (and are currently used for the object serialization methods ___sleep() and ___wakeup(), described later in this chapter, among others), so it's recommended that you do not begin your method names with this sequence

20.3.2 this

Within a method, the $this variable contains a reference to the object on which the method was called. For instance, if you call $rasmus->birthday(), inside the birth day() method, $this holds the same value as $rasmus. Methods use the $this variable to access the properties of the current object and to call other methods on that object

Here's a simple class definition of the Person class that shows the $this variable in action

```
class Person
{
    public $name = '';
    function getName()
    {
        return $this->name;
    }
    function setName($newName)
    {
        $this->name = $newName;
    }
}
```

20.3.3 static

To declare a method as a static method, use the static keyword. Inside of static methods the variable $this is not defined.

20.3.4 final

If you declare a method using the final keyword, subclasses cannot override that method

```
class Person
{
    public $name;
    final function getName()
    {
        return $this->name;
    }
}
class Child extends Person
{
    // syntax error
    function getName()
    {
        // do something
    }
```

20.3.5 access modifiers

Using access modifiers, you can change the visibility of methods. Methods that are accessible outside methods on the object should be declared public; methods on an instance that can only be called by methods within the same class should be declared private. Finally, methods declared as protected can only be called from within the object's class methods and the class methods of classes inheriting from the class. Defining the visibility of class methods is optional; if a visibility is not specified, a method is public.

20.3.6 Declaring Properties

In the previous definition of the Person class, we explicitly declared the $name property. Property declarations are optional and are simply a courtesy to whomever maintains your program. It's good PHP style to declare your properties, but you can add new properties at any time.

Here's a version of the Person class that has an undeclared $name property:

```
class Person
{
    function getName()
    {
        return $this->name;
    }
    function setName($newName)
    {
        $this->name = $newName;
    }
}
```

You can assign default values to properties, but those default values must be simple constants:

```
public $name = "J Doe"; // works
public $age = 0; // works
public $day = 60 * 60 * 24; // doesn't work
```

Using access modifiers, you can change the visibility of properties. Properties that are accessible outside the object's scope should be declared public; properties on an instance that can only be accessed by methods within the same class should be declared private. Finally, properties declared as protected can only be accessed by the object's class methods and the class methods of classes inheriting from the class

In addition to properties on instances of objects, PHP allows you to define static properties, which are variables on an object class, and can be accessed by referencing the property with the class name. For example:

```php
class Person
{
    static $global = 23;
}
$localCopy = Person::$global;
```

Inside an instance of the object class, you can also refer to the static property using the self keyword, like echo self::$global;.

If a property is accessed on an object that doesn't exist, and if the ___get() or ___set() method is defined for the object's class, that method is given an opportunity to either retrieve a value or set the value for that property.

20.3.7 Declaring Constants

Like global constants, assigned through the define() function, PHP provides a way to assign constants within a class. Like static properties, constants can be accessed directly through the class or within object methods using the self notation. Once a constant is defined, its value cannot be changed

```php
class PaymentMethod
{
    const TYPE_CREDITCARD = 0;
    const TYPE_CASH = 1;
}
echo PaymentMethod::TYPE_CREDITCARD;
```

20.4 Inheritance

To inherit the properties and methods from another class, use the extends keyword in the class definition, followed by the name of the base class:

```php
class Person
{
    public $name, $address, $age;
}
class Employee extends Person
{
    public $position, $salary;
}
```

The Employee class contains the $position and $salary properties, as well as the $name, $address, and $age properties inherited from the Person class

If a derived class has a property or method with the same name as one in its parent class, the property or method in the derived class takes precedence over the property or method in the parent class. Referencing the property returns the value of the property on the child, while referencing the method calls the method on the child.

To access an overridden method on an object's parent class, use the parent:: method() notation:

```
parent::birthday(); // call parent class's birthday() method
```

If a method might be subclassed and you want to ensure that you're calling it on the current class, use the self::method() notation:

```
self::birthday(); // call this class's birthday() method
```

To check if an object is an instance of a particular class or if it implements a particular interface, you can use the instanceof operator:

20.5 Interfaces

Interfaces provide a way for defining contracts to which a class adheres; the interface provides method prototypes and constants, and any class that implements the interface must provide implementations for all methods in the interface. Here's the syntax for an interface definition:

```
interface interfacename
{
    [ function functionname();
    ...
    ]
}
```

To declare that a class implements an interface, include the implements keyword and any number of interfaces, separated by commas:

```
interface Printable
{
    function printOutput();
}
class ImageComponent implements Printable
{
    function printOutput()
    {
        echo "Printing an image...";
    }
}
```

An interface may inherit from other interfaces (including multiple interfaces) as long as none of the interfaces it inherits from declare methods with the same name as those declared in the child interface.

20.6 Traits

Traits provide a mechanism for reusing code outside of a class hierarchy. Traits allow you to share functionality across different classes that don't (and shouldn't) share a common ancestor in a class hierarchy. Here's the syntax for a trait definition

```
trait traitname [ extends baseclass ]
{
    [ use traitname, [ traitname, ... ]; ]
    [ visibility $property [ = value ]; ... ]
    [ function functionname (args) {
        // code
    }
    ...
    ]
}
```

To declare that a class should include a trait's methods, include the use keyword and any number of traits, separated by commas:

```
trait Logger
{
    public log($logString)
    {
        $className = __CLASS__;
        echo date("Y-m-d h:i:s", time()) . ": [{$className}]
        ↪ {$logString}";
    }
}
class User
{
    use Logger;
    ...
}
```

Traits can be composed of other traits by including the use statement in the trait's declaration, followed by one or more trait names separated by commas

Traits can declare abstract methods.

If a class uses multiple traits defining the same method, PHP gives a fatal error. However, you can override this behavior by telling the compiler specifically which implementation of a given method you want to use. When defining which traits a class includes, use the insteadof keyword for each conflict:

```php
trait Command
{
    function run()
    {
        echo "Executing a command\n";
    }
}
trait Marathon
{
    function run()
    {
        echo "Running a marathon\n";
    }
}
class Person
{
    use Command, Marathon {
        Marathon::run insteadof Command;
    }
}
```

Instead of picking just one method to include, you can use the as keyword to alias a trait's method within a class including it to a different name. You must still explicitly resolve any conflicts in the included traits. For example

```php
trait Command
{
    function run()
    {
        echo "Executing a command";
    }
}
trait Marathon
{
    function run()
    {
        echo "Running a marathon";
    }
}
class Person
{
    use Command, Marathon {
        Command::run as runCommand;
        Marathon::run insteadof Command;
    }
}
$person = new Person;
$person->run();
$person->runCommand();
```

20.7 Abstract Methods

PHP also provides a mechanism for declaring that certain methods on the class must be implemented by subclasses—the implementation of those methods is not defined in the parent class. In these cases, you provide an abstract method; in addition, if a class has any methods in it defined as abstract, you must also declare the class as an abstract class:

```
abstract class Component
{
    abstract function printOutput();
}
class ImageComponent extends Component
{
    function printOutput()
    {
        echo "Pretty picture";
    }
}
```

Abstract classes cannot be instantiated. Also note that unlike some languages, you cannot provide a default implementation for abstract methods

Traits can also declare abstract methods. Classes that include a trait that defines an abstract method must implement that method:

20.8 Constructors

A constructor is a function in the class called ___construct(). Here's a constructor for the Person class:

```
class Person
{
    function __construct($name, $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
}
```

PHP does not provide for an automatic chain of constructors; that is, if you instantiate an object of a derived class, only the constructor in the derived class is automatically called. For the constructor of the parent class to be called, the constructor in the derived class must explicitly call the constructor

20.9 Destructors

When an object is destroyed, such as when the last reference to an object is removed or the end of the script is reached, its destructor is called. Because PHP automatically cleans up all resources when they fall out of scope and at the end of a script's execution, their application is limited. The destructor is a method called ___destruct():

```
class Building
{
    function __destruct()
    {
        echo "A Building is being destroyed!";
    }
}
```

20.10 Introspection

Introspection is the ability of a program to examine an object's characteristics, such as its name, parent class (if any), properties, and methods. With introspection, you can write code that operates on any class or object. You don't need to know which methods or properties are defined when you write your code; instead, you can discover that information at runtime, which makes it possible for you to write generic debuggers, serializers, profilers, etc. In this section, we look at the introspective functions provided by PHP.

20.10.1 Examining Classes

To determine whether a class exists, use the class_exists() function, which takes in a string and returns a Boolean value. Alternately, you can use the get_ declared_classes() function, which returns an array of defined classes and checks if the class name is in the returned array:

```
$doesClassExist = class_exists(classname);
$classes = get_declared_classes();
$doesClassExist = in_array(classname, $classes);
```

You can get the methods and properties that exist in a class (including those that are inherited from superclasses) using the get_class_methods() and get_class_vars() functions. These functions take a class name and return an array:

```
$methods = get_class_methods(classname);
$properties = get_class_vars(classname);
```

The class name can be a bare word, a quoted string, or a variable containing the class name

```
$class = "Person";
$methods = get_class_methods($class);
$methods = get_class_methods(Person); // same
$methods = get_class_methods("Person"); // same
```

The array returned by get_class_methods() is a simple list of method names. The associative array returned by get_class_vars() maps property names to values and also includes inherited properties

One quirk of get_class_vars() is that it returns only properties that have default values and are visible in the current scope; there's no way to discover uninitialized properties

Use get_parent_class() to find a class's parent class:

```
$superclass = get_parent_class(classname);
```

we also have the display_classes() function, which displays all currently declared classes and the methods and properties for each.

20.10.2 Examining an Object

To get the class to which an object belongs, first make sure it is an object using the is_object() function, and then get the class with the get_class() function:

```
$isObject = is_object(var);
$classname = get_class(object);
```

Before calling a method on an object, you can ensure that it exists using the method_exists() function:

```
$methodExists = method_exists(object, method);
```

Calling an undefined method triggers a runtime exception.

Just as get_class_vars() returns an array of properties for a class, get_object_vars() returns an array of properties set in an object:

```
$array = get_object_vars(object);
```

And just as get_class_vars() returns only those properties with default values, get_object_vars() returns only those properties that are set:

The get_parent_class() function accepts either an object or a class name. It returns the name of the parent class, or FALSE if there is no parent class:

Super globals

- **$_COOKIE**: Contains any cookie values passed as part of the request, where the keys of the array are the names of the cookies
- **$_GET**: Contains any parameters that are part of a GET request, where the keys of the array are the names of the form parameters
- **$_POST**: Contains any parameters that are part of a POST request, where the keys of the array are the names of the form parameters
- **$_FILES**: Contains information about any uploaded files
- **$_SERVER**: Contains useful information about the web server, as described in the next section
- **$_ENV**: Contains the values of any environment variables, where the keys of the array are the names of the environment variables
- **$_REQUEST**: The $_REQUEST array contains the elements of the $_GET, $_POST, and $_COOKIE arrays all in one array variable.

Server information

- **PHP_SELF**: The name of the current script, relative to the document root (e.g., /store/ cart.php). You should already have noted seeing this used in some of the sample code in earlier chapters. This variable is useful when creating self-referencing scripts, as we'll see later.
- **SERVER_SOFTWARE**: A string that identifies the server (e.g., "Apache/1.3.33 (Unix) mod_perl/1.26 PHP/ 5.0.4").
- **SERVER_NAME**: The hostname, DNS alias, or IP address for self-referencing URLs (e.g., www.example.com).
- **GATEWAY_INTERFACE**: The version of the CGI standard being followed (e.g., "CGI/1.1").
- **SERVER_PROTOCOL**: The name and revision of the request protocol (e.g., "HTTP/1.1").
- **SERVER_PORT**: The server port number to which the request was sent (e.g., "80").
- **REQUEST_METHOD**: The method the client used to fetch the document (e.g., "GET").
- **PATH_INFO**: Extra path elements given by the client (e.g., /list/users).
- **PATH_TRANSLATED**: The value of PATH_INFO, translated by the server into a filename (e.g., /home/httpd/ htdocs/list/users).
- **SCRIPT_NAME**: The URL path to the current page, which is useful for self-referencing scripts (e.g., / me/menu.php).
- **QUERY_STRING**: Everything after the ? in the URL (e.g., name=Fred+age=35).
- **REMOTE_HOST**: The hostname of the machine that requested this page (e.g., "dialup-192-168-0-1.example.com (http://dialup-192-168-0-1.example.com)"). If there's no DNS for the machine, this is blank and REMOTE_ADDR is the only information given.
- **REMOTE_ADDR**: A string containing the IP address of the machine that requested this page (e.g., "192.168.0.250").
- **AUTH_TYPE**: If the page is password-protected, this is the authentication method used to protect the page (e.g., "basic").
- **REMOTE_USER**: If the page is password-protected, this is the username with which the client authenticated (e.g., "fred"). Note that there's no way to find out what password was used.
- **REMOTE_IDENT**: If the server is configured to use identd (RFC 931) identification checks, this is the username fetched from the host that made the web request (e.g., "barney"). Do not use this string for authentication purposes, as it is easily spoofed.
- **CONTENT_TYPE**: The content type of the information attached to queries such as PUT and POST (e.g., "x-url-encoded")

HTML with php

23.1 Html Forms

HTML forms are used to collect user input and send it to a server for processing. They are commonly found in websites for actions like logging in, signing up, submitting feedback, or performing searches.

An HTML form is created with the <form> tag, which contains input elements like text fields, checkboxes, and submit buttons. Here's a basic example:

```
<form action="/submit-form" method="post">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" required>

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>

  <input type="submit" value="Submit">
</form>
```

- **action:** Specifies the URL where the form data should be sent when submitted.
- **method:** Defines the HTTP method to use for form submission, either "get" or "post".
  - **GET:** Sends form data as part of the URL (useful for simple data retrieval).
  - **POST:** Sends data in the HTTP request body, which is more secure and commonly used for sensitive data.

23.1.1 Input

The <input> tag in HTML is used to create various types of user-input controls within a form. Each <input> element can take different types of input, like text, numbers, emails, passwords, files, dates, and more. The specific behavior and appearance of each <input> is determined by the type attribute.

The types are

1. **Text Input:** Collects single-line text input, like names or emails.

   ```
   <input type="text" name="username" placeholder="Enter your
   ↪  name">
   ```

2. **Password Input:** Similar to text but hides characters as they're typed.

```
<input type="password" name="password" placeholder="Enter your password">
```

3. **Radio Buttons:** Allow users to select a single option from a set.

```
<input type="radio" name="gender" value="male"> Male
<input type="radio" name="gender" value="female"> Female
```

4. **Checkboxes:** Let users select multiple options independently.

```
<input type="checkbox" name="subscribe" value="newsletter"> Subscribe to newsletter
```

5. **Submit Button:** Sends the form data to the server.

```
<input type="submit" value="Submit">
```

6. **Color Picker:** Allows users to pick a color from a color wheel or enter a color code.

```
<input type="color" name="favcolor" value="#ff0000">
```

7. **Date Picker:** Provides a calendar interface for selecting a date.

```
<input type="date" name="birthdate">
```

8. **Email Input:** Accepts and validates email addresses.

```
<input type="email" name="email" placeholder="Enter your email" required>
```

9. **File Upload:** Allows users to upload files from their device.

```
<input type="file" name="profilePicture">
```

10. **Hidden Input:** Stores data that isn't visible to the user but is submitted with the form.

```
<input type="hidden" name="userID" value="12345">
```

11. **Image Button:** Acts as a submit button but uses an image instead of text.

```
<input type="image" src="submit_button.png" alt="Submit"
↪ width="50" height="50">
```

12. **Month Picker:** Allows users to select a specific month and year.

```
<input type="month" name="startmonth">
```

13. **Number Input:** Accepts numeric values, optionally within a specified range.

```
<input type="number" name="quantity" min="1" max="10"
↪ step="1">
```

14. **Range Slider:** Allows users to select a number within a specified range by sliding a handle.

```
<input type="range" name="volume" min="0" max="100"
↪ step="10">
```

15. **Reset Button:** Clears all input fields in the form to their default values.

```
<input type="reset" value="Reset">
```

16. **Search Field:** Provides a field for entering search queries.

```
<input type="search" name="query" placeholder="Search...">
```

17. **Telephone Input:** Used for entering telephone numbers, with optional pattern for specific formats.

```
<input type="tel" name="phone" placeholder="123-456-7890"
↪ pattern="[0-9]{3}-[0-9]{3}-[0-9]{4}">
```

18. **URL Input:** Accepts and validates URLs.

```
<input type="url" name="website"
↪ placeholder="https://example.com">
```

The <label> tag in HTML is used to define labels for form elements, improving accessibility and usability by linking text to a specific input field. Labels help users understand what information is needed in each input field and can also make form elements easier to interact with, especially for screen readers.

The <label> tag can be associated with form elements like <input>, <textarea>, <select>, etc., either by using the for attribute or by nesting the input element inside the <label> tag.

he for attribute of the <label> tag should match the id of the corresponding input element. This way, when a user clicks on the label, it automatically focuses or activates the associated input field.

```
<label for="username">Username:</label>
<input type="text" id="username" name="username">
```

- **name**: gives a name to the input field, used to identify the data in the form submission. This is essential for sending data to a server.
- **value**: specifies the initial value for the input. It's commonly used with text, checkbox, radio, submit, and hidden types.
- **id**: gives a unique identifier to the input, which is useful for linking the <label> element or for JavaScript and CSS targeting.
- **placehold**: displays a hint in the input field, showing users what to enter. The placeholder text disappears when the user starts typing.
- **required**: Specifies that the input must be filled out before submitting the form.
- **readonly**: boolean flag makes the input field read-only, so users cannot edit its value. Useful for showing values that should not be altered.
- **disabled**: disables the input, so it cannot be used or submitted with the form. This often makes the field appear "grayed out."
- **maxlength**: sets a maximum number of characters that the user can enter in the field. Often used with text or password inputs.
- **min**: and max define a minimum and maximum value for numeric fields (type="number", type="range", etc.).
- **step**: specifies the legal number intervals for numeric input types like number and range.
- **pattern**: defines a regular expression that the input's value must match to be valid. This attribute works with text, tel, email, etc.
- **autocomplete**: indicates whether the input should have autocomplete enabled (on) or disabled (off).
- **autofocus**: specifies that the input field should automatically get focus when the page loads.
- **multiple**: allows the selection of multiple values for file input (type="file") or email input (type="email").
- **size**: Specifies the width of the input field in characters. This attribute is rarely used nowadays, as CSS provides more control.
- **accept**: specifies the types of files that the server accepts (used only with type="file"). Common values include file extensions, MIME types, or both.
- **form**: Specifies the ID of a form to which the input belongs, allowing it to be placed outside the <form> element but still be associated with it.

23.1.2 Create the HTML Form

- action="process_form.php" tells the form to submit its data to a file named process_form.php.
- method="POST" specifies that the form data will be sent via POST.

23.1.3 Set Up the PHP Script to Process the Form

Now, create the PHP file process_form.php to handle the form submission. The PHP script will process the form data when it's submitted.

```php
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    // Capture the form data
    $name = htmlspecialchars($_POST['name']); // Use
    ↪ htmlspecialchars to prevent XSS attacks
    $email = htmlspecialchars($_POST['email']);

    // Validate form data (optional)
    if (empty($name) || empty($email)) {
        echo "All fields are required!";
    } else {
        // Process the data (e.g., save to a database, send an
        ↪ email, etc.)
        echo "Thank you, $name. We have received your email as
        ↪ $email.";
    }
}
?>
```

Regular expressions

24.1 Patterns in php

Creating regular expression patterns in PHP involves defining a string pattern between delimiters, commonly / /. Within these delimiters, you place various characters, symbols, and special sequences to create the pattern for matching, searching, or replacing text

You can also use other characters like #, ∼, or % as delimiters. This is useful if your pattern contains many forward slashes, as it reduces the need for escaping.

24.2 Lookaheads and lookbehinds

- **Positive Lookahead:** (?=...) ensures that what follows matches the given pattern.

```php
$pattern = "/apple(?= pie)/";
$string = "apple pie is tasty";
preg_match($pattern, $string, $matches);
print_r($matches); // Output: Array ( [0] => apple )
```

- **Negative Lookahead:** (?!...) ensures that what follows does not match the given pattern.

```php
$pattern = "/apple(?! pie)/";
$string = "apple juice is refreshing";
preg_match($pattern, $string, $matches);
print_r($matches); // Output: Array ( [0] => apple )
```

- **Positive Lookbehind:** (?<=...) ensures that what precedes matches the given pattern.

```php
$pattern = "/(?<=apple )pie/";
$string = "apple pie is delicious";
preg_match($pattern, $string, $matches);
print_r($matches); // Output: Array ( [0] => pie )
```

- **Negative Lookbehind:** (?<!...) ensures that what precedes does not match the given pattern.

```php
$pattern = "/(?<!apple )pie/";
$string = "banana pie is tasty";
preg_match($pattern, $string, $matches);
print_r($matches); // Output: Array ( [0] => pie )
```

24.3 Flags

- **i**: Case-Insensitive
- **m**: Changes the behavior of ∧ and $ so they match the start and end of each line, not just the start and end of the entire string.
- **s**: Allows the dot (.) to match newline characters (\n), so it matches any character including newlines.
- **x**: Allows you to add whitespace and comments within the pattern for readability. Whitespace within the pattern is ignored unless escaped or in a character class.
- **A**: Ensures that the pattern matches only if it's at the beginning of the string, even in multi-line mode.
- **D**: Changes $ to match the end of the string only, ignoring newline characters.
- **U**: Ungreedy, Makes quantifiers like * and + ungreedy by default, meaning they match as little as possible rather than as much as possible.

24.4 preg_match()

Checks if a string matches a regular expression pattern.

```
$pattern = "/php/i"; // "i" for case-insensitive
$string = "Learn PHP";
if (preg_match($pattern, $string)) {
    echo "Match found!";
}
```

24.5 preg_match_all()

Finds all matches of a pattern within a string and returns them in an array.

```
$pattern = "/\d+/"; // Matches any sequence of digits
$string = "2023 is a year with 365 days";
preg_match_all($pattern, $string, $matches);
print_r($matches); // Outputs all sequences of digits found in
↪  the string
```

24.6 preg_replace()

Replaces all occurrences of a pattern within a string with a replacement.

```php
$pattern = "/\s+/"; // Matches whitespace
$replacement = "-";
$string = "Hello World!";
$newString = preg_replace($pattern, $replacement, $string);
echo $newString; // Output: Hello-World!
```

24.7 preg_split()

Splits a string by a pattern and returns an array of substrings.

```php
$pattern = "/[\s,]+/"; // Matches whitespace or commas
$string = "apple, banana orange";
$fruits = preg_split($pattern, $string);
print_r($fruits); // Output: Array ( [0] => apple [1] => banana
↪  [2] => orange )
```

24.8 Capture spaces

In PHP regular expressions, you can capture specific parts of a pattern by placing them inside parentheses (). The captured portions, or subpatterns, can then be retrieved from the result array returned by functions like preg_match or preg_match_all

The preg_match() function finds the first match of the pattern in a string and stores the matched text, along with captured subpatterns, in an array.

```php
$pattern = "/(\w+)\s+(\d+)/"; // Pattern with two capture groups
$string = "Item 123";
preg_match($pattern, $string, $matches);
print_r($matches);
```

If you want to find all occurrences of a pattern in a string and capture all subpatterns for each match, use preg_match_all().

```php
$pattern = "/(\w+)\s+(\d+)/";
$string = "Item 123 and Product 456";
preg_match_all($pattern, $string, $matches);
print_r($matches);
```

24.8.1 Naming captures

PHP also supports named capturing groups, which allow you to name each captured group for easier reference. Named groups use the syntax (?<name>...) or (?P<name>...).

```php
$pattern = "/(?<item>\w+)\s+(?<number>\d+)/";
$string = "Item 123";
preg_match($pattern, $string, $matches);
print_r($matches);
```

PDO: Working with sql databases

- **Why SQL via PHP?**: Can provide an interface for the user that does not require them to worry about database design specifics.

  Although there are other ways to provide this interface, the web-based interface is very common, and PHP is a common and relatively easy way of making it work.

- **Application Programming Interface (API)**: In order for our PHP application to interface with our DBMS, we will need to use an appropriate API (Application Programming Interface) An API is the set of function calls and other resources that are provided to allow you to interface with a given application via your program code.

- **Which API?**: Even for a given DBMS, there can be many API's present. PHP has been around for a while, so many things have evolved and died out. Many of the API's still work. Some work but are considered deprecated, and others are no longer supported at all. In this class, we will be working with the PDO library.

- **PDO Library**: The PHP Data Objects (PDO) library is an object oriented API to connect PHP to SQL servers. It allows you to use a common interface to interact with many different DBMS programs.

  It supports most of the popular relational DBMSes, including MySQL, Postgresql, and SQLite, in a fairly transparent way, so it is more portable than using the other, DBMS specific API's. **Note:** Because PDO is object oriented, it requires at least version 5 of PHP. If you need to use a lower version, you'll need to look into the other API's available.

- **PHP Data Objects**: The PDO library is object oriented. That means that our interactions with the database will be done using objects and their members. There are three basic objects that we will be concerned with:

  - **PDO**: this object handles the connection to the database
  - **PDOStatement**: this object handles prepared statements, and is used to work with result sets
  - **PDOException**: this object is used to store information on errors that have occurred

- **Specifying a Database with a DSN**: Although, once properly initialized, PDO should function the same for any DBMS, you need to properly initialize it by telling it which type of server you are connecting to. To do this, you need to make a DSN string.

DBMS	DSN Format
MySQL	mysql:host=HOSTNAME;dbname=DBNAME
Postgresql	pgsql:host=HOSTNAME;dbname=DBNAME
SQLite 3	sqlite:PATHTODB
SQLite 2	sqlite2:PATHTODB

  MariaDB will use the MySQL interface.

- **Initializing PDO**: Once you've chosen the DBMS you'll be using and you've chosen an appropriate DSN string, you can use that DSN to construct an instance of the PDO class. This object represents a connection to the database specified by the DSN

```php
try { // if something goes wrong, an exception is thrown
    $dsn = "mysql:host=courses;dbname=z123456";
    $pdo = new PDO($dsn, $username, $password);
}
catch(PDOexception $e) { // handle that exception
    echo "Connection to database failed: " .
    ↪ $e->getMessage();
}
```

- **Using PDO to Talk to DBMS**: The PDO library provides three basic ways of running queries for a database, once connected:
  - the PDO::exec() function is used to execute an SQL query that does not return a result (INSERT, UPDATE, etc.)
  - the PDO::query() function is used to execute an SQL query that will return a result (SELECT)
  - the PDO::prepare() function should be used when constructing a query from user input.
- **Using PDO::exec()**: PDO::exec() is used to run a query that does not return any results. Instead of returning a result set, it will tell you how many rows were affected by your query.

```php
// Three examples follow.
$n = $pdo->exec("INSERT INTO Students (FName) VALUES
↪  ('Victor');");
$n = $pdo->exec("UPDATE Students SET LName='Husky' WHERE
↪  FName='Victor';");
$n = $pdo->exec("DELETE FROM OldJunk;");
```

- **Using PDO::query()**: PDO::query() is used to run a query that does return a result. The result set is returned as a PDOStatement object.

```php


$sql = "SELECT phone FROM Customer;";


$result = $pdo->query($sql);
```

- **Using PDO::prepare()**: The third option is to use the PDO::prepare() command. This is useful for situations where the same query is run multiple times in the same script, and can also help you to avoid some SQL Injection issues. Once prepare() succeeds, you run the query with the execute() method of the PDOStatement returned by prepare()

  You can use a colon before a value name in your query to denote where the execute statement will insert the value of the given name:

```php
<?php
    # Notice that we use :color below in our SQL template
    $sql = 'SELECT name, color, calories
    FROM fruit
    WHERE calories < :calories AND color = :color';

    $prepared = $pdo->prepare($sql, array(PDO::ATTR_CURSOR
    => PDO::CURSOR_FWDONLY));
    # The value associated with the :color key will be used
    when executing

    $success = $prepared->execute(array(':calories' => 150,
    ':color' => 'red'));
    # if(success == true)thenprepared will be ready to be
    used as a result set
    # with fetch() or fetchAll() -- just like the object
    returned by query()
?>
```

It is also possible to use a ? in your query as a positional parameter.

```php
<?php
    # Execute a prepared statement by passing an array of
    values
    $prepared = $pdo->prepare('SELECT name, color, calories
                               FROM fruit
                               WHERE calories < ? AND color =
    ?');

    # Here we execute the query twice with different
    parameters.
    # The ?'s will be replaced with the values in the array
    specified,
    # in the order they are specified.
    $prepared->execute(array(150, 'red'));
    $red = $prepared->fetchAll();
    $prepared->execute(array(175, 'yellow'));
    $yellow = $prepared->fetchAll();
?>
```

- **Named placeholders**: In PDO (PHP Data Objects), named placeholders are a way to represent dynamic values in SQL statements using identifiable names instead of anonymous ? placeholders. Named placeholders make SQL statements easier to read and maintain, especially when there are multiple variables or parameters.

  A named placeholder is written as :name in an SQL query, where name is an identifier you choose. When you prepare the SQL statement, you can bind actual values to these placeholders by their names, allowing you to execute the query with specific data values.

- **anonymous placeholders**: In PDO (PHP Data Objects), anonymous placeholders (also called positional or unnamed placeholders) are represented by question marks ? in SQL statements. These placeholders are used to represent values that will be dynamically bound to the SQL query at execution time.

  When using anonymous placeholders, each placeholder corresponds to a specific value based on its position in the SQL query. When you prepare and execute the query, you provide an array of values that will replace each ? in the order they appear.

- **Execute**: In PHP's PDO (PHP Data Objects) extension, the execute() method is used to run a prepared SQL statement with specific values for placeholders. It's part of the PDOStatement class and is essential for safely executing queries that include dynamic data, protecting against SQL injection attacks.

  After preparing a statement with PDO::prepare(), you call execute() to supply any values for placeholders (either named or anonymous) and run the query.

- **Dealing with result sets**: Once you have a result set (stored in a PDOStatement from query() you can use its PDOStatement::fetch() or PDOStatement::fetchAll() methods to get the data returned.

  If you would like to work on one row at a time, as if using the mysql_fetch_array() function from the original MySQL API, use fetch().

  To grab all of the rows at the same time, use fetchAll().

```php
<?php
    # FETCH_BOTH means that you will get both position
    ↪ indices and the column names
    # as keys in the array returned
    $row = $result->fetch(PDO::FETCH_BOTH);

    # this returns all of the rows at once in a
    ↪ two-dimensional array
    $allrows = $result->fetchAll();
?>
```

- **Handling Errors**: As of the time of this writing, there are three modes PDO can use to handle errors.
  - **PDO::ERRMODE_SILENT**: the default mode. PDO will set the error code for you to inspect using the errorCode and errorInfo methods on your PDO and PDOStatement objects
  - **PDO::ERRMODE_WARNING**: in addition to setting the error code, emits a traditional E_WARNING message. Use for debugging/testing when you just want to see what problems occurred without interrupting the flow of the application.
  - **PDO::ERRMODE_EXCEPTION**: in addition to setting the error code, also throw PDOException when an error occurs.

  You can set which one you'd like PDO to use with the setAttribute method of the PDO object you're using, as below:

```php
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT);
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
$pdo->setAttribute(PDO::ATTR_ERRMODE,
    ↪ PDO::ERRMODE_EXCEPTION);
```

- **Members of the PDO object**

```php
PDO {
    // constructor has a bunch of optional parameters, check
    ↪ reference if needed
    public __construct ( string $dsn [, string $username [,
    ↪ string $password [,
    array $options ]]] )
    public bool beginTransaction ( void )
    public bool commit ( void )
    public mixed errorCode ( void )
    public array errorInfo ( void )
    public int exec ( string $statement )
    public mixed getAttribute ( int $attribute )
    public static array getAvailableDrivers ( void )
    public bool inTransaction ( void )
    public string lastInsertId ([ string $name = NULL ] )
    public PDOStatement prepare ( string $statement [, array
    ↪ $driver_options ] )
    public PDOStatement query ( string $statement )
    public string quote ( string $string [, int
    ↪ $parameter_type = PDO::PARAM_STR ] )
    public bool rollBack ( void )
    public bool setAttribute ( int $attribute , mixed $value
    ↪ )
}
```

- **Members of the PDOStatement object**:

```php
PDOStatement implements Traversable {
    /* Properties */
    readonly string $queryString;
    /* Methods */
    public bool bindColumn ( mixed $column , mixed &$param
    ↪  [, int $type [, int $maxlen [, mixed $driverdata ]]]
    ↪  )
    public bool bindParam ( mixed $parameter , mixed
    ↪  &$variable [,
    int $data_type = PDO::PARAM_STR [, int $length [, mixed
    ↪  $driver_options ]]] )
    public bool bindValue ( mixed $parameter , mixed $value
    ↪  [, int $data_type = PDO::PARAM_STR ] )
    public bool closeCursor ( void )
    public int columnCount ( void )
    public void debugDumpParams ( void )
    public string errorCode ( void )
    public array errorInfo ( void )
    public bool execute ([ array $input_parameters ] )
    public mixed fetch ([ int $fetch_style [, int
    ↪  $cursor_orientation = PDO::FETCH_ORI_NEXT [,
    int $cursor_offset = 0 ]]] )
    public array fetchAll ([ int $fetch_style [, mixed
    ↪  $fetch_argument [, array $ctor_args = array() ]]] )
    public mixed fetchColumn ([ int $column_number = 0 ] )
    public mixed fetchObject ([ string $class_name =
    ↪  "stdClass" [, array $ctor_args ]] )
    public mixed getAttribute ( int $attribute )
    public array getColumnMeta ( int $column )
    public bool nextRowset ( void )
    public int rowCount ( void )
    public bool setAttribute ( int $attribute , mixed $value
    ↪  )
    public bool setFetchMode ( int $mode )
}
```

- **Members of the PDOException object**

```
PDOException extends RuntimeException {
    /* Properties */
    public array $errorInfo ;
    protected string $code ;
    /* Inherited properties */
    protected string $message ;
    protected int $code ;
    protected string $file ;
    protected int $line ;
    /* Inherited methods */
    final public string Exception::getMessage ( void )
    final public Throwable Exception::getPrevious ( void )
    final public mixed Exception::getCode ( void )
    final public string Exception::getFile ( void )
    final public int Exception::getLine ( void )
    final public array Exception::getTrace ( void )
    final public string Exception::getTraceAsString ( void )
    public string Exception::__toString ( void )
    final private void Exception::__clone ( void )
}
```

25.1 PDO: Could not find driver

The ”could not find driver” error indicates that the MySQL PDO driver is missing on your system. Here's how to resolve it on Arch Linux:

On Arch Linux, enabling PDO MySQL support requires modifying the PHP configuration file:

Open the PHP INI file for editing:

```
sudo nano /etc/php/php.ini
```

Uncomment the pdo_mysql Extension:

```
;extension=pdo_mysql
```

Remove the ; at the beginning to uncomment it, so it looks like this:
If you're using Apache, restart it to apply the changes:

```
sudo systemctl restart httpd
```

25.2 Example

Consider the following mariadb relation under the database *test*

```
USE test;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE
);

INSERT INTO users (name, email) VALUES
('Alice', 'alice@example.com'),
('Bob', 'bob@example.com'),
('Charlie', 'charlie@example.com');
```

Then with php, we can connect to the database

```
try {
    $host = "localhost";
    $db = "test";
    $dsn = "mysql:host=$host;dbname=$db";
    $pdo = new PDO($dsn, $username, $password);
} catch (PDOException $e) {
    echo "Connection failed";
    echo $e->getMessage();
}
```

25.2.1 pdo->exec()

We can use exec() to interact with the DDL, for operations on the database that dont return rows. For example, suppose we wanted to add a new column *lastname*

```
try {
    $affectedRows = $pdo->exec("ALTER TABLE users ADD lastname
    ↪  VARCHAR(50);");
} catch (PDOException $e) {
    echo $e->getMessage() . "<br/>";
}
```

If this column were to already exist in the relation, exec would throw an error, so we catch it.

25.2.2 pdo->query()

We can start by selecting all columns from the users relation

```
$s1 = "SELECT * from users";
$result = $pdo->query($s1);

foreach ($result as $item) {
    echo $item["id"] . "\t" . $item["name"] . "\t" .
    ↪  $item["email"] . "<br/>";
}
```

The result of the query is stored in the multidimensional array $result. Then, we can loop over the arrays in $result, where each array is a tuple from the returned query. The keys in the tuple array are the names of the columns.

25.2.3 pdo->prepare() with execute and fetch

Instead, we can use prepare

```
$s1 = "SELECT * from users";
// Builds prepared object
$prepared = $pdo->prepare($s1);
// Executes query, storing in prepared
$success = $prepared->execute();
if ($success) {
    $result = $prepared->fetchAll();
}

foreach ($result as $item) {
    echo $item["id"] . "\t" . $item["name"] . "\t" .
    ↪  $item["email"] . "<br/>";
}
```

Calling fetchAll() on the prepared object after execute gives us the same multidimensional arary that query() did.

Instead, we could use fetch() to grab single rows

```
$s1 = "SELECT * from users";
// Builds prepared object
$prepared = $pdo->prepare($s1);
// Executes query, storing in prepared
$success = $prepared->execute();

// Get the first and second tuple with fetch()
$row1 = $prepared->fetch();
$row2 = $prepared->fetch();
```

25.2.4 Arguments to fetch and fetchAll

- **PDO:**:FETCH_COLUMN: Fetches only a single column from the next row in the result set. This is useful if you're interested in just one column.
- **PDO:**:FETCH_KEY_PAIR: Fetches the result set as an associative array where the first column is the key and the second column is the value. This is useful for creating key-value pairs from the result set.
- **PDO:**:FETCH_UNIQUE: Fetches the result set as an associative array where the first column is a unique key. Each row's other columns form an associative array.
- **PDO:**:FETCH_GROUP: Fetches the result set as a multidimensional array grouped by the values of the first column.
- **PDO:**:FETCH_CLASS: Maps each row to an instance of a specified class. The class properties are populated by column names.
- **PDO:**:FETCH_INTO: Fetches data into an existing object, rather than creating a new instance each time.
- **PDO:**:FETCH_FUNC: Calls a specified function for each row of data fetched, passing the column values as arguments.

Note: We can also give these to query.

25.2.5 Placeholders with prepare

Using placeholders with prepare in PHP's PDO (PHP Data Objects) helps prevent SQL injection and makes your queries cleaner and easier to read. Here's how to use placeholders with prepare

- Positional Placeholders (?)
- Named Placeholders (:name)

Using positional placeholders is straightforward when you don't need to name each placeholder individually.

Placeholders in prepared statements cannot be used for identifiers like column names, table names, or other SQL syntax elements. Placeholders can only be used for values (e.g., in WHERE clauses or INSERT values).

Placeholders cannot be used with exec, we must use prepare with execute

```
$statement = "UPDATE users SET lastname=? WHERE id=?";
try {
    $p = $pdo->prepare($statement);
    $s = $p->execute(array("l3", 3));
} catch (PDOException $e) { echo "noop"; }
```

Named placeholders are useful for readability and when there are many parameters.

```
$statement = "UPDATE users SET lastname=:name WHERE id=:id";
try {
    $p = $pdo->prepare($statement);
    $s = $p->execute(array(":name" => "Smith", ":id" => 1));
} catch (PDOException $e) { echo "noop"; }
```

The same rules that we discussed for positional placeholders apply for named placeholder

Alternately, we can use bindParam() and bindValue()

- **bindParam():** Binds a variable by reference, which means it allows the variable's value to change before execute() is called.
- **bindValue():** Binds a specific value directly to the placeholder.

```
$statement = "UPDATE users SET lastname=:name WHERE id=:id";
try {
    $p = $pdo->prepare($statement);

    $p->bindValue(':name', "Appleseed");
    $p->bindValue(':id', "2");

    $s = $p->execute();
} catch (PDOException $e) { echo "noop"; }
```

25.2.6 Enable PDO Exception Mode

To catch errors effectively, set the error mode to exceptions:

```
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

25.2.7 Transactions

PHP's PDO (PHP Data Objects) extension provides a consistent interface for accessing databases. Transactions in PDO are used to ensure a set of database operations are executed together in a single unit of work. If any operation within the transaction fails, the changes can be rolled back to maintain data integrity.

1. **Begin a Transaction:** Start a transaction with the database.
2. **Execute Queries:** Perform a series of SQL operations that are part of the transaction.
3. **Commit:** Save the changes made during the transaction.
4. **Rollback:** Undo all changes if something goes wrong.

Start a Transaction: Begin the transaction with:

```
1 $pdo->beginTransaction();
```

Then, perform the desired database operations using prepare() and execute()

If all operations succeed, make the changes permanent:

```
1 $pdo->commit();
```

If an error occurs, undo all changes by rolling back the transaction:

```
1 $pdo->rollBack();
```

```php
<?php
try {
    // Create a new PDO instance
    $pdo = new PDO('mysql:host=localhost;dbname=testdb',
    ↪  'username', 'password');
    $pdo->setAttribute(PDO::ATTR_ERRMODE,
    ↪  PDO::ERRMODE_EXCEPTION);

    // Start the transaction
    $pdo->beginTransaction();

    // Perform database operations
    $stmt = $pdo->prepare("INSERT INTO accounts (name, balance)
    ↪  VALUES (?, ?)");
    $stmt->execute(['Alice', 500]);
    $stmt->execute(['Bob', 1000]);

    // Simulate an error to demonstrate rollback
    // This statement will fail if the `balance` column is set
    ↪  to NOT NULL
    $stmt->execute(['Charlie', null]);

    // Commit the transaction
    $pdo->commit();
    echo "Transaction committed successfully.";
} catch (Exception $e) {
    // Rollback the transaction on failure
    $pdo->rollBack();
    echo "Transaction failed: " . $e->getMessage();
}
```