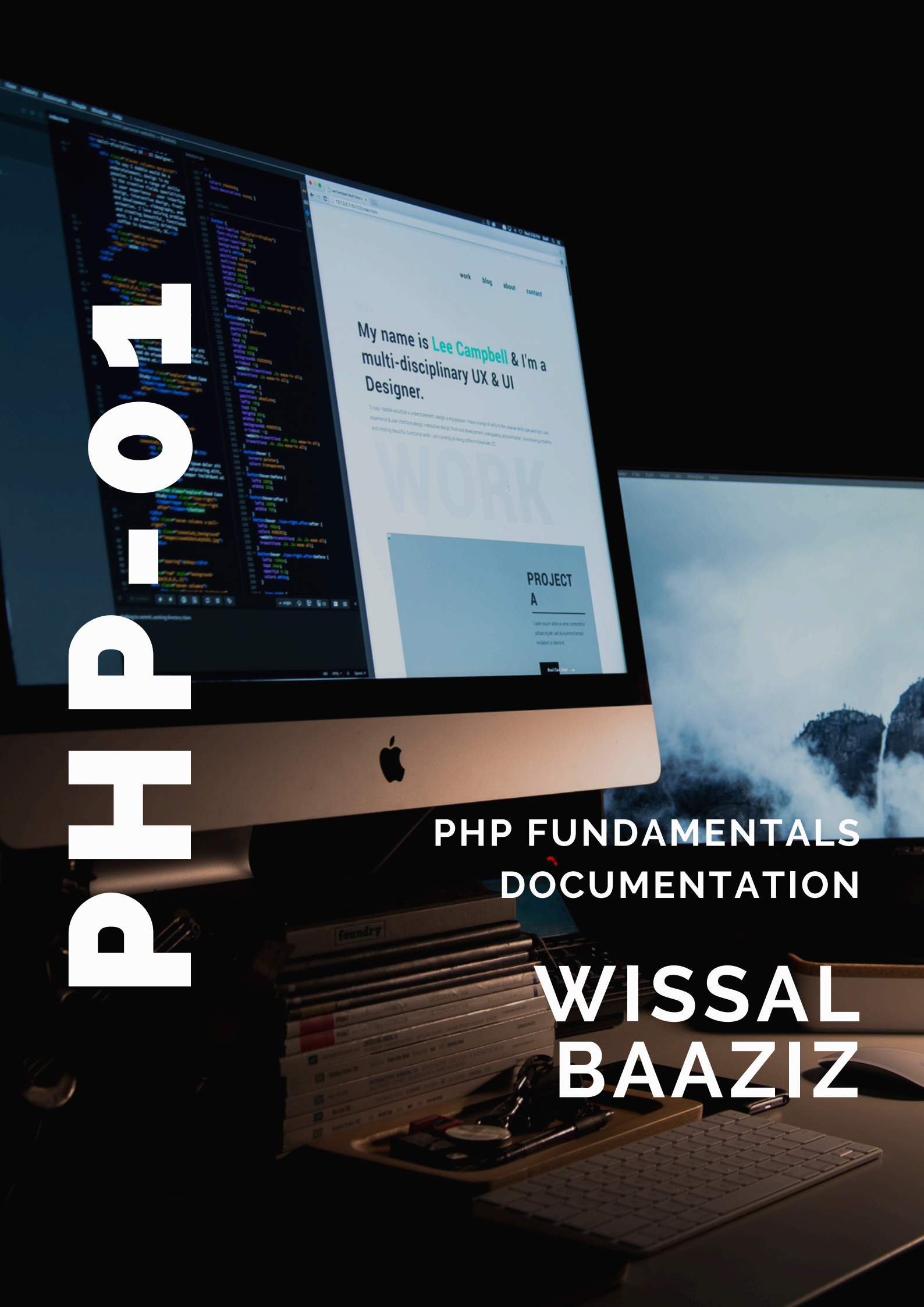


PHP FUNDAMENTALS DOCUMENTATION

WISSAL BAAZIZ

SOMMAIRE

PHP Tags

Dynamic Integration

Termination & Tags

Comment Overview

Types in PHP

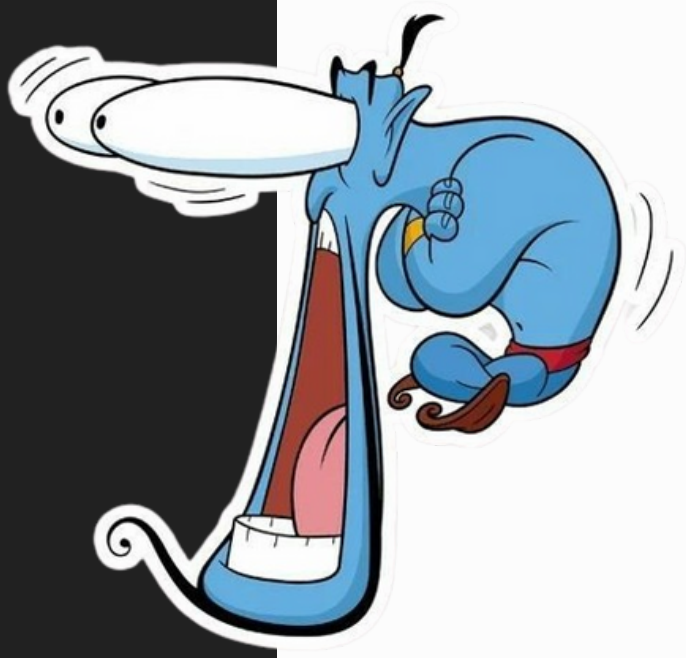
Variable Basics

Constant Definition

Expressions & Operators

Control Structures

Function Concepts



When documenting PHP code, it's essential to provide clear explanations about PHP tags, their usage, and best practices.

PHP TAGS

In PHP, code is encapsulated within opening and closing tags that inform the PHP parser when to start and stop interpreting the enclosed code. The primary tags are <?php and ?>, designating the beginning and end of the PHP code block. Anything outside these tags is ignored by the PHP parser.

Example: Opening and Closing Tags

```
<?php
    echo 'If you want to serve PHP code in XHTML or XML documents, use these tags.';
?>

You can use the short echo tag to <?= 'print this string' ?>.
It's equivalent to <?php echo 'print this string' ?>.

<? echo 'This code is within short tags, but will only work '.
           'if short_open_tag is enabled'; ?>
```

Short tags, such as **<?** and **<?=**, are available by default but can be disabled through the **short_open_tag** php.ini configuration directive. To maximize compatibility, it is recommended to use the full tags (**<?php ?>** and **<?= ?>**).

Note on Short Tags

As short tags can be disabled, it is advisable to use only the standard tags to ensure broader compatibility. For example:

```
<?php
    echo "Hello world";

    // ... more code

    echo "Last statement";

    // The script ends here with no PHP closing tag
```

Omitting the closing tag at the end of a file is preferred to prevent unintended whitespace or new lines after the PHP closing tag, which could lead to unexpected effects.

DYNAMIC INTEGRATION

PHP seamlessly integrates with HTML, allowing the creation of dynamic web content. Enclosed within **<?php** and **?>** tags, PHP code coexists with static HTML, enabling the generation of dynamic elements within the HTML structure.
This integration empowers developers to embed server-side logic, making web applications responsive and interactive. The conditional processing and sequential execution of PHP code contribute to the versatility of PHP in creating dynamic content.

Conditional Content Escaping in PHP

Conditional content escaping in PHP provides flexibility in including or excluding code blocks based on specified conditions. The example below illustrates this concept:

Example: Conditional Escaping

```
<?php if ($expression == true): ?>
  Displayed when the expression is true.
<?php else: ?>
  Displayed when the expression is false.
<?php endif; ?>
```

In this example:

- The **if** statement checks if **$expression** is true.
- The content within the first block is displayed if true.
- If false, the content within the second block is displayed.

TERMINATION & TAGS

In PHP, terminate statements with semicolons; the closing tag implies a semicolon, even with a trailing newline. Example:

```
<?php echo "PHP code here"; ?> No newline <?= "Another line" ?>
```

Entering/exiting PHP:

```
<?php echo 'Entering PHP with a block'; ?> <?php echo 'Entering PHP with a single line'
?> <?php echo 'No need for the last closing tag';
```

Entering/exiting PHP:

```
<?php echo "File content"; ?> // No closing tag here
```

Omitting the closing tag ensures cleaner file inclusion and compatibility, especially with include or require. It promotes cleaner code and prevents whitespace issues.

COMMENT OVERVIEW

PHP comments play a crucial role in code documentation and clarity. Here's a concise guide to different comment types:

Single-Line Comments:

```
// This is a single-line comment
$variable = 42; // Comments can follow code on the same line
```

Multi-Line Comments:

```
/*
   This is a
   multi-line comment
*/
```

Mastering PHP comments enhances code readability, fostering a well-documented and collaborative codebase.

TYPES IN PHP

PHP supports a variety of data types and concepts related to types. Let's explore each of them:

1. NULL:

Represents the absence of a value or a variable that has been explicitly set to null.

2. Booleans:

Logical values, either true or false, used in conditional statements.

3. Integers:

Whole numbers without decimal points, e.g., 42 or -17.

4. Floating Point Numbers:

Numbers with decimal points or in exponential form, e.g., 3.14 or 1.2e3.

5. Strings:

Sequence of characters, enclosed in single or double quotes.

6. Numeric Strings:

Strings that represent numeric values, e.g., "42".

7. Arrays:

Ordered maps that hold data in key-value pairs.

8. Objects:

Instances of user-defined classes that encapsulate data and behavior.

9. Enumerations:

Not directly supported in PHP, but can be simulated using class constants or associative arrays.

10. Resources:

Special variable types holding references to external resources, like database connections.

11. Callbacks/Callables:

Functions, methods, or anonymous functions that can be called via call_user_func().

12. Mixed:

Represents a value that can have any data type.

13. Void:

Indicates that a function does not return any value.

PHP is dynamically typed, meaning variable types are determined at runtime. Type declarations can be used to statically define aspects of the language. Operations are restricted by types, but PHP attempts type juggling if an unsupported operation occurs.
Tip: Refer to type comparison tables for examples of different type comparisons.

Type Casting and Evaluation

Use type casting or settype() to force expression evaluation to a specific type. The var_dump() function displays value and type, while get_debug_type() retrieves the type. Check types using is_type functions.

```php
$a_bool = true;   // bool
$a_str  = "foo";  // string
$a_str2 = 'foo';  // string
$an_int = 12;     // int

echo get_debug_type($a_bool), "\n";
echo get_debug_type($a_str), "\n";

if (is_int($an_int)) {
    $an_int += 4;
}
var_dump($an_int);

if (is_string($a_bool)) {
    echo "String: $a_bool";
}
```

Output (PHP 8):

```
bool
string
int(16)
```

Note: Prior to PHP 8.0.0, use gettype() instead of get_debug_type().

VARIABLES

Basics

In PHP, variables are used to store and manipulate data. Variable names in PHP are case-sensitive and must start with a dollar sign $, followed by the variable name.

```php
$variableName = "Hello, PHP!";
```

Predefined Variables

PHP also has a set of predefined variables, often referred to as superglobals, which are accessible from any part of your script. Examples include $_GET, $_POST, and $_SESSION. These hold data received from external sources like forms or server settings.

Variable Scope

Variable scope defines where a variable can be accessed or modified. PHP has local, global, and static scope variables. Understanding scope is crucial for avoiding naming conflicts and unintended variable modifications.

```php
$globalVariable = "I am global";

function exampleFunction() {
    $localVariable = "I am local";
    echo $globalVariable; // Accessing global variable inside the function
}

exampleFunction();
```

Variable Variables

PHP allows you to dynamically create variable names using the values of other variables. This is known as variable variables.

```php
$firstName = "John";
$$firstName = "Doe"; // Creates a variable $John with the value "Doe"
echo $John; // Outputs "Doe"
```

Variables from External Sources

When handling user input or data from external sources, it's crucial to validate and sanitize the input. PHP provides superglobal arrays like $_GET and $_POST to access data sent through forms or URLs.

```
phpCopy code
$userInput = $_POST['username'];
```

Understanding the basics of variables, predefined variables, variable scope, and handling variables from external sources is fundamental for effective PHP programming.

CONSTANTS

In PHP, constants are used to store unchangeable values, like configuration settings or fixed values used throughout a script. Once defined, constants cannot be changed or undefined during script execution.

Syntax

Constants are defined using the define() function. By convention, constant names are typically uppercase.

```
define("PI", 3.14);
define("SITE_NAME", "My Website");
```

Predefined Constants

PHP also has a set of predefined constants that provide information about the environment, version, and other settings. Examples include PHP_VERSION, PHP_OS, and PHP_INT_MAX.

```
echo "Current PHP Version: " . PHP_VERSION;
```

Magic Constants

Magic constants are predefined constants that change based on their usage in a script. They are identified by a double underscore prefix. Examples include __LINE__ (current line number) and __FILE__ (current file name).

```
echo "This is line " . __LINE__ . " in file " . __FILE__;
```

EXPRESSIONS & OPERATORS

Expressions

In PHP, expressions are combinations of values, variables, operators, and functions that, when evaluated, produce a result. Expressions are the building blocks of PHP code, used to perform calculations, make decisions, and manipulate data.

Operators

Operators in PHP perform operations on variables and values. They include arithmetic, increment/decrement, assignment, bitwise, comparison, error control, execution, logic, string, array, and type operators.

Operator Precedence

Operator precedence determines the order in which operators are evaluated in an expression. It helps avoid ambiguity and defines the priority of operators. For example, multiplication takes precedence over addition.

Arithmetic Operators

Arithmetic operators perform basic mathematical operations like addition, subtraction, multiplication, and division.

```php
$result = 5 + 3; // Addition
$result = 8 - 2; // Subtraction
$result = 4 * 6; // Multiplication
$result = 9 / 3; // Division
$result = 7 % 2; // Modulus (remainder)
```

Increment and Decrement Operators

Increment (++) and decrement (--) operators are used to increase or decrease the value of a variable by 1.

```php
$counter = 5;
$counter++; // $counter is now 6
$counter--; // $counter is now 5 again
```

Assignment Operators

Assignment operators are used to assign values to variables. The basic assignment operator is =.

```
$number = 10; // Assignment
$number += 5; // Addition assignment (same as $number = $number + 5)
```

Bitwise Operators

Bitwise operators perform operations at the bit level. They include AND (&), OR (|), XOR (^), and others.

```
$result = $a & $b; // Bitwise AND
$result = $a | $b; // Bitwise OR
$result = $a ^ $b; // Bitwise XOR
```

Comparison Operators

Comparison operators compare values and return a boolean result. Examples include equal (==), identical (===), greater than (>), and less than (<).

```
$result = ($a == $b); // Equal
$result = ($a === $b); // Identical
$result = ($a > $b); // Greater than
```

CONTROL STRUCTURES

Introduction

Control structures in PHP are constructs that allow you to control the flow of your program. These include conditional statements (if, else, elseif), loops (while, do-while, for, foreach), and other constructs that alter the program's execution flow.

if Statement

The if statement allows you to execute a block of code based on a specified condition.

```php
if ($condition) {
    // Code to execute if the condition is true
}
```

else Statement

The else statement is used in conjunction with if to execute a different block of code when the initial condition is false.

```php
if ($condition) {
    // Code to execute if the condition is true
} else {
    // Code to execute if the condition is false
}
```

elseif/else if Statement

The elseif or else if statement allows you to add additional conditions.

```php
if ($condition1) {
    // Code to execute if condition1 is true
} elseif ($condition2) {
    // Code to execute if condition2 is true
} else {
    // Code to execute if both conditions are false
}
```

Alternative Syntax for Control Structures

PHP also supports an alternative syntax for control structures, particularly useful within templates or HTML.

```php
<?php if ($condition): ?>
    <!-- HTML or other content here -->
<?php elseif ($anotherCondition): ?>
    <!-- More content -->
<?php else: ?>
    <!-- Default content -->
<?php endif; ?>
```

while Loop

The while loop executes a block of code as long as a specified condition is true.

```php
while ($condition) {
    // Code to execute as long as the condition is true
}
```

do-while Loop

The do-while loop is similar to while, but it guarantees that the code block is executed at least once.

```php
do {
    // Code to execute at least once
} while ($condition);
```

for Loop

The for loop is used to iterate a code block for a specified number of times.

```php
for ($i = 0; $i < 5; $i++) {
    // Code to execute in each iteration
}
```

foreach Loop

The foreach loop is used to iterate over arrays or other iterable objects.

```php
foreach ($array as $value) {
    // Code to execute for each element in the array
}
```

break Statement

The break statement is used to exit the current loop or switch statement.

```
while ($condition) {
    if ($someCondition) {
        break; // Exit the loop
    }
}
```

continue Statement

The continue statement is used to skip the rest of the current loop iteration and move to the next one.

```
foreach ($array as $value) {
    if ($someCondition) {
        continue; // Skip to the next iteration
    }
}
```

switch Statement

The switch statement is used to perform different actions based on different conditions.

```
switch ($variable) {
    case 'value1':
        // Code to execute if $variable equals 'value1'
        break;
    case 'value2':
        // Code to execute if $variable equals 'value2'
        break;
    default:
        // Code to execute if none of the cases match
}
```

match Statement

The match statement is a more powerful and concise replacement for the switch statement introduced in PHP 8.0.

```
$result = match ($variable) {
    'value1' => 'Result 1',
    'value2' => 'Result 2',
    default => 'Default Result',
};
```

declare Statement

The declare statement is used to set execution directives for a block of code.

```
declare(strict_types=1);
// Code with strict type checking
```

return Statement

The return statement is used to exit a function and return a value to the calling code.

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

require, include, require_once, include_once

These statements are used to include files into a script. require and include include the file each time it is called, while require_once and include_once include the file only if it has not been included before.

```
require 'header.php';
include_once 'functions.php';
```

goto Statement

The goto statement allows jumping to another section in the code. However, it is generally considered bad practice and should be used cautiously.

```
goto a;
echo 'This will not be executed.';

a:
echo 'This will be executed.';
```

Understanding control structures is crucial for writing dynamic and flexible PHP code. Choose the appropriate structure based on the specific requirements of your program.

FUNCTIONS

User-defined Functions

In PHP, user-defined functions allow you to encapsulate a set of statements to perform a specific task. They enhance code modularity and reusability.

```php
function greet($name) {
    echo "Hello, $name!";
}

// Call the function
greet("John");
```

Function Arguments

Functions can accept parameters, which act as variables within the function. You can define default values for parameters.

```php
function add($a, $b = 1) {
    return $a + $b;
}

$result = add(5); // $result is 6
```

Returning Values

Functions can return values using the return statement. The returned value can be assigned to a variable or used directly.

```php
function multiply($a, $b) {
    return $a * $b;
}

$result = multiply(3, 4); // $result is 12
```

Variable Functions

PHP supports variable functions, allowing you to dynamically call a function based on a variable value.

```php
$operation = "add";

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

$result = $operation(2, 3); // $result is 5
```

Internal (Built-in) Functions

PHP provides a rich set of built-in functions for various purposes, such as string manipulation, array handling, and more.

```php
$string = "Hello, PHP!";
$length = strlen($string); // $length is 12
```

Anonymous Functions

Anonymous functions, also known as closures, allow you to create functions without specifying a name. They are useful for one-time use or passing as arguments.

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

$result = $add(5, 3); // $result is 8
```

Arrow Functions

Arrow functions, introduced in PHP 7.4, provide a concise syntax for writing anonymous functions with a simplified syntax.

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

$result = $add(5, 3); // $result is 8
```

First-Class Callable Syntax

PHP treats functions as first-class citizens, allowing them to be used as parameters, returned from other functions, and assigned to variables.

```php
function square($x) {
    return $x * $x;
}

$func = "square";
$result = $func(4); // $result is 16
```

Understanding functions is fundamental for writing modular and maintainable PHP code. Choose the appropriate function type based on the task at hand.

In conclusion, mastering the fundamentals of PHP lays a robust foundation for creating dynamic and interactive web applications. With a solid understanding of PHP's syntax, variables, and control structures, you are well-equipped to embark on a journey of building powerful and efficient web solutions. As you delve deeper into the world of PHP, remember that a strong grasp of the fundamentals is the key to unlocking the full potential of this versatile scripting language.

Stay tuned for Part Two of this documentation, where we will delve into Object-Oriented Programming (OOP) in PHP. Discover advanced techniques, design patterns, and best practices that will elevate your PHP programming skills to the next level. Happy coding and get ready for an exciting exploration of OOP in PHP!