LECTURE NOTES

Modern PHP Developer

Luis Ramirez

Section 1: Introduction

In this section, we talk about what PHP is, what you can expect to learn, and how to install PHP on your machines.

What is PHP?

There are no notes for this lecture.

Download the Free E-Book

There are no notes for this lecture.

PHP Environment

In this lecture, we learned about environments and took the time to set up an environment for PHP. First, let's understand the concept of an environment.

In the programming world, an environment is a place where you can write, test, and run computer code. Think of it like a virtual workspace with all the tools you need to make a program work.

Generally, there are two types of environments, local and production.

Local Environment

A local environment refers to the setup you have on your personal computer or local machine, where you can write, test, and run your code. Local environments are primarily used for development and testing purposes.

This gives you, the developer, the opportunity to break things without disrupting your users. You can freely modify and make changes before shipping a program to the real world.

Production Environment

On the other hand, a production environment is a live environment where your code is actually deployed and running for end users.

A production environment is typically hosted on a remote server, and it is designed to handle the demands of real-world usage, including security, scalability, and reliability. The production environment is also monitored, managed, and maintained by a team of professionals to ensure optimal performance.

Using Replit

For this course, we'll be using Replit to help us get started. Watch the video for instructions on how to set up a PHP environment on Replit.

What about Docker?

Professional PHP developers use Docker to manage their environments. Docker is a bit too advanced for us right now, but it will be covered in a future lecture. For now, Replit will suffice.

Resources

- XAMPP
- MAMP
- Laragon
- Replit

Section 2: Working with Data

In this section, we discovered the basic syntax of PHP to help us write dynamic web pages.

The PHP Interpreter

In this lecture, we learned about the PHP interpreter, whose job it is to transform our code into machine code. PHP is an open-source general-purpose scripting language for developing web applications.

- Open-source software (OSS) refers to code that has been freely released to the public.
- A general-purpose language is a programming language that can be utilized to develop different kinds of programs.
- PHP is considered to be a scripting language because it's interpreted.

Compiled vs. Interpreted

So, what does it mean when a programming language is interpreted? Computers are only capable of understanding binary code. Binary code can be difficult to read and write. Herein lies the problem.

For this reason, programming languages were introduced for writing instructions to be executed by a machine. While programming languages are easier to read, computers don't understand the instructions we're giving them. Therefore, our code must be transformed into machine code. This process is known as compilation.

A compiler and an interpreter are both tools used to translate computer code into a form that a computer can execute. The main difference between the two is how they do it:

- A compiler translates the entire source code of a program into machine code (binary) in one go and saves it as an executable file. This file can then be run as many times as needed without the need to recompile it.
- An interpreter, on the other hand, executes the code line by line, translating each line of code into machine code and then executing it immediately. It does not generate an executable file.

Think of it this way: **a compiler is like a translator who translates a book into another language, then gives you the translated book to read. An interpreter is like a translator who translates a sentence, then reads it to you immediately.**

PHP is interpreted. It's safe to say that PHP is a scripting language. However, you can also call it a programming language, and no one would complain otherwise.

Resources

- PHP GitHub

Running a PHP Script

In this lecture, we ran our first PHP script. PHP files can be created with the **.php** extension. In a PHP file, you can freely write plain text like so:

```
Hello World!
```

PHP supports all HTML syntax as well.

```
<h1>Hello World</h1>
```

Static vs. Dynamic Websites

Websites that deliver the same content to visitors regardless of what action the user performs are known as static websites.

1. Browsers can send a request to a server for a specific file.
2. Servers will search for that file.
3. If a file is found, the file is sent back to the browser.

On the other hand, dynamic websites generate unique content to its visitors based on their actions.

1. Browsers can send a request to a server for a specific file.
2. Servers will search for that file.
3. If the file is a PHP file, the PHP file will run through the PHP interpreter, generating an HTML output.
4. The HTML output is sent back to the browser.

PHP Tags

In this lecture, we learned about how to enter PHP mode. By default, PHP does not process raw text or HTML. We must explicitly enter PHP mode to begin giving machines instructions. Entering PHP mode can be done with a pair of PHP tags.

```
<?php ?>
```

Multiple PHP tags can be written inside a PHP file.

```
<?php ?>
<?php ?>
```

They can be written from within HTML like so.

```
<h1><?php ?></h1>
```

However, you cannot write HTML inside PHP like so.

```
<?php <h1></h1> ?>
```

This is because the rules of PHP mode are different from HTML mode.

Inside PHP tags, we must write valid code. In the English language, there are rules for how English is read and written. There's a specific structure we must follow for writing English, such as ending a sentence with a period or capitalizing the first word of a sentence. The rules for human languages are referred to as grammar.

In a similar sense, programming languages have rules for how they're read and written. These rules are referred to as a programming language's syntax.

The echo Keyword

In this lecture, we learned about the `echo` keyword. Keywords are a feature in PHP for performing specific actions. If we type a keyword, the PHP interpreter understands that we're trying to perform a specific action based on that keyword.

The `echo` keyword allows us to render content onto the screen.

```
echo "Hello World!";
```

After the `echo` keyword, we can write our message in pair of quotes (`""`). Lastly, we must end our line of code with a semicolon character (`;`).

A statement is another way of describing a single instruction given to a machine or a single line of code. Programming languages give us the ability to communicate with a machine. Instructions can be given to a machine to perform various tasks, from sending an email to processing a transaction.

You can think of a statement as a single sentence in the English language. Multiple sentences can be combined to create a book. In a similar sense, multiple statements can be combined to create an application.

Multiple statements can be written like so.

```
echo "Hello World!";
echo "My name is John.";
```

Resources

- List of Keywords

Comments

In this lecture, we explored comments in PHP, which are similar to comments in HTML and CSS. They're notes that developers can add without PHP trying to process the contents of a comment.

There are three syntax options for writing comments.

```
// Single-line comment

/*
Multiline Comment
*/

# Single-line comment
```

If you were to run the above code in a PHP file, nothing would get outputted since comments don't get processed.

Variables

In this lecture, learned how to create variables. Variables were introduced for storing data, ranging from addresses to the price of a product. Here's an example of a variable.

```
$age;
```

Variables must always start with the `$` character. This character is followed by the name. The following rules must be adhered to for names to be valid:

- May contain alphabetic characters, numbers, and underscores.
- **MAY NOT** contain special characters. (i.e., **#**, **^**, **&**, **@**)
- The first letter may start with a letter or underscore. Numbers are not allowed.

In some cases, you may need to use multiple words in a variable name. Typically, there are three common naming conventions adopted by developers.

- **Camel Casing** - All words are capitalized except for this first word. Example: `$thisIsMyAge`
- **Pascal Casing** - All words are capitalized. Example: `$ThisIsMyAge`
- **Snake Casing** - All words are separated with an `_` character. Example: `$this_is_my_age`

Assignment Operator

In this lecture, we learned how to use the assignment operator to store a value in a variable. But what is an operator? Operators are symbols that accept a value to create/produce a new value.

The assignment operator will instruct PHP to store a value inside a variable. It's written with the `=` character. You can use it like so.

```
$age = 29;
```

We can reference a variable by typing the variable's name like so.

```
echo $age;
```

This outputs:

```
29
```

We have the option of updating a variable after creating it using the same syntax. PHP is smart enough to update a variable if it has already been created.

```
$age = 29;
$age = 30;

echo $age;
```

This outputs:

```
30
```

Keep in mind that variables are case-sensitive. If you make a typo when updating a variable, you may accidentally create a new variable. `$age` is completely different from `$Age`.

Resources

- Operators

Data Types

In this lecture, we learned about data types. Data types are categories for your data. Programming languages require data to be put into categories for easier debugging and optimization of your program. There are two types of systems seen across programming languages.

- **Statically-Typed** – Developers must explicitly set the type of a variable. The type may never change after declaration.
- **Dynamically-Typed** – Developers do not need to set the type. Data types may change at any time.

PHP is considered to be a dynamically-typed language. Dynamically-typed languages are easier for beginners to learn, but there are downsides. They can be slower since the programming language takes care of assigning the type. In addition, variables may not always have the same type throughout the lifetime of a program, which can cause unexpected behavior.

PHP has nine data types which are the following:

- `null`
- `bool`
- `int`
- `float`

- `string`
- `array`
- `object`
- `callable`
- `resource`

The var_dump() Function

In this lecture, we learned about functions, which are a feature for performing a specific set of instructions. An official list of functions can be found in the resource section.

To get started, we looked at the `var_dump()` function, which outputs the data type and value of a variable. All functions are written by typing their name, followed by a pair of parentheses. Inside the parentheses, we can pass on a value to the function.

```
var_dump(30);
```

This outputs:

```
int(30)
```

Resources

- List of Functions

Null Data Types

In this lecture, we learned about the `null` data type. This data type is used for variables that store nothing. It can be useful for variables that will be needed later, but you don't have a value to assign to them immediately.

You can assign a `null` value to a variable like so:

```
$data = null;
```

`null` is case-insensitive. You can also use `NULL`. Either solution is valid. Most developers prefer to use lowercase letters.

We can view the value and data type of a variable by using the `var_dump()` function like so:

```
var_dump($data);
```

This should output `NULL`.

Boolean Data Type

In this lecture, we learned about the boolean data type. They're useful for storing values that answer yes or no questions. You can set a variable to `true` or `false`. These values assign the `bool` data type to a variable.

```
$isHungry = true;
$hasPermission = false;
```

Alternatively, you can use uppercase letters like so: `TRUE` or `FALSE`. You'll notice that the variable names are questions. It's common practice for variables that store booleans to be named as questions.

Integers and Floats

In this lecture, we learned about integers and floats, which are data types for numbers. PHP introduces two data types for numbers because numbers with decimal values use more memory than whole numbers. To optimize your program, two separate data types were introduced to reduce memory usage.

The **integer** data type can store whole numbers. The **float** data type can store numbers with decimal values.

```
$age = 29; // integer
$price = 123.45 // float
```

Why is it called float? Since the decimal separator can be positioned anywhere in the number, it's considered to be a character that can float anywhere within a number.

Both data types support negative numbers.

```
$age = -29; // integer
$price = -123.45 // float
```

Lastly, both data types can have `_` characters for readability.

```
$distance = 5_000_00_0;
```

PHP will strip these characters away during runtime. They do not affect your program. You can insert them wherever you'd like. They're completely optional.

String Data Type

The string data type is available for storing text. The word "string" can be a weird name for a data type. If you think about it, a group of characters is strung together, hence the word string.

Strings are created with single quotes or double quotes.

```
$firstName = 'John';
$lastName = "Smith";
```

PHP supports string interpolation, which is the process of replacing a placeholder in a string with a variable. For example, we can do the following:

```
$firstName = 'John';
$lastName = "$firstName Smith";
```

This feature is only available for strings created with double quotes. In some cases, you may want to write text immediately after a placeholder. You can wrap the variable with curly brackets to prevent the text from being interpreted as part of the variable name.

```
$firstName = 'John';
$lastName = "{$firstName}™ Smith";
```

Lastly, we have the power to access a specific character of a string by using square brackets. PHP assigns an index to each character in a string. The first character has an index of 0, the second character has an index of 1, and so on and so forth.

To access the third character of a string, we can do the following:

```
var_dump($lastName[2]);
```

We can also update a specific character with this syntax.

```
$lastName[2] = "X";
```

Exercise: Data Type

This exercise is meant to be tackled on Udemy. Check out the resources for a link to the exercise on GitHub.

Resources

- Exercise PHP Data Types

Arrays

In this lecture, we learned about arrays. Arrays are useful for storing a collection of data. Arrays can be created by using a pair of `[]` as the value for a variable like so:

```
$food = ["Salad", "Burger", "Steak"];
```

Existing items can be accessed or updated by their index. Behind the scenes, PHP assigns a number to each item, starting from `0` and incrementing by 1 for each item. For example, we can update the second item in the array like so:

```
$food[1] = "Chicken"; // Change "Burger" to "Chicken"
```

New items can be added to an array by omitting a number from the square brackets. PHP will add the item to the end of the array.

```
$food[] = "Tomato Soup";
```

Associative Arrays

In this lecture, we learned how to assign names to keys in arrays. Instead of allowing PHP to assign numbers, you can override this behavior with custom names to better identify each item in array.

```
$food = [
  "john" => "Salad",
  "jane" => "Burger",
  "sam" => "Steak"
];
```

Associative arrays can be created using the `=>` character to separate the key and value. We can access a specific item from an associative array by the key name.

```
var_dump($food["john"]);
```

Multidimensional Arrays

In this lecture, we learned about multidimensional arrays. A multidimensional array is an array that contains more arrays. This allows you to create a nested structure of complex data. Here's an example:

```
$food = [
  "john" => ["Salad", "Curry"],
  "jane" => "Burger",
  "sam" => "Steak"
];
```

As you can see, the `john` key in the array is storing an additional array. We can access items from within a nested array by using square bracket syntax

```
var_dump($food["john"][1]);
```

In some cases, you may accidentally attempt to access an array from within an array that doesn't exist like so.

```
var_dump($food["bob"][1]);
```

In these cases, you may get the following error: **Trying to access array offset on value of type null.**

The error states that the inner array doesn't exist, so it can't access the item in the array. If you ever encounter this error, you should always double-check that you're accessing an existing array correctly.

Type Casting

In this lecture, we learned how to typecast a value, which is the process of changing a value from one datatype to another. The value itself may change to accommodate the datatype.

Converting a value into a boolean will always produce a `true` value unless the value is the following:

- the integer value `0`
- the float values `0.0` and `-0.0`
- an empty string
- the string `"0"`
- an array with zero elements
- the type `NULL`

Converting a value into an integer follows the current rules:

- A boolean `false` becomes `0`, and a boolean `true` becomes `1`.
- Floating point numbers will always be rounded toward zero.
- A string that is either fully numeric or leading numeric is converted to the corresponding numeric values. All other strings evaluate to zero.

- The value null is always converted to zero.

Converting a value into a float follows the same rules as integers with the addition of the following rules:

- A numeric or leading numeric string will resolve to the corresponding float value. All other strings will be evaluated to zero.
- All other types of values are converted first to integers and then to a float.

Converting a value into a string follows the current rules:

- A `false` boolean value becomes an empty string, and a `true` value becomes the string `"1"`.
- Integers and floats are converted to a textual representation of those numbers.
- All arrays are converted to the string `"Array"`.
- The value NULL is always converted to an empty string.

An array is used to store a bunch of elements to be accessed later. Arrays can contain zero, one, or more elements. Therefore, converting values of type integer, float, string, bool, and resource creates an array with a single element. The element can be accessed at the index zero within the new array. Casting a `NULL` into an array will give you an empty array.

Resources

- Typecasting Gist

Type Juggling

In this lecture, we learned that type juggling is a behavior in PHP where values are automatically type cast without our explicit permission. If we pass on the wrong data type, PHP may attempt to typecast the value automatically to the correct data type.

For example, the `echo` keyword expects a string. If we pass in an integer, the value is typecasted into a string.

```
echo 50;
```

PHP automatically performs this task, so we don't have to do the following:

```
echo (string) 50;
```

Resources

- Echo Keyword

Arithmetic Operators

In this lecture, we learned about arithmetic operators. These operators allow us to perform math. Here's an example of using the operators for addition, subtraction, multiplication, and division.

```
$data = 1 + 2 - 3 * 4 / 5;
```

PHP adheres to the order of operations. We can use parentheses to change the outcome of an equation like so:

```
$data = 1 + (2 - 3 * 4) / 5;
```

The modulo operator (`%`) performs division and returns the remainder.

```
$data = 11 % 2; // -> 1
```

The exponentiation operator (`**`) performs exponentiation where the value on the left of the operator is the base and the value on the right is the exponent.

```
$data = 10 ** 2; // -> 100
```

The Problem with Floats

Whenever you're working with floats, you come across situations where the float value is unexpected. This is a problem with most programming languages since dealing with decimal values is challenging.

Before PHP can execute our application, it must be compiled into machine code. Machine code is the only language machines are able to understand. Programming languages were introduced to be easier to read.

During this process, float values can become broken. There are ways to avoid this, which will be explored in future lectures.

Resources

- Arithmetic Operators
- Floating Guide

Assignment Operators

In this lecture, we explored different ways of assigning values to variables. PHP offers variations of arithmetic operators with the assignment operator. These operators allow you to apply mathematical operations to an existing value.

```
// Assignment Operators (= += -= *= /= %= **=)
$a = 10;
var_dump($a); // -> 10

$a += 2;      // $a = $a + 2;
var_dump($a); // -> 12

$a -= 2;      // $a = $a - 2;
var_dump($a); // -> 10

$a *= 10;     // $a = $a * 10;
var_dump($a); // -> 100

$a /= 10;     // $a = $a / 10;
var_dump($a); // -> 10

$a %= 6;      // $a = $a % 6;
var_dump($a); // -> 4

$a **= 4;    // $a = $a ** 4;
var_dump($a); // -> 256
```

Resources

- Assignment Operators

Comparison Operators

In this lecture, we learned about comparison operators. The comparison operators allow you to compare two values. PHP can compare values with different types.

If you were to use `==` or `!=`, the datatypes of a value are typecasted if they're not the same data type. Whereas the `===` and `!==` will not, which can cause a comparison to fail. In most cases, you should be strict with your comparisons to avoid headaches.

PHP also has two operators comparing values that aren't equal to each other, which are `!=` and `<>`. There are no differences between these operators.

```
// Comparison Operators (== === != <> !== < > <= =>)
var_dump(1 == "1"); // -> true
var_dump(1 === 1);  // -> true
var_dump(1 != 2);   // -> true
var_dump(1 <> 2);   // -> true
var_dump(1 !== "1"); // -> true
var_dump(1 < 2);    // -> true
var_dump(2 > 1);    // -> true
var_dump(1 <= 2);   // -> true
var_dump(2 >= 1);   // -> true
```

Resources

- Comparison Operators

Error Control Operators

In this lecture, we learned how to suppress errors. You can suppress errors with the error control operator (`@`). This operator should be used sparingly. PHP offers various features to help you avoid errors, so you shouldn't need this operator often.

```
// Error Control Operators
@var_dump((string) [1]);
```

Incrementing and Decrementing Numbers

In this lecture, we learned how to update numbers by one using the increment/decrement operators. The increment/decrement operators allow you to add/subtract a value by 1. You can position these operators before or after a value. If placed after a value, PHP will return a value before updating the value. Whereas placing it before the value will cause the value to be updated before returning it.

```
// Increment/Decrement Operators(++ --)
$a = 10;

var_dump($a++);
var_dump($a--);
```

Logical Operators

In this lecture, we learned how to support multiple conditions with logical operators. Logical operators allow us to chain multiple conditions. The `&&` or `and` operators allow us to verify that multiple conditions are truthy. The `||` or `or` operators allow only one condition to be true for the entire expression to evaluate to `true`. Lastly, the `!` operator allows us to check for a `false` value instead of a `true` value.

```
// Logical Operators (&& || ! and or)
var_dump(true && true); // -> true
var_dump(true and true); // -> true
var_dump(true || false); // -> true
var_dump(true or false); // -> true
var_dump(!true);        // -> false
```

Operator Precedence

In this lecture, we learned about operator precedence. If multiple operators are used within a single expression, PHP will prioritize specific operators over others. You can check out the link in the resource section of this lecture for the order.

For example, multiplication has higher precedence than addition.

```
$a = 5 + 2 * 10;
```

But what if operators have the same precedence, like multiplication and division.

```
$a = 5 / 2 * 10;
```

In this case, PHP uses associativity. It varies from operator to operator. However, it'll either start with the operator on the left or on the right to determine which operators have higher precedence.

Some operators have non-associativity. These operators don't allow you to use them within the same expression multiple times like the `<` and `>` operators.

```
$a = 5 < 2 > 8;
```

Lastly, you might encounter problems when using operators that have super lower precedence, like the `and` operator. PHP offers two operators for performing the same thing, which is the `&&` and `and` operators.

However, they have different precedences, which can lead to different results.

```
$a = true && false; // -> false
$a = true and false; // -> true
```

In the above example, the second expression evaluates to `true` even though the second condition fails, which should technically lead to a `false` value. However, the `=` operator has higher precedence, so PHP will assign `true` to the variable before checking the second condition.

This behavior can lead to unexpected results. For this reason, you should avoid the `and` and `or` operators. Instead, use their original variations, which are `&&` and `||`, respectively.

Resources

- Operator Precedence

Constants

In this lecture, we learned about another type of variable called constants. Constants are variables that can never change their value after being created. They can be created with the `const` keyword.

In PHP, we have flexibility with naming things. However, there is a set of reserved keywords we should avoid to prevent conflicts.

Here's how a constant is created.

```
const FULL_NAME = "John Smith";
```

The naming rules for variables apply to constants with the exception of adding the `$` character at the beginning of the name. It's common practice to use all uppercase letters so that it's easier to identify constants. Not required but recommended.

Resources

- Predefined Constants

String Concatenation

In this lecture, we learned how to combine two strings into a single string with a feature called string concatenation. Two strings can be combined with the concatenator operator, which is written with the `.` character.

```
const FULL_NAME = "John Smith";

var_dump("Hello, my name is" . FULL_NAME);
```

This feature can be helpful for inserting constants into a string since string interpolation does not support constants.

Section 3: Adding Logic

In this section of the course, we looked at features in PHP for using data to control the logic of our program.

Terminology: Expressions

In this lecture, we took the time to understand some programming terminology, which will be useful in upcoming lectures. Specifically, we learned about expression. **An expression is any line of code that produces a value. It's as simple as that.**

Resources

- Echo

Control Structures

In this lecture, we learned how to control the logic flow of our program with If statements. An if statement accepts an expression that must evaluate to a boolean value. If the value is `true`, a block of code is executed. Otherwise, nothing happens.

```
$score = 95;

if ($score > 90) {
  var_dump("A");
} else if ($score > 80) {
  var_dump("B");
} elseif ($score > 80) {
  var_dump("C");
} else {
  var_dump("F");
}
```

We can also add the `else if` / `elseif` keyword to perform an alternative condition. If prior conditions fail, the conditions from these statements will be checked. Once a condition passes, other conditions are ignored. An `else` statement can be added for default behavior. You can compare values in an expression.

Resources

- If Statement

Switch Statements

In this lecture, we learned an alternative solution to If statements called switch statements. Switch statements don't have as much flexibility as If statements. They only compare values that match. If a match is found, the code below it is executed.

A switch statement can be written with the `switch` keyword. With this keyword, we can provide a value to match. Afterward, we can add a list of values to compare with the value from the switch statement by using the `case` keyword.

```php
$paymentStatus = 1;

switch ($paymentStatus) {
  case 1:
    var_dump("Success");
    break;
  case 2:
    var_dump("Denied");
    break;
  default:
    var_dump("Unknown");
}
```

In this example, we're also adding the `break` keyword. Once PHP finds a match, it'll execute written below the case statement. This includes code from other case statements. To prevent that from happening, we can end the switch statement with the `break` keyword.

Something to keep in mind with switch statements is that data types do not have to match. For example, take the following:

```php
$paymentStatus = "2";

switch ($paymentStatus) {
  case 1:
    var_dump("Success");
    break;
  case 2:
    var_dump("Denied");
    break;
  default:
    var_dump("Unknown");
}
```

The `$paymentStatus` variable stores a string. However, the cases are integers. Regardless, the values can still match because PHP will typecast the values during comparison.

Match Expressions

In this lecture, we learned about match expressions, which are a feature to compare values and return the results. We can write a match expression with the `match` keyword. This keyword accepts the value to compare with a list of values inside a pair of curly brackets like so.

```php
$paymentStatus = 2;

$message = match ($paymentStatus) {
  1 => "Success",
  2 => "Denied",
  default => "Unknown"
};
```

We can add as many values to compare with. A default value can be added when a match can't be found with the `default` keyword. Match expressions must always return a value. Otherwise, PHP will throw an error. It's always considered good practice to have a default value.

Something to keep in mind is that comparisons are strict. The values must have the same value and data type. If the values are the same but different types the match won't be registered.

Functions

In this lecture, we learned about functions. Functions are blocks of code that are reusable. By using functions, we don't have to copy and paste the same solution over multiple files.

Functions can be defined with the `function` keyword followed by the name, pair of parentheses, and curly brackets. Function names follow the same rules as variable names, with the exception of the name starting with the `$` character.

Here's an example of the variable definition:

```php
function getStatus()
{
  $paymentStatus = 2;

  $message = match ($paymentStatus) {
    1 => "Success",
    2 => "Denied",
    default => "Unknown"
  };

  var_dump($message);
}
```

Functions can be invoked by writing the name with a pair of parentheses. Invoking a function is the process of running the code inside the function. By default, PHP does not execute the code unless it's invoked.

```
getStatus();
```

Function Parameters

In this lecture, we learned how to pass on data to a function by using parameters. Parameters are variables defined in a function's parentheses that can be updated when a function is called. For example, a parameter can be defined like so.

```
function getStatus($paymentStatus)
{
  // Some code...
}
```

We can pass on data to a function like so:

```
getStatus(1);
```

The `$paymentStatus` parameter will hold the value 1. If we do not pass on data, PHP will throw an error.

We can add additional parameters by separating each parameter with a comma. In addition, parameters may have optional values. If a function is not provided data, PHP will fall back to the default value.

```
function getStatus($paymentStatus, $showMessage = true)
{
  // Some code...
}
```

In some cases, you may hear the word **argument** to describe a parameter. Parameters and arguments can be interchangeable terms, but there is a difference between them. A parameter describes the variable in the function definition, whereas an argument describes the value itself passed into the function.

Function Return Values

In this lecture, we learned how to expose data outside of a function. Data defined inside a function is only available to a function. If we want to expose data outside the function, we must use the `return` keyword followed by the value.

```
function getStatus($paymentStatus, $showMessage = true)
{
  $message = match ($paymentStatus) {
    1 => "Success",
    2 => "Denied",
    default => "Unknown"
  };

  if ($showMessage) {
    var_dump($message);
  }

  return $message;
}
```

Keep in mind, PHP stops executing logic from within a function once a value is returned. Typically, it's advised to return a value at the end of the function.

The value returned by the function can be stored in a variable like so:

```
$statusMessage = getStatus(1);
```

This allows us to use the message returned by the function in other areas outside of the function. Returning values is optional but can be a great way to use data after a function has finished executing.

Type Hinting & Union Types

In this lecture, we discovered type hinting for enforcing data types in our function's parameters and return types. PHP can guess the data type of our variables, but we have the power to explicitly set the data type for debugging.

A function's data type can be set by adding a `:` followed by the type like so:

```
function getStatus($paymentStatus, $showMessage = true): string
```

In some cases, we may want to return null from our function. We can add the `?` character before the return type like so:

```
function getStatus($paymentStatus, $showMessage = true): ?string
```

Alternatively, our functions may not return data at all. In these cases, we can use the `void` data type, which was designed for functions that don't return anything.

```
function getStatus($paymentStatus, $showMessage = true): void
```

We're not limited to setting the return type. Parameters can be type hinted too. The data type must be added before the parameter name like so.

```
function getStatus(int $paymentStatus, bool $showMessage = true): ?string
```

Multiple data types can be assigned by separating them with the `|` character. These are known as union types.

```
function getStatus(int|float $paymentStatus, bool $showMessage = true): ?string
```

Lastly, if you don't want to chain data types and would rather accept any data type, you can use the `mixed` type.

```
function getStatus(mixed $paymentStatus, bool $showMessage = true): ?string
```

Strict Types

In this lecture, we enforced strict types for functions by using the `declare` directive. This keyword allows us to enable specific PHP features. PHP is a weakly typed language. Even if we use type hinting, that doesn't mean PHP will stop us from passing in a value with an invalid data type.

For debugging purposes, it's considered good practice to enable strict typing.

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

Adding this bit of code will apply strict typing to a specific file. If you have another PHP file in your application, that file won't have this feature enabled. You must add it to every file that you want strict types.

While Loop

In this lecture, we learned how to repeat a series of actions based on a condition with the `while` loop. The `while` loop accepts a condition. If the condition evaluates to `true`, a block of code gets executed. This process is repeated until the condition evaluates to `false`.

```
$a = 1;

while ($a <= 15) {
  echo $a;
}
```

We're using a new keyword called `echo`, which outputs a value. Unlike the `var_dump()` function, the datatype is not rendered with the value.

This code creates an infinite loop. Infinite loops consume resources, which can cause a system to crash. Since the `$a` variable is never updated, the condition evaluates to `true`. It's considered good practice to update the variable on each iteration.

```
while ($a <= 15) {
  echo $a;

  $a++;
}
```

We're using the `++` increment operator. This operator adds 1 to a value. This should prevent the loop from running infinitely.

In some cases, you may want to execute the block of code at least once. You can use the `do while` loop to achieve this behavior.

```
do {
  echo $a;
```

```
  $a++;
} while ($a <= 15);
```

The condition in the `while` keyword is performed **after** the block of code has been executed.

For Loop

In this lecture, we looked at an alternative to the `while` loop called the `for` loop. Similar to the `while` loop, we can use a condition to determine when a loop should run. However, rather than partially writing the logic pertaining to the loop outside of the parentheses, everything is centralized in one location.

```
for ($i = 1; $i <= 15; $i++) {
  echo $i;
}
```

There are three steps for `for` loops.

1. Initializer - A variable that can be initialized for iterating through the loop.
2. Condition - An expression that evaluates to a boolean. If `true`, the block of code is executed.
3. Increment - An expression that should update the variable from the first step.

PHP always executes the initializer once before checking the condition. Afterward, it'll execute the block of code. After the block of code finishes running, the third expression is executed. PHP checks the condition and repeats the process.

In some cases, you might want to skip an iteration or stop the loop completely. You can do so with the `continue` or `break` keywords, respectively.

```
for ($i = 1; $i <= 15; $i++) {
  if ($i == 6) {
    continue;
  }

  echo $i;
}
```

A new comparison operator was introduced called `==`. This operator compares two values for equality. In this example, we're checking if the `$i` variable is equal to `6`. If it is, the current iteration is skipped. PHP will not echo the number and move on to the next iteration.

If you want to stop the loop altogether, you can use the `break` keyword.

Foreach Loop

In this lecture, we learned about the `foreach` loop, which allows us to loop through an array. This loop accepts the array to loop through, followed by the `as` keyword and a variable to store a reference to each item in the array.

```
$names = ["John", "Jane", "Sam"];

foreach ($names as $name) {
  var_dump($name);
}
```

In some cases, you may want the key to the current item. You can update the expression to include a variable for the key like so.

```
$names = ["John", "Jane", "Sam"];

foreach ($names as $key => $name) {
  var_dump($key);
  var_dump($name);
}
```

Short-Circuiting

In this lecture, we learned about a feature called short-circuiting. PHP will not bother checking other conditions if a previous condition evaluates to `false`. This increases performance since PHP does not perform additional logic.

Take the following example:

```
function example()
{
  echo "example() invoked";

  return true;
```

```
}

var_dump(false && example());
```

Since the first condition is `false`, the `example()` function is never executed. If you were to run the above code, you would not see the message from the function appear in the output.

Section 4: Beginner Challenges

In this section, we tried a few challenges to get us to start thinking like a developer.

Getting Started with Challenges

This lecture does not have any notes. The content is meant to be consumed via video.

Coding Exercise: Resistor Colors

If you want to build something using a Raspberry Pi, you'll probably use resistors. For this exercise, you need to know two things about them:

- Each resistor has a resistance value.
- Resistors are small - so small, in fact, that if you printed the resistance value on them, it would be hard to read.

To get around this problem, manufacturers print color-coded bands onto the resistors to denote their resistance values. Each band has a position and a numeric value.

The first 2 bands of a resistor have a simple encoding scheme: each color maps to a single number.

In this exercise, you are going to create a helpful program so that you don't have to remember the values of the bands.

These colors are encoded as follows:

- black: 0
- brown: 1
- red: 2
- orange: 3
- yellow: 4
- green: 5
- blue: 6
- violet: 7
- grey: 8
- white: 9

The goal of this exercise is to create a way to look up the numerical value associated with a particular color band. Mnemonics map the colors to the numbers, that, when stored as an array, happen to map to their index in the array.

More information on the color encoding of resistors can be found in the Electronic color code Wikipedia article.

Starter Code

```
<?php

// 1. Create function for accepting the color as a string.
// 2. Create an array resistor colors with their relative numeric code
// 3. Return a numeric code based on the color's name

function colorCode($color)
{

}
```

Coding Solution: Resistor Colors

```
<?php

// 1. Create function for accepting the color as a string.
// 2. Create an array resistor colors with their relative numeric code
// 3. Return a numeric code based on the color's name

declare(strict_types=1);

function colorCode(string $color): int
{
```

```
    $colors = [
      "black" => 0,
      "brown" => 1,
      "red" => 2,
      "orange" => 3,
      "yellow" => 4,
      "green" => 5,
      "blue" => 6,
      "violet" => 7,
      "grey" => 8,
      "white" => 9
    ];

    return $colors[$color];
}
```

Coding Exercise: Two Fer

Two-fer or 2-fer is short for two for one. One for you and one for me.

Given a name, return a string with the message:

```
One for name, one for me.
```

Where "name" is the given name.

However, if the name is missing, return the string:

```
One for you, one for me.
```

Here are some examples:

- Alice = `One for Alice, one for me.`
- Bob = `One for Bob, one for me.`
- `One for you, one for me.`
- Zaphod = `One for Zaphod, one for me.`

Starter Code

```
<?php

declare(strict_types=1);

function twoFer(string $name): string
{

}
```

Coding Solution: Two Fer

```
<?php

// 1. Update $name parameter to be optional by adding a default value.
// 2. Return a string using string interpolation to inject the $name variable into the phrase.

declare(strict_types=1);

function twoFer(string $name = "you"): string
{
    return "One for {$name}, one for me.";
}
```

Coding Exercise: Leap Year

Given a year, report if it is a leap year.

The tricky thing here is that a leap year in the Gregorian calendar occurs:

- on every year that is evenly divisible by 4
- except every year that is evenly divisible by 100
- unless the year is also evenly divisible by 400

For example, 1997 is not a leap year, but 1996 is. 1900 is not a leap year, but 2000 is.

For a delightful, four minute explanation of the whole leap year phenomenon, go watch this youtube video.

Starter Code

```php
<?php

declare(strict_types=1);

function isLeap(int $year): bool
{

}
```

Coding Solution: Leap Year

```php
<?php

// 1. Check if year isn't evenly divisible by 4. If so, return false.
// 2. Check if year is evenly divisible by 100 AND
// isn't evenly divisible by 400. If so, return false.
// 3. return true if all conditions fail.

declare(strict_types=1);

function isLeap(int $year): bool
{
  if ($year % 4 !== 0) {
    return false;
  }

  if ($year % 100 === 0 && $year % 400 !== 0) {
    return false;
  }

  return true;
}
```

Section 5: Filling in the Gaps

In this section, we covered a wide variety of topics in PHP that were missing from the previous 4 sections.

Predefined Constants

In this lecture, we explored some predefined constants by PHP. These are some of the more common ones.

- `PHP_VERSION` - Outputs the currently installed PHP version.
- `PHP_INT_MAX` - The largest integer supported in this build of PHP.
- `PHP_INT_MIN` - The smallest integer supported in this build of PHP.
- `PHP_FLOAT_MAX` - Largest representable floating point number.
- `PHP_FLOAT_MIN` - Smallest representable positive floating point number.

In addition, PHP has magic constants, which are constants that have dynamic values but cannot be updated by us. The most common magic constants are the following:

- `__LINE__` - The current line number of the file.
- `__FILE__` - The full path and filename of the file with symlinks are resolved. If used inside an include, the name of the included file is returned.
- `__DIR__` - The directory of the file. If used inside an include, the directory of the included file is returned. This is equivalent to dirname(**FILE**). This directory name does not have a trailing slash unless it is the root directory.

Resources

- Predefined Constants
- Magic Constants

Alternative Syntax for Constants

In this lecture, we learned how to define constants with the `define()` function. It has two parameters, which are the name of the constant and the value. Here's an example.

```php
define("FOO", "Hello World!");
```

```
echo FOO;
```

Other than how the constant is created, using the constant is similar to any other constant.

The main difference between the `define()` function and `const` keyword is that the `define()` function can be used in a conditional statement, whereas the `const` keyword cannot.

```
if (!defined("FOO")) {
  define("FOO", "Hello World!");
}
```

It's common practice to verify that a constant has been created by using the `defined()` function, which accepts the name of the constant to check. If the constant isn't created, we can proceed to create one.

Unsetting Variables

In this lecture, we learned how to release memory from our program by using the `unset()` function. This function accepts the variable to delete from your program. For example.

```
$name = "John";

unset($name);

echo $name; // <~ Throws undefined error
```

In the above example, the `$name` variable gets defined, unset, and then throws an error whenever we try to reference the variable afterward since it no longer exists in our program. PHP throws an undefined error.

In addition, you can remove specific items from an array using this method like so.

```
$names = ["John", "Jane", "Alice"];

unset($names[1]);
```

This will remove the second item from the array but does not reindex the array. If you want to reindex the array, you can use the `array_values` function.

```
$names = array_values($names);
```

You can verify the array was reindexed by using the `print_r()` function.

```
print_r($names);
```

This function outputs the items in array. It's different from the `var_dump()` function since it does not output the data types of each item in the array.

Reading the PHP Documentation

In this lecture, we explored reading the documentation for PHP. There are no notes for this lecture.

Resources

- PHP Manual

Rounding Numbers

In this lecture, we learned how to round numbers. PHP offers three functions for rounding numbers. The first of which is `floor()` and `ceil()`. The `floor()` function rounds a number down, and the `ceil()` function rounds a number up. Both round to the nearest whole number.

```
echo floor(4.5); // Result: 4
echo ceil(4.5); // Result 5
```

The `round()` function allows us to round a number while also adding precision to keep a certain number of digits in the decimal portion of a value. It accepts the number to round, precision, and whether to round up or down when the decimal value is halfway through to the next number.

```
echo round(4.455, 2); // Result: 4.6
```

Resources

- ceil()
- floor()
- round()

Alternative if statement syntax

In this lecture, we learned about an alternative syntax for writing conditional statements. Instead of curly brackets, we can use a `:` character after the condition to denote the beginning of a block of code. The ending of the block of code can be written with the `endif` keyword.

```
<?php $permission = 1; ?>

<?php if ($permission === 1) : ?>
  <h1>Hello Admin</h1>
<?php elseif ($permission === 2) : ?>
  <h1>Hello Mod</h1>
<?php else: ?>
  <h1>Hello Guest</h1>
<?php endif; ?>
```

Another difference between using curly brackets and keywords is that the `else if` keyword can't be used. We must stick with the `elseif` keyword.

Avoiding Functions in Conditions

In this lecture, we learned that you should avoid using functions in conditions whenever possible. Some functions can take a while to execute. If you're executing an expensive function multiple times because of a condition, this can severely affect the overall performance of your app.

For example, take the following:

```
<?php
function getPermission() {
  sleep(2);

  return 2;
}
?>

<?php if (getPermission() === 1) : ?>
  <h1>Hello Admin</h1>
<?php elseif (getPermission() === 2) : ?>
  <h1>Hello Mod</h1>
<?php else: ?>
  <h1>Hello Guest</h1>
<?php endif; ?>
```

In this example, the `getPermission()` function takes two seconds to execute cause of the `sleep()` function. We're calling it twice from both conditions, which can cause our application to take longer to respond.

A better solution would be to outsource the value returned by our function in a variable so that it's only called once.

```
<?php
function getPermission() {
  sleep(2);

  return 2;
}

$permission = getPermission();

?>

<?php if ($permission === 1) : ?>
  <h1>Hello Admin</h1>
<?php elseif ($permission === 2) : ?>
  <h1>Hello Mod</h1>
<?php else: ?>
  <h1>Hello Guest</h1>
<?php endif; ?>
```

Loops

This problem can also affect loops, too.

```php
function getUsers() {
  sleep(2);

  return ["John", "Jane"];
}

for ($i = 0; $i < count(getUsers()); $i++) {
  echo $i;
}
```

In this example, the `getUsers()` function is called on each iteration of the loop, which needs 2 seconds to finish executing.

To fix this, we can outsource the value returned by the function in a variable called `$userCount` and use that variable from within the condition like so.

```php
function getUsers() {
  sleep(2);

  return ["John", "Jane"];
}

$userCount = count(getUsers());
for ($i = 0; $i < $userCount; $i++) {
  echo $i;
}
```

Including PHP Files

In this lecture, we learned how to include code from other files by using the `include`, `include_once`, `require`, and `require_once` keywords. Each of these keywords can include a PHP file in another PHP file.

```php
include "example.php";
include_once "example.php";
require "example.php";
require_once "example.php";
```

The main difference between these keywords is that the `include` keyword(s) throw a warning if a file can't be found. Warnings don't interrupt the rest of the script from running.

The `require` keyword(s) throws a fatal error, which does stop the rest of the script from running. The keywords with the word `_once` only include a file once. If a file has been included before, it will not be included again.

Variadic Functions

A variadic function is a function that accepts an unlimited number of arguments. You can define a variadic function by adding the `...` operator before the parameter name like so.

```php
function sum (int|float ...$num) {
  return array_sum($num);
}
```

In this example, the `sum()` function accepts an unlimited set of numbers. All values are stored in an array called `$num`. Lastly, we're calculating the sum using the `array_sum()` function.

Named Arguments

In this lecture, we learned how to assign values to specific parameters by using named arguments. By default, PHP assigns values to parameters in the order they're presented. In some cases, we may want to set specific values to specific parameters like so.

```php
someFunction(x: 10, y, 5);
```

We can add the name of the parameter before the value with a `:` separating the parameter name and value. So, even if the order does not match, PHP won't have a problem assigning the value to the correct parameter.

Keep in mind, this feature is only available in PHP 8 and up.

Resources

- setcookies() Function

Global Variables

In this lecture, we learned how to use global variables. By default, functions do not have access to variables defined outside of the function. If we want to use a variable, we must use the `global` keyword followed by a list of variables we'd like to use. Multiple variables can be specified by separating them with a comma.

Here's an example:

```
$x = 5;

function foo() {
  global $x;

  echo $x;
}

foo();
```

Global variables can be convenient, but developers often avoid them. It's recommended to pass in the value to the function and have the function return a new value. Global variables can make your application behave unreliably.

Static Variables

In this lecture, we learned about static variables. Static variables are a feature that allows variables defined in functions to retain their values after a function has finished running.

By default, variables defined in a function are destroyed after a function is finished running. As a result, the value gets reset when the function is called again.

By adding the `static` keyword to a variable, the variable will retain the value like so.

```
function foo() {
  static $x = 1;
  return $x++;
}

echo foo() . "<br>"; // Outputs: 1
echo foo() . "<br>"; // Outputs: 2
echo foo() . "<br>"; // Outputs: 3
```

Anonymous and Arrow Functions

In this lecture, we learned about anonymous functions, which are functions without names. Typically, anonymous functions are assigned to variables or passed into functions that accept anonymous functions.

We can define an anonymous function like so.

```
$multiply = function ($num) {
  return $num * $multiplier;
};
```

By storing it in a variable, we can pass on this function as an argument to another function. For example:

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

echo sum(10, 5, $multiply);
```

We're referring to the function as `$callback`. Callback is a common naming convention for functions that get passed into another function that can be called at a later time.

In some cases, you may want to access variables in the parent scope. You can do so with the `use` keyword after the parameter list.

```
$multiplier = 2;
$multiply = function ($num) use ($multiplier) {
  return $num * $multiplier;
};
```

Unlike global variables, PHP creates a unique copy of the variable, so the original variable does not get modified if you attempt to update the value.

A shorter way of writing anonymous functions is to use arrow functions. Here's the same example as an arrow function.

```
$multiply = fn ($num) => $num * $multiplier;
```

Unilke before, we don't have to use the `use` keyword since all variables in the parent scope are accessible to the arrow function. In addition, the code to the right of the `=>` character is treated as an expression. The value evaluated by the expression is the return value.

Keep in mind, arrow functions cannot have a body. You may only write an expression, and arrow functions always return a value. Regular anonymous functions can optionally return values.

Callable Type

In this lecture, we learned about the `callable` type, which is the data type you can add to a parameter that holds a function. Here's an example:

```
function sum (int|float $a, int|float $b, callable $callback) {
  return $callback($a + $b);
}
```

You have a few options when passing in a function. You can either pass in a variable that holds the function, write the function directly, or pass in the name of a regular function.

```
echo sum(10, 5, $someFunction);
echo sum(10, 5, fn($num) => $num * 2);
echo sum(10, 5, 'someOtherFunction');
```

Passing by Reference

In this lecture, we learned how to pass in a variable by reference. If we were to add the `&` character before the name of the parameter, PHP does not create a unique copy of the value and assign it to the parameter.

Instead, PHP will allow us to reference the original variable through the parameter. If we modify the reference, the original variable gets modified too.

```
$cup = "empty";

function fillCup(&$cupParam) {
  $cupParam = "filled";
}

fillCup($cup);

echo $cup; // Output: filled
```

In this example, the `$cup` variable will be set to `filled` since the value gets manipulated from within the `fillCup()` function.

Array Functions

In this lecture, we explored various functions for manipulating arrays. PHP offers a wide assortment of functions. Here are some of the most common ones you may use from time to time.

The `isset()` and `array_key_exists()` functions can be used to verify that a specific key in a function has a value. However, the `isset()` function does not consider `null` or `false` values to be acceptable.

Take the following:

```
$users = ["John", "Jane", "Bob", null];

if (isset($users[3])) {
  echo "User found!";
}
```

In this example, the `isset()` function will return `false` since `$users[3]` contains a `null` value. Despite having a value, this index is considered to be empty. If we want the condition to pass for arrays that have `null` values, we can use the `array_key_exists()` function.

```
$users = ["John", "Jane", "Bob", null];

if (array_key_exists(3, $users)) {
  echo "User found!";
}
```

Another function worth looking at is the `array_filter()` function, which will allow us to filter the items in the array. By default, this function removes empty values in an array.

```
$users = ["John", "Jane", "Bob", null];

$users = array_filter($users);
```

In the example above, the `array_filter()` function returns the `$users` array without the `null` value. If we want to customize the filtering, we can pass in a callback function. This callback function will be given each item in the array and can return a boolean to determine if the value should remain in the array.

```
$users = ["John", "Jane", "Bob", null];

$users = array_filter($users, fn($user) => $user !== "Bob");
```

In this example, `"Bob"` gets filtered out of the array.

The `array_map()` function can be used to modify each item in the array. It has two arguments, which are the array and a callback function that will be given each item in the array. The callback function must return the value or a modified version of the value.

```
$users = ["John", "Jane", "Bob", null];

$users = array_map($users, fn($user) => strtoupper($user));
```

In this example, each user is run through the `strtoupper()` function to convert lowercase letters into uppercase.

The `array_merge()` function can be used to merge two or more arrays into a single array.

```
$users = ["John", "Jane", "Bob", null];

$users = array_merge($users, ["Sam", "Jessica"]);
```

In this example, the second array will be appended to the array creating the following output:

```
["John", "Jane", "Bob", null, "Sam", "Jessica"]
```

We can also sort arrays using PHP's various sorting functions. Refer to the resource section for a link to a complete list of sorting functions.

The first function you'll often use is the `sort()` function, which sorts an array in ascending order.

```
sort($users);
```

Keep in mind, the `sort()` function does not return an array. Instead, it references an array and can modify the existing array.

Another thing to know is that the `sort()` function reindexes your array. If you wish to keep the original indexes, you can use the `asort()` function instead.

```
asort($users);
```

Resources

- Array Functions
- array_filter() Function
- array_merge() - Function
- sort() Function
- Sorting Arrays

Destructuring Arrays

In this lecture, we learned how to destructure an array to extract specific values. There are two ways to grab items from an array. Firstly, we can use the `list` keyword like so.

```
$numbers = [10, 20];

list($a, $b) = $numbers;

echo $a;
```

PHP extracts the values in the order they're defined. So, the `$a` variable will have the value of `10`.

A shorthand syntax is available by using `[]`.

```
[$a, $b] = $numbers;
```

In addition, we can grab named keys from an array like so.

```
$numbers = ["example" => 10, 20];

["example" => $a, 0 => $b ] = $numbers;
```

Working with Files

In this lecture, we explored the various functions for working with file data. The first function we looked at is the `scandir()` function.

```
scandir(__DIR__);
```

This function returns a complete list of files and directories in a specific directory. You can use `__DIR__` constant to check the current directory you're in.

In almost all cases, PHP will also add the following items to the array of files/directories.

- `.` - The current directory.
- `..` - The parent directory.

In addition, we looked at the `mkdir()` and `rmdir()` functions for creating and removing directories, respectively. Both functions accept the directory to create/delete.

```
mkdir("foo");
rmdir("foo");
```

Before working with a file, you should always verify that it exists by using the `file_exists()` function, which accepts the path to a file. It'll return a boolean on whether the file was found or not.

```
file_exists("example.txt");
```

The file size can be retrieved with the `filesize()` function. This function also accepts a path to the file.

```
filesize("example.txt");
```

Data can be written to a file by using the `file_put_contents()` function. This function accepts the name of the file and the data to insert into the file.

```
file_put_contents("example.txt", "hello world");
```

Sometimes, you may find yourself reading file info from the same file multiple times. Behind the scenes, PHP caches the results of some of its filesystem functions. If you want to clear the cache, you can use the `clearstatcache()` function.

Lastly, you can read the contents of a file by using the `file_get_contents()` function.

```
echo file_get_contents("example.txt");
```

Resources

- Filesystem Functions

Section 6: More PHP Challenges

In this section, we tried a few challenges to get us to start thinking like a developer.

Exploring the Challenges

This lecture does not have any notes. The content is meant to be consumed via video.

Coding Exercise: Robot Name

Manage robot factory settings. When a robot comes off the factory floor, it has no name.

The first time you turn on a robot, a random name is generated in the format of two uppercase letters followed by three digits, such as **RX837** or **BC811**.

Create a function that generates names with this format. The names must be random: they should not follow a predictable sequence. 🤖

Hints

Here are the steps you can take to complete this exercise:

1. Generate an array of characters A - Z. Check out the links below for a function to use.
2. Randomize the items in the array. Check out the links below for a function to use.
3. Generate a random number between 100 - 999. Check out the links below for a function to use.
4. Return a string with the first two items from the array and a random number.

Check out these functions for helping you curate a solution to this challenge:

- range() - For generating letters A - Z
- shuffle() - For randomizing the letters.
- mt_rand() - For generating a random number between 100 - 999

Starter Code

```php
<?php

declare(strict_types=1);

function getNewName(): string
{

}
```

Coding Solution: Robot Name

```php
<?php

declare(strict_types=1);

function getNewName(): string
{
    $letters = range("A", "Z");
    shuffle($letters);

    $number = mt_rand(100, 999);

    return "{$letters[0]}{$letters[1]}{$number}";
}
```

Coding Exercise: Armstrong Numbers

An Armstrong number is a number that is the sum of its own digits, each raised to the power of the number of digits.

For example:

- 9 is an Armstrong number, because `9 = 9^1 = 9`
- 10 is not an Armstrong number, because `10 != 1^2 + 0^2 = 1`
- 153 is an Armstrong number, because: `153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153`
- 154 is not an Armstrong number, because: `154 != 1^3 + 5^3 + 4^3 = 1 + 125 + 64 = 190`

Write some code to determine whether a number is an Armstrong number.

Warning: Please do not use arrow functions. They're not supported on Udemy.

Hints

Here are the steps you can take to complete this exercise:

1. Convert the string into an array with each character as an individual value. Check out the links below for a function to use.
2. Loop through the array. Check out the links below for a function to use.

3. Calculate exponentiation with the number in the current iteration with the number of items in the array. Check out the links below for a function to use.
4. Calculate the sum of all values in the array and check if it equals the original number. Check out the links below for a function to use.
5. Return the result as a boolean

Check out these functions to help you curate a solution to this challenge:

- str_split() - To help you grab each digit in the number
- array_map() - To help you go through each number
- count() - To help you count the items in the array
- array_sum() - To help you calculate the sum of all numbers in an array

Starter Code

```php
<?php

declare(strict_types=1);

function isArmstrongNumber(int $number): bool
{

}
```

Coding Solution: Armstrong Numbers

```php
<?php

declare(strict_types=1);

function isArmstrongNumber(int $number): bool
{
    $digits = str_split((string) $number);

    $digits = array_map(function($digit) use ($digits) {
      return $digit ** count($digits);
    }, $digits);

    return array_sum($digits) === $number;
}
```

Coding Exercise: Series

Given a string of digits, output all the contiguous substrings of length n in that string in the order that they appear.

For example, the string "49142" has the following 3-digit series:

- "491"
- "914"
- "142"

And the following 4-digit series:

- "4914"
- "9142"

And if you ask for a 6-digit series from a 5-digit string OR if you ask for less than 1 digit, return an empty array.

Note that these series are only required to occupy adjacent positions in the input; the digits need not be *numerically consecutive*.

Hints

Here are the steps you can take to complete this exercise:

1. Create a conditional statement to check if length of the string is longer than the requested size or smaller than 1.
2. Return an empty array if either condition is true.
3. Otherwise, create an array to store the results.
4. Start looping through series minus the size.
5. Insert a sub-string into the results
6. Return the results

Check out these functions to help you curate a solution to this challenge:

- Google: "PHP How to get the number of characters in a string"
- substr() - To help you get a portion of a string

Starter Code

```
<?php

declare(strict_types=1);

function slices(string $series, int $size) : array
{

}
```

Coding Solution: Series

```
<?php

declare(strict_types=1);

function slices(string $series, int $size) : array
{
    $seriesLength = strlen($series);

    if ($size > $seriesLength || $size < 1) {
        return [];
    }

    $results = [];

    for ($i = 0; $i <= $seriesLength - $size; $i++) {
        $results[] = substr($series, $i, $size);
    }

    return $results;
}
```

Section: 7: Object-Oriented Programming

In this section, we started exploring the basics of object-oriented programming in PHP.

What is OOP (Object-Oriented Programming)?

In this lecture, we talked about programming paradigms and how they apply to OOP. There are no notes for this lecture.

Classes

In this lecture, we learned about classes. Classes are blueprints for creating objects. An object can be anything we want. We can use an object to represent a UI element or a database entry.

Classes can be defined with the `class` keyword.

```
class Account {

}
```

It's standard practice to name your class after the name of your file and to use pascalcasing.

Since classes are just blueprints, we can't interact with the data inside of a class until it has been instantiated. Instantiation is the process of creating an object from a class. We can instantiate a class using the `new` keyword like so.

```
$myAccount = new Account;
$johnsAccount = new Account;
```

We can even create multiple instances.

Properties

In this lecture, we talked about properties, which are variables defined inside a class. We must define a property by using an access modifier followed by the name of the variable.

```
class Account {
  public $name;
```

```
  public $balance;
}
```

An access modifier determines how accessible the property is outside of the class. The `public` keyword allows for a property to be updated anywhere from our script.

We can access the property using the `->` from the respective instance.

```
$myAccount = new Account;

$myAccount->balance = 10;

var_dump($myAccount->balance);
```

Data types can be applied to a property after the access modifier.

```
public string $name;
public float $balance;
```

If we don't assign a value to a property, the value will be `uninitialized`. We can set properties to any type of value, but we can't use complex operations, such as setting a property to a function.

```
public float $balance = intval(5.5); // Not allowed
```

Resources

- Access Modifiers

Magic Methods

In this lecture, we learned about magic methods. A method is the terminology for a function defined inside a class. Magic methods are functions defined inside a class that is automatically called by PHP.

There are various magic methods available, but the most common method is `__construct()`. Most magic methods begin with two `_` characters. It's recommended to avoid using two `_` characters in your own methods to reduce confusion.

```
class Account
{
  public string $name;
  public float $balance;

  public function __construct(
    string $newName,
    float $newBalance
  ) {
    $this->name = $newName;
    $this->balance = $newBalance;
  }
}
```

In this example, the `__construct()` method has the `public` access modifier. This allows for our method to be accessible anywhere. Whenever we create a new instance of the `Account` class, the `__construct()` method gets called.

We can pass on data to this method like so.

```
$myAccount = new Account("John", 100);
```

Resources

- Magic Methods

Constructor Property Promotion

In this lecture, we learned of a shorter way of defining properties and setting their values from within a `__construct()` method. If we add the access modifier to the parameter, PHP will automatically treat the parameter as a class property and set the value passed into the method as the value for the respective parameter.

The solution from the previous lecture can be shortened to this:

```
class Account
{
  public function __construct(
    public string $name,
    public float $balance
  ) { }
}
```

It'll result in the same behavior as before.

Null-safe Operator

In this lecture, we talked about the null-safe operator, which allows us to access a property method without worrying about PHP throwing an error because the instance does not exist. If we attempt to access a property or method on a `null` value, PHP throws an error.

```
$myAccount = null;

$myAccount?->deposit(500);
```

The null-safe operator will check if the value is `null` before accessing a property or method. If a variable is `null`, PHP will not bother running the next portion of code, which prevents an error from being thrown.

Understanding Namespaces

In this lecture, we talked about namespaces. There are no notes for this lecture.

Creating a Namespace

In this lecture, we created a namespace by using the `namespace` keyword. Following this keyword, we must provide a name. Namespaces follow the same naming rules for classes or functions. Generally, most developers use pascalcasing for namespaces.

```
namespace App;

class Account {
  // Some code here...
}
```

We can access a class from within a namespace by typing the namespace and class like so.

```
$myAccount = new App\Account();
```

However, you may not want to type out the entire path. You can use the `use` keyword to import a class from a namespace. This will allow you to create an instance with just the class name.

```
use App\Account;

$myAccount = new Account();
```

Working with Namespaces

In this lecture, we went over various things to be aware of when using namespaces. Firstly, it's common practice for the namespace to mimick a project's directory. For example, if a class exists in a folder called `App`. The namespace should be called `App`. Not required but recommended to make it easier to find classes.

The second thing to be aware of is that we don't need to use the `use` statement or type out the complete namespace when classes exist in the same namespace.

Another thing to know is that PHP will only resolve classes to the current namespace. For example, let's say we wanted to use a class called `DateTime` from a file in the `App` namespace.

```
new DateTime();
```

PHP will throw an error because it'll attempt to look for the class with the following path: `App\DateTime`. In order to use a class from a different namespace, such as the global namespace, we must add the `\` character before the classname like so.

```
new \DateTime();
```

Alternatively, we can import the class with the `use` keyword without the `\` character.

```
use DateTime;

new DateTime();
```

It's important that the `use` statement to be in the same file as where we're using the class. The `use` statement only applies to the current file.

Lastly, we can add aliases for classes with the `as` keyword.

```
use DateTime as DT;

new DT();
```

Resources

- Name Resolution Rules
- Namespace FAQ

Autoloading Classes

In this lecture, we learned how to automate the process of loading PHP files with classes. There's a function called `spl_autoload_register()` that we can call for intercepting classes that are used in a program, but haven't been defined.

We can pass in a function that will be responsible for loading a specific class, which is passed on to our function as a parameter.

```
spl_autoload_register(function($class) {
  $formattedClass = str_replace("\\", "/", $class);
  $path = "{$formattedClass}.php";
  require $path;
});
```

In the above example, the first thing we're doing is replacing the `\` character with a `/` character since the `require` statement does not allow `\` characters. To accomplish this, we're using the `str_replace()` function, which accepts the string to search for in a given string, the string to replace it with, and the string to search.

In the first argument, we're escaping the `\` character by using `\\` since the `\` character escapes a string. Escaping a string is the process of telling PHP not to close the string with the closing double-quote.

Next, we created a variable called `$path`, which contains the path to the file that contains the class. Lastly, we use the `require` keyword to load the file.

Using Constants in Classes

In this lecture, we used a constant in a class. Classes do not support the `define()` function for constants. We must use the `const` keyword like so.

```
class Account {
  public const INTEREST_RATE = 2;
}
```

Next, we can access a constant by using the scope resolution operator (`::`).

```
Account::INTEREST_RATE;
```

Alternatively, you can also access a constant from an instance of a class.

```
$myAccount = new Account();
$myAccount::INTEREST_RATE;
```

Static Properties and Methods

In this lecture, we learned how to use static properties and methods. Static properties are similar to constants, except they can have their values modified. We can define a static property with the `static` keyword. This keyword can be positioned before or after an access modifier.

```
class Account {
  public static int $count = 0;
```

```
}
```

We can access the static property like so.

```
Account::$count;
```

Alternatively, you can also access a static property from an instance of a class.

```
$myAccount = new Account();
$myAccount::$count;
```

Typically, static properties are avoided since they can be altered anywhere, just like global variables. On the other hand, static methods are popular as a means of creating utility methods. Take the following:

```
class Utility {
  public static function printArray(array $array) {
    echo '<pre>';
    print_r($array);
    echo '</pre>';
  }
}
```

Since the method is static, we can invoke the method without requiring an instance like so.

```
Utility::printArray([1, 2, 3]);
```

OOP Principle: Encapsulation

In this lecture, we learned about an object-oriented programming principle called encapsulation. Encapsulation is the idea of restricting data and methods to a single class.

In your classes, you can prevent outside sources from modifying properties by using the `private` access modifier.

```
public function __construct(
  private string $name,
  private float $balance
) { }
```

By changing the modifier to `private`, only the current class can modify a given property.

But what if we want to read or update the property? In this case, we can create getter or setter methods, which are just methods that can be used for accessing a specific property. Here's an example of a getter.

```
public function getBalance() {
  return "$" . $this->balance;
}
```

It's common practice to set the name of the method to the word "get" followed by the name of the property to grab. One of the main benefits of getters is being able to format values before returning them.

As for setters, they're methods for updating a property. It's standard practice to set the name to the word "set" followed by the name of the property.

```
public function setBalance(float $newBalance) {
  if ($newBalance < 0) return;

  $this->balance = $newBalance;
}
```

The main benefit of setters is that we can validate data before updating a property. Overall, getters and setters give us more power over how our properties are read and set.

We're not limited to private properties. Methods can be private too. This can be useful when you want to subdivide a method into multiple actions without exposing a method outside of a class.

```
public function setBalance(float $newBalance) {
  if ($newBalance < 0) return;

  $this->balance = $newBalance;
  $this->sendEmail();
```

```
}

private function sendEmail() {}
```

In the following example, the `sendEmail()` method is private, which will be responsible for sending an email to the user if the balance is successfully updated.