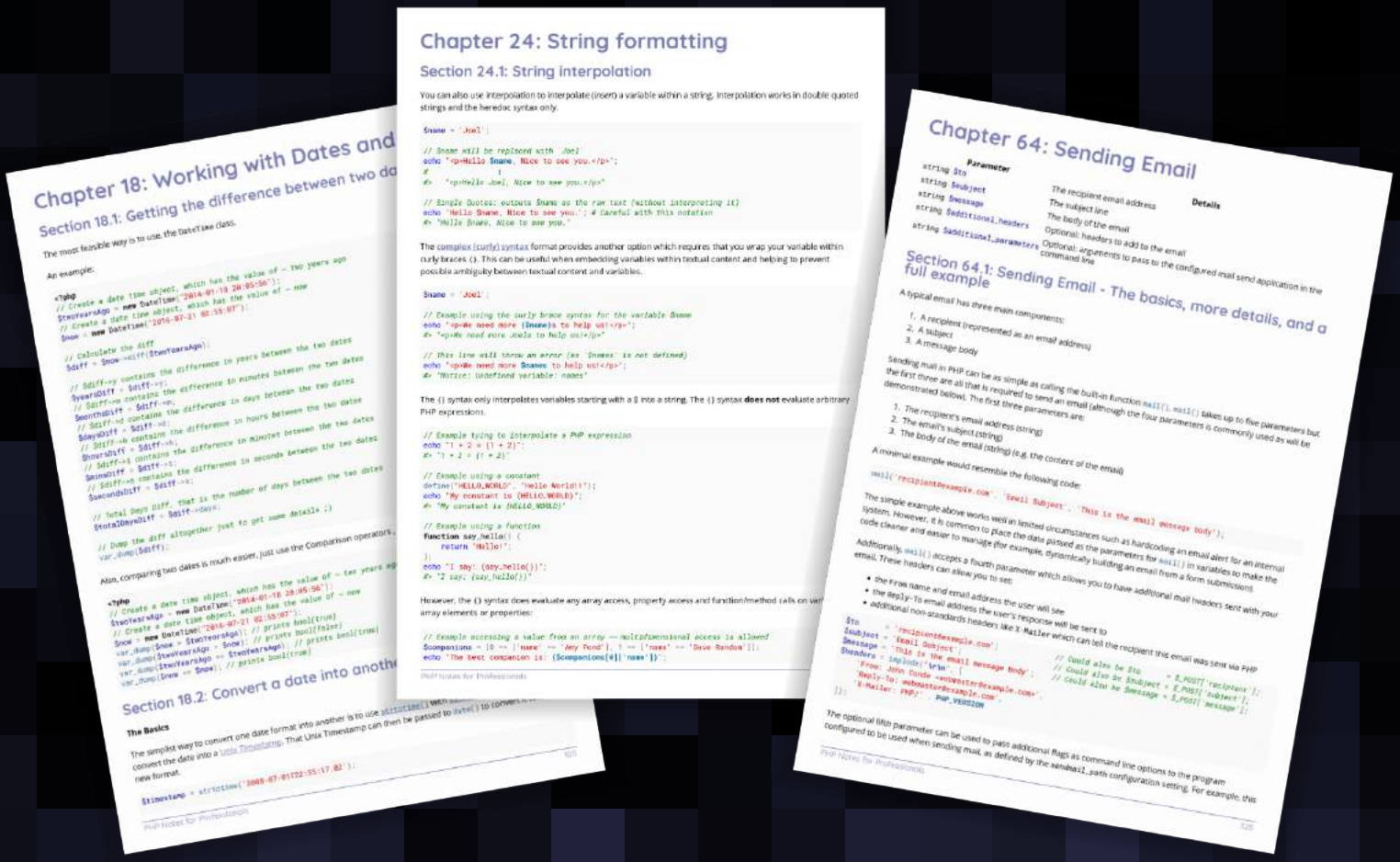
PHP

Notes for Professionals



400+ pages

of professional hints and tricks

Contents

About 1
Chapter 1: Getting started with PHP 2
Section 1.1: HTML output from web server 2
Section 1.2: Hello, World! 3
Section 1.3: Non-HTML output from web server 3
Section 1.4: PHP built-in server 5
Section 1.5: PHP CLI 5
Section 1.6: Instruction Separation 6
Section 1.7: PHP Tags 7
Chapter 2: Variables 9
Section 2.1: Accessing A Variable Dynamically By Name (Variable variables) 9
Section 2.2: Data Types 10
Section 2.3: Global variable best practices 13
Section 2.4: Default values of uninitialized variables 14
Section 2.5: Variable Value Truthiness and Identical Operator 15
Chapter 3: Variable Scope 18
Section 3.1: Superglobal variables 18
Section 3.2: Static properties and variables 18
Section 3.3: User-defined global variables 19
Chapter 4: Superglobal Variables PHP 21
Section 4.1: Suberglobals explained 21
Section 4.2: PHP5 SuperGlobals 28
Chapter 5: Outputting the Value of a Variable 32
Section 5.1: echo and print 32
Section 5.2: Outputting a structured view of arrays and objects 33
Section 5.3: String concatenation with echo 35
Section 5.4: printf vs sprintf 36
Section 5.5: Outputting large integers 36
Section 5.6: Output a Multidimensional Array with index and value and print into the table 37
Chapter 6: Constants 39
Section 6.1: Defining constants 39
Section 6.2: Class Constants 40
Section 6.3: Checking if constant is defined 40
Section 6.4: Using constants 42
Section 6.5: Constant arrays 42
Chapter 7: Magic Constants 43
Section 7.1: Difference between __FUNCTION__ and __METHOD__ 43
Section 7.2: Difference between __CLASS__, get_class() and get_called_class() 43
Section 7.3: File & Directory Constants 44
Chapter 8: Comments 45
Section 8.1: Single Line Comments 45
Section 8.2: Multi Line Comments 45
Chapter 9: Types 46
Section 9.1: Type Comparison 46
Section 9.2: Boolean 46
Section 9.3: Float 47

Section 9.4: Strings ... 48
Section 9.5: Callable ... 50
Section 9.6: Resources ... 50
Section 9.7: Type Casting ... 51
Section 9.8: Type Juggling ... 51
Section 9.9: Null ... 52
Section 9.10: Integers ... 52
Chapter 10: Operators ... 54
Section 10.1: Null Coalescing Operator (??) ... 54
Section 10.2: Spaceship Operator (<=>) ... 55
Section 10.3: Execution Operator (``) ... 55
Section 10.4: Incrementing (++) and Decrementing Operators (--) ... 55
Section 10.5: Ternary Operator (?:) ... 56
Section 10.6: Logical Operators (&&/AND and ||/OR) ... 57
Section 10.7: String Operators (. and .=) ... 57
Section 10.8: Object and Class Operators ... 57
Section 10.9: Combined Assignment (+= etc) ... 59
Section 10.10: Altering operator precedence (with parentheses) ... 59
Section 10.11: Basic Assignment (=) ... 60
Section 10.12: Association ... 60
Section 10.13: Comparison Operators ... 60
Section 10.14: Bitwise Operators ... 62
Section 10.15: instanceof (type operator) ... 64
Chapter 11: References ... 67
Section 11.1: Assign by Reference ... 67
Section 11.2: Return by Reference ... 67
Section 11.3: Pass by Reference ... 68
Chapter 12: Arrays ... 71
Section 12.1: Initializing an Array ... 71
Section 12.2: Check if key exists ... 73
Section 12.3: Validating the array type ... 74
Section 12.4: Creating an array of variables ... 74
Section 12.5: Checking if a value exists in array ... 74
Section 12.6: ArrayAccess and Iterator Interfaces ... 75
Chapter 13: Array iteration ... 79
Section 13.1: Iterating multiple arrays together ... 79
Section 13.2: Using an incremental index ... 80
Section 13.3: Using internal array pointers ... 80
Section 13.4: Using foreach ... 81
Section 13.5: Using ArrayObject Iterator ... 83
Chapter 14: Executing Upon an Array ... 84
Section 14.1: Applying a function to each element of an array ... 84
Section 14.2: Split array into chunks ... 85
Section 14.3: Imploding an array into string ... 86
Section 14.4: "Destructuring" arrays using list() ... 86
Section 14.5: array_reduce ... 86
Section 14.6: Push a Value on an Array ... 87
Chapter 15: Manipulating an Array ... 89
Section 15.1: Filtering an array ... 89
Section 15.2: Removing elements from an array ... 90

Section 15.3: Sorting an Array ... 91
Section 15.4: Whitelist only some array keys ... 96
Section 15.5: Adding element to start of array ... 96
Section 15.6: Exchange values with keys ... 97
Section 15.7: Merge two arrays into one array ... 97
Chapter 16: Processing Multiple Arrays Together ... 99
Section 16.1: Array intersection ... 99
Section 16.2: Merge or concatenate arrays ... 99
Section 16.3: Changing a multidimensional array to associative array ... 100
Section 16.4: Combining two arrays (keys from one, values from another) ... 100
Chapter 17: Datetime Class ... 102
Section 17.1: Create Immutable version of DateTime from Mutable prior PHP 5.6 ... 102
Section 17.2: Add or Subtract Date Intervals ... 102
Section 17.3: getTimestamp ... 102
Section 17.4: setDate ... 103
Section 17.5: Create DateTime from custom format ... 103
Section 17.6: Printing DateTimes ... 103
Chapter 18: Working with Dates and Time ... 105
Section 18.1: Getting the difference between two dates / times ... 105
Section 18.2: Convert a date into another format ... 105
Section 18.3: Parse English date descriptions into a Date format ... 107
Section 18.4: Using Predefined Constants for Date Format ... 107
Chapter 19: Control Structures ... 109
Section 19.1: if else ... 109
Section 19.2: Alternative syntax for control structures ... 109
Section 19.3: while ... 109
Section 19.4: do-while ... 110
Section 19.5: goto ... 110
Section 19.6: declare ... 110
Section 19.7: include & require ... 111
Section 19.8: return ... 112
Section 19.9: for ... 112
Section 19.10: foreach ... 113
Section 19.11: if elseif else ... 113
Section 19.12: if ... 114
Section 19.13: switch ... 114
Chapter 20: Loops ... 116
Section 20.1: continue ... 116
Section 20.2: break ... 117
Section 20.3: foreach ... 118
Section 20.4: do...while ... 118
Section 20.5: for ... 119
Section 20.6: while ... 120
Chapter 21: Functions ... 121
Section 21.1: Variable-length argument lists ... 121
Section 21.2: Optional Parameters ... 122
Section 21.3: Passing Arguments by Reference ... 123
Section 21.4: Basic Function Usage ... 124
Section 21.5: Function Scope ... 124

Chapter 22: Functional Programming 125
Section 22.1: Closures 125
Section 22.2: Assignment to variables 126
Section 22.3: Objects as a function 126
Section 22.4: Using outside variables 127
Section 22.5: Anonymous function 127
Section 22.6: Pure functions 128
Section 22.7: Common functional methods in PHP 128
Section 22.8: Using built-in functions as callbacks 129
Section 22.9: Scope 129
Section 22.10: Passing a callback function as a parameter 129
Chapter 23: Alternative Syntax for Control Structures 131
Section 23.1: Alternative if/else statement 131
Section 23.2: Alternative for statement 131
Section 23.3: Alternative while statement 131
Section 23.4: Alternative foreach statement 131
Section 23.5: Alternative switch statement 132
Chapter 24: String formatting 133
Section 24.1: String interpolation 133
Section 24.2: Extracting/replacing substrings 134
Chapter 25: String Parsing 136
Section 25.1: Splitting a string by separators 136
Section 25.2: Substring 136
Section 25.3: Searching a substring with strpos 138
Section 25.4: Parsing string using regular expressions 139
Chapter 26: Classes and Objects 140
Section 26.1: Class Constants 140
Section 26.2: Abstract Classes 142
Section 26.3: Late static binding 144
Section 26.4: Namespacing and Autoloading 145
Section 26.5: Method and Property Visibility 147
Section 26.6: Interfaces 149
Section 26.7: Final Keyword 152
Section 26.8: Autoloading 153
Section 26.9: Calling a parent constructor when instantiating a child 154
Section 26.10: Dynamic Binding 155
Section 26.11: $this, self and static plus the singleton 156
Section 26.12: Defining a Basic Class 159
Section 26.13: Anonymous Classes 160
Chapter 27: Namespaces 162
Section 27.1: Declaring namespaces 162
Section 27.2: Referencing a class or function in a namespace 162
Section 27.3: Declaring sub-namespaces 163
Section 27.4: What are Namespaces? 164
Chapter 28: Sessions 165
Section 28.1: session_start() Options 165
Section 28.2: Session Locking 165
Section 28.3: Manipulating session data 166
Section 28.4: Destroy an entire session 166

Section 28.5: Safe Session Start With no Errors 167
Section 28.6: Session name 167
Chapter 29: Cookies 169
Section 29.1: Modifying a Cookie 169
Section 29.2: Setting a Cookie 169
Section 29.3: Checking if a Cookie is Set 170
Section 29.4: Removing a Cookie 170
Section 29.5: Retrieving a Cookie 170
Chapter 30: Output Buffering 171
Section 30.1: Basic usage getting content between buffers and clearing 171
Section 30.2: Processing the buffer via a callback 171
Section 30.3: Nested output buffers 172
Section 30.4: Running output buffer before any content 173
Section 30.5: Stream output to client 174
Section 30.6: Using Output buffer to store contents in a file, useful for reports, invoices etc 174
Section 30.7: Typical usage and reasons for using ob_start 174
Section 30.8: Capturing the output buffer to re-use later 175
Chapter 31: JSON 177
Section 31.1: Decoding a JSON string 177
Section 31.2: Encoding a JSON string 180
Section 31.3: Debugging JSON errors 183
Section 31.4: Using JsonSerializable in an Object 184
Section 31.5: Header json and the returned response 185
Chapter 32: SOAP Client 187
Section 32.1: WSDL Mode 187
Section 32.2: Non-WSDL Mode 187
Section 32.3: Classmaps 187
Section 32.4: Tracing SOAP request and response 188
Chapter 33: Using cURL in PHP 190
Section 33.1: Basic Usage (GET Requests) 190
Section 33.2: POST Requests 190
Section 33.3: Using Cookies 191
Section 33.4: Using multi_curl to make multiple POST requests 192
Section 33.5: Sending multi-dimensional data and multiple files with CurlFile in one request 193
Section 33.6: Creating and sending a request with a custom method 196
Section 33.7: Get and Set custom http headers in php 196
Chapter 34: Reflection 198
Section 34.1: Feature detection of classes or objects 198
Section 34.2: Testing private/protected methods 198
Section 34.3: Accessing private and protected member variables 200
Chapter 35: Dependency Injection 202
Section 35.1: Constructor Injection 202
Section 35.2: Setter Injection 202
Section 35.3: Container Injection 204
Chapter 36: XML 205
Section 36.1: Create a XML using DomDocument 205
Section 36.2: Read a XML document with DOMDocument 206
Section 36.3: Leveraging XML with PHP's SimpleXML Library 207
Section 36.4: Create an XML file using XMLWriter 209

Section 36.5: Read a XML document with SimpleXML ... 210
Chapter 37: SimpleXML ... 212
Section 37.1: Loading XML data into simplexml ... 212
Chapter 38: Parsing HTML ... 213
Section 38.1: Parsing HTML from a string ... 213
Section 38.2: Using XPath ... 213
Section 38.3: SimpleXML ... 213
Chapter 39: Regular Expressions (regexp/PCRE) ... 215
Section 39.1: Global RegExp match ... 215
Section 39.2: String matching with regular expressions ... 216
Section 39.3: Split string into array by a regular expression ... 217
Section 39.4: String replacing with regular expression ... 217
Section 39.5: String replace with callback ... 217
Chapter 40: Traits ... 219
Section 40.1: What is a Trait? ... 219
Section 40.2: Traits to facilitate horizontal code reuse ... 220
Section 40.3: Conflict Resolution ... 221
Section 40.4: Implementing a Singleton using Traits ... 222
Section 40.5: Traits to keep classes clean ... 223
Section 40.6: Multiple Traits Usage ... 224
Section 40.7: Changing Method Visibility ... 224
Chapter 41: Composer Dependency Manager ... 226
Section 41.1: What is Composer? ... 226
Section 41.2: Autoloading with Composer ... 227
Section 41.3: Difference between 'composer install' and 'composer update' ... 227
Section 41.4: Composer Available Commands ... 228
Section 41.5: Benefits of Using Composer ... 229
Section 41.6: Installation ... 230
Chapter 42: Magic Methods ... 231
Section 42.1: __call() and __callStatic() ... 231
Section 42.2: __get(), __set(), __isset() and __unset() ... 232
Section 42.3: __construct() and __destruct() ... 233
Section 42.4: __toString() ... 234
Section 42.5: __clone() ... 235
Section 42.6: __invoke() ... 235
Section 42.7: __sleep() and __wakeup() ... 236
Section 42.8: __debugInfo() ... 236
Chapter 43: File handling ... 238
Section 43.1: Convenience functions ... 238
Section 43.2: Deleting files and directories ... 240
Section 43.3: Getting file information ... 240
Section 43.4: Stream-based file IO ... 242
Section 43.5: Moving and Copying files and directories ... 244
Section 43.6: Minimize memory usage when dealing with large files ... 245
Chapter 44: Streams ... 246
Section 44.1: Registering a stream wrapper ... 246
Chapter 45: Type hinting ... 248
Section 45.1: Type hinting classes and interfaces ... 248
Section 45.2: Type hinting scalar types, arrays and callables ... 249

Section 45.3: Nullable type hints ... 250
Section 45.4: Type hinting generic objects ... 251
Section 45.5: Type Hinting No Return(Void) ... 252
Chapter 46: Filters & Filter Functions ... 253
Section 46.1: Validating Boolean Values ... 253
Section 46.2: Validating A Number Is A Float ... 253
Section 46.3: Validate A MAC Address ... 254
Section 46.4: Sanitze Email Addresses ... 254
Section 46.5: Sanitize Integers ... 255
Section 46.6: Sanitize URLs ... 255
Section 46.7: Validate Email Address ... 256
Section 46.8: Validating A Value Is An Integer ... 256
Section 46.9: Validating An Integer Falls In A Range ... 257
Section 46.10: Validate a URL ... 257
Section 46.11: Sanitize Floats ... 259
Section 46.12: Validate IP Addresses ... 261
Section 46.13: Sanitize filters ... 262
Chapter 47: Generators ... 263
Section 47.1: The Yield Keyword ... 263
Section 47.2: Reading a large file with a generator ... 264
Section 47.3: Why use a generator? ... 264
Section 47.4: Using the send()-function to pass values to a generator ... 265
Chapter 48: UTF-8 ... 267
Section 48.1: Input ... 267
Section 48.2: Output ... 267
Section 48.3: Data Storage and Access ... 267
Chapter 49: Unicode Support in PHP ... 269
Section 49.1: Converting Unicode characters to “\uxxxx” format using PHP ... 269
Section 49.2: Converting Unicode characters to their numeric value and/or HTML entities using PHP ... 269
Section 49.3: Intl extention for Unicode support ... 271
Chapter 50: URLs ... 272
Section 50.1: Parsing a URL ... 272
Section 50.2: Build an URL-encoded query string from an array ... 272
Section 50.3: Redirecting to another URL ... 273
Chapter 51: How to break down an URL ... 275
Section 51.1: Using parse_url() ... 275
Section 51.2: Using explode() ... 276
Section 51.3: Using basename() ... 276
Chapter 52: Object Serialization ... 278
Section 52.1: Serialize / Unserialize ... 278
Section 52.2: The Serializable interface ... 278
Chapter 53: Serialization ... 280
Section 53.1: Serialization of different types ... 280
Section 53.2: Security Issues with unserialize ... 281
Chapter 54: Closure ... 284
Section 54.1: Basic usage of a closure ... 284
Section 54.2: Using external variables ... 284
Section 54.3: Basic closure binding ... 285

Section 54.4: Closure binding and scope 285
Section 54.5: Binding a closure for one call 287
Section 54.6: Use closures to implement observer pattern 287
Chapter 55: Reading Request Data 290
Section 55.1: Reading raw POST data 290
Section 55.2: Reading POST data 290
Section 55.3: Reading GET data 290
Section 55.4: Handling file upload errors 291
Section 55.5: Passing arrays by POST 291
Section 55.6: Uploading files with HTTP PUT 293
Chapter 56: Type juggling and Non-Strict Comparison Issues 294
Section 56.1: What is Type Juggling? 294
Section 56.2: Reading from a file 294
Section 56.3: Switch surprises 295
Section 56.4: Strict typing 296
Chapter 57: Sockets 298
Section 57.1: TCP client socket 298
Section 57.2: TCP server socket 299
Section 57.3: UDP server socket 299
Section 57.4: Handling socket errors 300
Chapter 58: PDO 301
Section 58.1: Preventing SQL injection with Parameterized Queries 301
Section 58.2: Basic PDO Connection and Retrieval 302
Section 58.3: Database Transactions with PDO 303
Section 58.4: PDO: connecting to MySQL/MariaDB server 305
Section 58.5: PDO: Get number of affected rows by a query 306
Section 58.6: PDO::lastInsertId() 306
Chapter 59: PHP MySQLi 308
Section 59.1: Close connection 308
Section 59.2: MySQLi connect 308
Section 59.3: Loop through MySQLi results 309
Section 59.4: Prepared statements in MySQLi 309
Section 59.5: Escaping Strings 310
Section 59.6: Debugging SQL in MySQLi 311
Section 59.7: MySQLi query 311
Section 59.8: How to get data from a prepared statement 312
Section 59.9: MySQLi Insert ID 314
Chapter 60: SQLite3 316
Section 60.1: SQLite3 Quickstart Tutorial 316
Section 60.2: Querying a database 317
Section 60.3: Retrieving only one result 318
Chapter 61: Using MongoDB 319
Section 61.1: Connect to MongoDB 319
Section 61.2: Get multiple documents - find() 319
Section 61.3: Get one document - findOne() 319
Section 61.4: Insert document 319
Section 61.5: Update a document 319
Section 61.6: Delete a document 320
Chapter 62: mongo-php 321

Section 62.1: Everything in between MongoDB and Php 321
Chapter 63: Using Redis with PHP 324
Section 63.1: Connecting to a Redis instance 324
Section 63.2: Installing PHP Redis on Ubuntu 324
Section 63.3: Executing Redis commands in PHP 324
Chapter 64: Sending Email 325
Section 64.1: Sending Email - The basics, more details, and a full example 325
Section 64.2: Sending HTML Email Using mail() 327
Section 64.3: Sending Email With An Attachment Using mail() 328
Section 64.4: Sending Plain Text Email Using PHPMailer 329
Section 64.5: Sending HTML Email Using PHPMailer 330
Section 64.6: Sending Email With An Attachment Using PHPMailer 331
Section 64.7: Sending Plain Text Email Using Sendgrid 331
Section 64.8: Sending Email With An Attachment Using Sendgrid 332
Chapter 65: Using SQLSRV 333
Section 65.1: Retrieving Error Messages 333
Section 65.2: Fetching Query Results 333
Section 65.3: Creating a Connection 334
Section 65.4: Making a Simple Query 334
Section 65.5: Invoking a Stored Procedure 334
Section 65.6: Making a Parameterised Query 335
Chapter 66: Command Line Interface (CLI) 336
Section 66.1: Handling Program Options 336
Section 66.2: Argument Handling 337
Section 66.3: Input and Output Handling 338
Section 66.4: Return Codes 339
Section 66.5: Restrict script execution to command line 339
Section 66.6: Behavioural differences on the command line 339
Section 66.7: Running your script 340
Section 66.8: Edge Cases of getopt() 340
Section 66.9: Running built-in web server 341
Chapter 67: Localization 343
Section 67.1: Localizing strings with gettext() 343
Chapter 68: Headers Manipulation 344
Section 68.1: Basic Setting of a Header 344
Chapter 69: Coding Conventions 345
Section 69.1: PHP Tags 345
Chapter 70: Asynchronous programming 346
Section 70.1: Advantages of Generators 346
Section 70.2: Using Icicle event loop 346
Section 70.3: Spawning non-blocking processes with proc_open() 347
Section 70.4: Reading serial port with Event and DIO 348
Section 70.5: HTTP Client Based on Event Extension 350
Section 70.6: HTTP Client Based on Ev Extension 353
Section 70.7: Using Amp event loop 357
Chapter 71: How to Detect Client IP Address 359
Section 71.1: Proper use of HTTP_X_FORWARDED_FOR 359
Chapter 72: Create PDF files in PHP 361
Section 72.1: Getting Started with PDFlib 361

Chapter 73: YAML in PHP ... 362
Section 73.1: Installing YAML extension ... 362
Section 73.2: Using YAML to store application configuration ... 362
Chapter 74: Image Processing with GD ... 364
Section 74.1: Image output ... 364
Section 74.2: Creating an image ... 365
Section 74.3: Image Cropping and Resizing ... 366
Chapter 75: Imagick ... 369
Section 75.1: First Steps ... 369
Section 75.2: Convert Image into base64 String ... 369
Chapter 76: SOAP Server ... 371
Section 76.1: Basic SOAP Server ... 371
Chapter 77: Machine learning ... 372
Section 77.1: Classification using PHP-ML ... 372
Section 77.2: Regression ... 373
Section 77.3: Clustering ... 375
Chapter 78: Cache ... 377
Section 78.1: Caching using memcache ... 377
Section 78.2: Cache Using APC Cache ... 378
Chapter 79: Autoloading Primer ... 380
Section 79.1: Autoloading as part of a framework solution ... 380
Section 79.2: Inline class definition, no loading required ... 380
Section 79.3: Manual class loading with require ... 381
Section 79.4: Autoloading replaces manual class definition loading ... 381
Section 79.5: Autoloading with Composer ... 382
Chapter 80: SPL data structures ... 383
Section 80.1: SplFixedArray ... 383
Chapter 81: IMAP ... 387
Section 81.1: Connecting to a mailbox ... 387
Section 81.2: Install IMAP extension ... 388
Section 81.3: List all folders in the mailbox ... 388
Section 81.4: Finding messages in the mailbox ... 389
Chapter 82: HTTP Authentication ... 391
Section 82.1: Simple authenticate ... 391
Chapter 83: WebSockets ... 392
Section 83.1: Simple TCP/IP server ... 392
Chapter 84: BC Math (Binary Calculator) ... 394
Section 84.1: Using bcmath to read/write a binary long on 32-bit system ... 394
Section 84.2: Comparison between BCMath and float arithmetic operations ... 395
Chapter 85: Docker deployment ... 397
Section 85.1: Get docker image for php ... 397
Section 85.2: Writing dockerfile ... 397
Section 85.3: Building image ... 397
Section 85.4: Starting application container ... 398
Chapter 86: APCu ... 399
Section 86.1: Iterating over Entries ... 399
Section 86.2: Simple storage and retrieval ... 399
Section 86.3: Store information ... 399

Chapter 87: PHP Built in server ... 400
Section 87.1: Running the built in server ... 400
Section 87.2: built in server with specific directory and router script ... 400
Chapter 88: PSR ... 401
Section 88.1: PSR-4: Autoloader ... 401
Section 88.2: PSR-1: Basic Coding Standard ... 402
Chapter 89: PHPDoc ... 403
Section 89.1: Describing a variable ... 403
Section 89.2: Adding metadata to functions ... 403
Section 89.3: Describing parameters ... 404
Section 89.4: Collections ... 405
Section 89.5: Adding metadata to files ... 406
Section 89.6: Inheriting metadata from parent structures ... 406
Chapter 90: Design Patterns ... 408
Section 90.1: Method Chaining in PHP ... 408
Chapter 91: Compile PHP Extensions ... 410
Section 91.1: Compiling on Linux ... 410
Chapter 92: Common Errors ... 411
Section 92.1: Call fetch_assoc on boolean ... 411
Section 92.2: Unexpected $end ... 411
Chapter 93: Compilation of Errors and Warnings ... 413
Section 93.1: Parse error: syntax error, unexpected T_PAAMAYIM_NEKUDOTAYIM ... 413
Section 93.2: Notice: Undefined index ... 413
Section 93.3: Warning: Cannot modify header information - headers already sent ... 413
Chapter 94: Exception Handling and Error Reporting ... 415
Section 94.1: Setting error reporting and where to display them ... 415
Section 94.2: Logging fatal errors ... 415
Chapter 95: Debugging ... 417
Section 95.1: Dumping variables ... 417
Section 95.2: Displaying errors ... 417
Section 95.3: phpinfo() ... 418
Section 95.4: Xdebug ... 418
Section 95.5: Error Reporting (use them both) ... 419
Section 95.6: phpversion() ... 419
Chapter 96: Unit Testing ... 420
Section 96.1: Testing class rules ... 420
Section 96.2: PHPUnit Data Providers ... 423
Section 96.3: Test exceptions ... 426
Chapter 97: Performance ... 428
Section 97.1: Profiling with Xdebug ... 428
Section 97.2: Memory Usage ... 429
Section 97.3: Profiling with XHProf ... 430
Chapter 98: Multiprocessing ... 432
Section 98.1: Multiprocessing using built-in fork functions ... 432
Section 98.2: Creating child process using fork ... 432
Section 98.3: Inter-Process Communication ... 433
Chapter 99: Multi Threading Extension ... 434
Section 99.1: Getting Started ... 434

Section 99.2: Using Pools and Workers 434
Chapter 100: Secure Remeber Me 436
Section 100.1: “Keep Me Logged In” - the best approach 436
Chapter 101: Security 437
Section 101.1: PHP Version Leakage 437
Section 101.2: Cross-Site Scripting (XSS) 437
Section 101.3: Cross-Site Request Forgery 439
Section 101.4: Command Line Injection 440
Section 101.5: Stripping Tags 441
Section 101.6: File Inclusion 442
Section 101.7: Error Reporting 442
Section 101.8: Uploading files 443
Chapter 102: Cryptography 446
Section 102.1: Symmetric Encryption and Decryption of large Files with OpenSSL 446
Section 102.2: Symmetric Cipher 448
Chapter 103: Password Hashing Functions 449
Section 103.1: Creating a password hash 449
Section 103.2: Determine if an existing password hash can be upgraded to a stronger algorithm 450
Section 103.3: Verifying a password against a hash 451
Chapter 104: Contributing to the PHP Manual 452
Section 104.1: Improve the official documentation 452
Section 104.2: Tips for contributing to the manual 452
Chapter 105: Contributing to the PHP Core 453
Section 105.1: Setting up a basic development environment 453
Appendix A: Installing a PHP environment on Windows 454
Section A.1: Download, Install and use WAMP 454
Section A.2: Install PHP and use it with IIS 454
Section A.3: Download and Install XAMPP 455
Appendix B: Installing on Linux/Unix Environments 458
Section B.1: Command Line Install Using APT for PHP 7 458
Section B.2: Installing in Enterprise Linux distributions (CentOS, Scientific Linux, etc) 458
Credits 460
You may also like 468

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/PHPBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with PHP

PHP 7.x

Version	Supported Until	Release Date
7.1	2019-12-01	2016-12-01
7.0	2018-12-03	2015-12-03

PHP 5.x

Version	Supported Until	Release Date
5.6	2018-12-31	2014-08-28
5.5	2016-07-21	2013-06-20
5.4	2015-09-03	2012-03-01
5.3	2014-08-14	2009-06-30
5.2	2011-01-06	2006-11-02
5.1	2006-08-24	2005-11-24
5.0	2005-09-05	2004-07-13

PHP 4.x

Version	Supported Until	Release Date
4.4	2008-08-07	2005-07-11
4.3	2005-03-31	2002-12-27
4.2	2002-09-06	2002-04-22
4.1	2002-03-12	2001-12-10
4.0	2001-06-23	2000-05-22

Legacy Versions

Version	Supported Until	Release Date
3.0	2000-10-20	1998-06-06
2.0		1997-11-01
1.0		1995-06-08

Section 1.1: HTML output from web server

PHP can be used to add content to HTML files. While HTML is processed directly by a web browser, PHP scripts are executed by a web server and the resulting HTML is sent to the browser.

The following HTML markup contains a PHP statement that will add `Hello World!` to the output:

```
<!DOCTYPE html>
<html>
    <head>
        <title>PHP!</title>
    </head>
    <body>
        <p><?php echo "Hello world!"; ?></p>
    </body>
</html>
```

When this is saved as a PHP script and executed by a web server, the following HTML will be sent to the user's browser:

```
<!DOCTYPE html>
<html>
    <head>
```

```
        <title>PHP!</title>
    </head>
    <body>
        <p>Hello world!</p>
    </body>
</html>
```

PHP 5.x `Version ≥ 5.4`

`echo` also has a shortcut syntax, which lets you immediately print a value. Prior to PHP 5.4.0, this short syntax only works with the short_open_tag configuration setting enabled.

For example, consider the following code:

```
<p><?= "Hello world!" ?></p>
```

Its output is identical to the output of the following:

```
<p><?php echo "Hello world!"; ?></p>
```

In real-world applications, all data output by PHP to an HTML page should be properly *escaped* to prevent XSS (Cross-site scripting) attacks or text corruption.

See also: Strings and PSR-1, which describes best practices, including the proper use of short tags (`<?= ... ?>`).

Section 1.2: Hello, World!

The most widely used language construct to print output in PHP is `echo`:

```
echo "Hello, World!\n";
```

Alternatively, you can also use `print`:

```
print "Hello, World!\n";
```

Both statements perform the same function, with minor differences:

- `echo` has a `void` return, whereas `print` returns an `int` with a value of `1`
- `echo` can take multiple arguments (without parentheses only), whereas `print` only takes one argument
- `echo` is slightly faster than `print`

Both `echo` and `print` are language constructs, not functions. That means they do not require parentheses around their arguments. For cosmetic consistency with functions, parentheses can be included. Extensive examples of the use of `echo` and `print` are available elsewhere.

C-style `printf` and related functions are available as well, as in the following example:

```
printf("%s\n", "Hello, World!");
```

See Outputting the value of a variable for a comprehensive introduction of outputting variables in PHP.

Section 1.3: Non-HTML output from web server

In some cases, when working with a web server, overriding the web server's default content type may be required. There may be cases where you need to send data as `plain text`, `JSON`, or `XML`, for example.

The `header()` function can send a raw HTTP header. You can add the `Content-Type` header to notify the browser of the content we are sending.

Consider the following code, where we set `Content-Type` as `text/plain`:

```
header("Content-Type: text/plain");
echo "Hello World";
```

This will produce a plain text document with the following content:

> Hello World

To produce JSON content, use the `application/json` content type instead:

```
header("Content-Type: application/json");

// Create a PHP data array.
$data = ["response" => "Hello World"];

// json_encode will convert it to a valid JSON string.
echo json_encode($data);
```

This will produce a document of type `application/json` with the following content:

> {"response":"Hello World"}

Note that the `header()` function must be called before PHP produces any output, or the web server will have already sent headers for the response. So, consider the following code:

```
// Error: We cannot send any output before the headers
echo "Hello";

// All headers must be sent before ANY PHP output
header("Content-Type: text/plain");
echo "World";
```

This will produce a warning:

> **Warning:** Cannot modify header information - headers already sent by (output started at /dir/example.php:2) in **/dir/example.php** on line **3**

When using `header()`, its output needs to be the first byte that's sent from the server. For this reason it's important to not have empty lines or spaces in the beginning of the file before the PHP opening tag **`<?php`**. For the same reason, it is considered best practice (see PSR-2) to omit the PHP closing tag **`?>`** from files that contain only PHP and from blocks of PHP code at the very end of a file.

View the **output buffering section** to learn how to 'catch' your content into a variable to output later, for example, after outputting headers.

Section 1.4: PHP built-in server

PHP 5.4+ comes with a built-in development server. It can be used to run applications without having to install a production HTTP server such as nginx or Apache. The built-in server is only designed to be used for development and testing purposes.

It can be started by using the `-S` flag:

```
php -S <host/ip>:<port>
```

Example usage

1. Create an `index.php` file containing:

```
<?php
echo "Hello World from built-in PHP server";
```

2. Run the command `php -S localhost:8080` from the command line. Do not include

   `http://`

   . This will start a web server listening on port 8080 using the current directory that you are in as the document root.

3. Open the browser and navigate to `http://localhost:8080`. You should see your "Hello World" page.

Configuration

To override the default document root (i.e. the current directory), use the `-t` flag:

```
php -S <host/ip>:<port> -t <directory>
```

E.g. if you have a **`public/`** directory in your project you can serve your project from that directory using `php -S localhost:8080 -t public/`.

Logs

Every time a request is made from the development server, a log entry like the one below is written to the command line.

```
[Mon Aug 15 18:20:19 2016] ::1:52455 [200]: /
```

Section 1.5: PHP CLI

PHP can also be run from command line directly using the CLI (Command Line Interface).

CLI is basically the same as PHP from web servers, except some differences in terms of standard input and output.

Triggering

The PHP CLI allows four ways to run PHP code:

1. Standard input. Run the `php` command without any arguments, but pipe PHP code into it: echo '<?php echo "Hello world!";' | php
2. Filename as argument. Run the `php` command with the name of a PHP source file as the first argument: php hello_world.php

3. Code as argument. Use the `-r` option in the `php` command, followed by the code to run. The **`<?php`** open tags are not required, as everything in the argument is considered as PHP code: php -r 'echo "Hello world!";'
4. Interactive shell. Use the `-a` option in the `php` command to launch an interactive shell. Then, type (or paste) PHP code and hit `return`: $ php -a Interactive mode enabled php > echo "Hello world!"; Hello world!

Output

All functions or controls that produce HTML output in web server PHP can be used to produce output in the stdout stream (file descriptor 1), and all actions that produce output in error logs in web server PHP will produce output in the stderr stream (file descriptor 2).

`Example.php`

```
<?php
echo "Stdout 1\n";
trigger_error("Stderr 2\n");
print_r("Stdout 3\n");
fwrite(STDERR, "Stderr 4\n");
throw new RuntimeException("Stderr 5\n");
?>
Stdout 6
```

Shell command line

```
$ php Example.php 2>stderr.log >stdout.log;\
> echo STDOUT; cat stdout.log; echo;\
> echo STDERR; cat stderr.log\

STDOUT
Stdout 1
Stdout 3

STDERR
Stderr 4
PHP Notice:  Stderr 2
 in /Example.php on line 3
PHP Fatal error:  Uncaught RuntimeException: Stderr 5
 in /Example.php:6
Stack trace:
#0 {main}
 thrown in /Example.php on line 6
```

Input

See: Command Line Interface (CLI)

Section 1.6: Instruction Separation

Just like most other C-style languages, each statement is terminated with a semicolon. Also, a closing tag is used to terminate the last line of code of the PHP block.

If the last line of PHP code ends with a semicolon, the closing tag is optional if there is no code following that final line of code. For example, we can leave out the closing tag after `echo "No error";` in the following example:

```
<?php echo "No error"; // no closing tag is needed as long as there is no code below
```

However, if there is any other code following your PHP code block, the closing tag is no longer optional:

```
<?php echo "This will cause an error if you leave out the closing tag"; ?>
<html>
    <body>
```

```
    </body>
</html>
```

We can also leave out the semicolon of the last statement in a PHP code block if that code block has a closing tag:

```
<?php echo "I hope this helps! :D";
echo "No error" ?>
```

It is generally recommended to always use a semicolon and use a closing tag for every PHP code block except the last PHP code block, if no more code follows that PHP code block.

So, your code should basically look like this:

```
<?php
    echo "Here we use a semicolon!";
    echo "Here as well!";
    echo "Here as well!";
    echo "Here we use a semicolon and a closing tag because more code follows";
?>
<p>Some HTML code goes here</p>
<?php
    echo "Here we use a semicolon!";
    echo "Here as well!";
    echo "Here as well!";
    echo "Here we use a semicolon and a closing tag because more code follows";
?>
<p>Some HTML code goes here</p>
<?php
    echo "Here we use a semicolon!";
    echo "Here as well!";
    echo "Here as well!";
    echo "Here we use a semicolon but leave out the closing tag";
```

Section 1.7: PHP Tags

There are three kinds of tags to denote PHP blocks in a file. The PHP parser is looking for the opening and (if present) closing tags to delimit the code to interpret.

Standard Tags

These tags are the standard method to embed PHP code in a file.

```
<?php
    echo "Hello World";
?>
```

PHP 5.x `Version ≥ 5.4`

Echo Tags

These tags are available in all PHP versions, and since PHP 5.4 are always enabled. In previous versions, echo tags could only be enabled in conjunction with short tags.

```
<?= "Hello World" ?>
```

Short Tags

You can disable or enable these tags with the option `short_open_tag`.

```
<?
    echo "Hello World";
?>
```

Short tags:

- are disallowed in all major PHP coding standards
- are discouraged in the official documentation
- are disabled by default in most distributions
- interfere with inline XML's processing instructions
- are not accepted in code submissions by most open source projects

PHP 5.x `Version ≤ 5.6`

ASP Tags

By enabling the `asp_tags` option, ASP-style tags can be used.

```
<%
    echo "Hello World";
%>
```

These are an historic quirk and should never be used. They were removed in PHP 7.0.

Chapter 2: Variables

Section 2.1: Accessing A Variable Dynamically By Name (Variable variables)

Variables can be accessed via dynamic variable names. The name of a variable can be stored in another variable, allowing it to be accessed dynamically. Such variables are known as variable variables.

To turn a variable into a variable variable, you put an extra $ put in front of your variable.

```
$variableName = 'foo';
$foo = 'bar';

// The following are all equivalent, and all output "bar":
echo $foo;
echo ${$variableName};
echo $$variableName;

//similarly,
$variableName  = 'foo';
$$variableName = 'bar';

// The following statements will also output 'bar'
echo $foo;
echo $$variableName;
echo ${$variableName};
```

Variable variables are useful for mapping function/method calls:

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

$funcName = 'add';

echo $funcName(1, 2); // outputs 3
```

This becomes particularly helpful in PHP classes:

```
class myClass {
    public function __construct() {
        $functionName = 'doSomething';
        $this->$functionName('Hello World');
    }

    private function doSomething($string) {
        echo $string; // Outputs "Hello World"
    }
}
```

It is possible, but not required to put `$variableName` between {}:

```
${$variableName} = $value;
```

The following examples are both equivalent and output "baz":

```
$fooBar = 'baz';
```

```php
$varPrefix = 'foo';

echo $fooBar;                // Outputs "baz"
echo ${$varPrefix . 'Bar'}; // Also outputs "baz"
```

Using {} is only mandatory when the name of the variable is itself an expression, like this:

```php
${$variableNamePart1 . $variableNamePart2} = $value;
```

It is nevertheless recommended to always use {}, because it's more readable.

While it is not recommended to do so, it is possible to chain this behavior:

```php
$$$$$$$$DoNotTryThisAtHomeKids = $value;
```

> It's important to note that the excessive usage of variable variables is considered a bad practice by many developers. Since they're not well-suited for static analysis by modern IDEs, large codebases with many variable variables (or dynamic method invocations) can quickly become difficult to maintain.

Differences between PHP5 and PHP7

Another reason to always use {} or (), is that PHP5 and PHP7 have a slightly different way of dealing with dynamic variables, which results in a different outcome in some cases.

In PHP7, dynamic variables, properties, and methods will now be evaluated strictly in left-to-right order, as opposed to the mix of special cases in PHP5. The examples below show how the order of evaluation has changed.

Case 1 : `$$foo['bar']['baz']`

- PHP5 interpretation : `${$foo['bar']['baz']}`
- PHP7 interpretation : `($$foo)['bar']['baz']`

Case 2 : `$foo->$bar['baz']`

- PHP5 interpretation : `$foo->{$bar['baz']}`
- PHP7 interpretation : `($foo->$bar)['baz']`

Case 3 : `$foo->$bar['baz']()`

- PHP5 interpretation : `$foo->{$bar['baz']}()`
- PHP7 interpretation : `($foo->$bar)['baz']()`

Case 4 : `Foo::$bar['baz']()`

- PHP5 interpretation : `Foo::{$bar['baz']}()`
- PHP7 interpretation : `(Foo::$bar)['baz']()`

Section 2.2: Data Types

There are different data types for different purposes. PHP does not have explicit type definitions, but the type of a variable is determined by the type of the value that is assigned, or by the type that it is casted to. This is a brief overview about the types, for a detailed documentation and examples, see the PHP types topic.

There are following data types in PHP: null, boolean, integer, float, string, object, resource and array.

Null

Null can be assigned to any variable. It represents a variable with no value.

```
$foo = null;
```

This invalidates the variable and it's value would be undefined or void if called. The variable is cleared from memory and deleted by the garbage collector.

Boolean

This is the simplest type with only two possible values.

```
$foo = true;
$bar = false;
```

Booleans can be used to control the flow of code.

```
$foo = true;

if ($foo) {
    echo "true";
} else {
    echo "false";
}
```

Integer

An integer is a whole number positive or negative. It can be in used with any number base. The size of an integer is platform-dependent. PHP does not support unsigned integers.

```
$foo = -3;  // negative
$foo = 0;   // zero (can also be null or false (as boolean)
$foo = 123; // positive decimal
$bar = 0123; // octal = 83 decimal
$bar = 0xAB; // hexadecimal = 171 decimal
$bar = 0b1010; // binary = 10 decimal
var_dump(0123, 0xAB, 0b1010); // output: int(83) int(171) int(10)
```

Float

Floating point numbers, "doubles" or simply called "floats" are decimal numbers.

```
$foo = 1.23;
$foo = 10.0;
$bar = -INF;
$bar = NAN;
```

Array

An array is like a list of values. The simplest form of an array is indexed by integer, and ordered by the index, with the first element lying at index 0.

```
$foo = array(1, 2, 3); // An array of integers
$bar = ["A", true, 123 => 5]; // Short array syntax, PHP 5.4+
```

```
echo $bar[0];    // Returns "A"
echo $bar[1];    // Returns true
echo $bar[123];  // Returns 5
echo $bar[1234]; // Returns null
```

Arrays can also associate a key other than an integer index to a value. In PHP, all arrays are associative arrays behind the scenes, but when we refer to an 'associative array' distinctly, we usually mean one that contains one or more keys that aren't integers.

```
$array = array();
$array["foo"] = "bar";
$array["baz"] = "quux";
$array[42] = "hello";
echo $array["foo"]; // Outputs "bar"
echo $array["bar"]; // Outputs "quux"
echo $array[42]; // Outputs "hello"
```

String

A string is like an array of characters.

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

Like an array, a string can be indexed to return its individual characters:

```
$foo = "bar";
echo $foo[0]; // Prints 'b', the first character of the string in $foo.
```

Object

An object is an instance of a class. Its variables and methods can be accessed with the `->` operator.

```
$foo = new stdClass(); // create new object of class stdClass, which a predefined, empty class
$foo->bar = "baz";
echo $foo->bar; // Outputs "baz"
// Or we can cast an array to an object:
$quux = (object) ["foo" => "bar"];
echo $quux->foo; // This outputs "bar".
```

Resource

Resource variables hold special handles to opened files, database connections, streams, image canvas areas and the like (as it is stated in the manual).

```
$fp = fopen('file.ext', 'r'); // fopen() is the function to open a file on disk as a resource.
var_dump($fp); // output: resource(2) of type (stream)
```

To get the type of a variable as a string, use the `gettype()` function:

```
echo gettype(1); // outputs "integer"
echo gettype(true); // "boolean"
```

Section 2.3: Global variable best practices

We can illustrate this problem with the following pseudo-code

```
function foo() {
    global $bob;
    $bob->doSomething();
}
```

Your first question here is an obvious one

> Where did `$bob` come from?

Are you confused? Good. You've just learned why globals are confusing and considered a **bad practice**.

If this were a real program, your next bit of fun is to go track down all instances of `$bob` and hope you find the right one (this gets worse if `$bob` is used everywhere). Worse, if someone else goes and defines `$bob` (or you forgot and reused that variable) your code can break (in the above code example, having the wrong object, or no object at all, would cause a fatal error).

Since virtually all PHP programs make use of code like `include('file.php');` your job maintaining code like this becomes exponentially harder the more files you add.

Also, this makes the task of testing your applications very difficult. Suppose you use a global variable to hold your database connection:

```
$dbConnector = new DBConnector(...);

function doSomething() {
    global $dbConnector;
    $dbConnector->execute("...");
}
```

In order to unit test this function, you have to override the global `$dbConnector` variable, run the tests and then reset it to its original value, which is very bug prone:

```
/**
 * @test
 */
function testSomething() {
     global $dbConnector;

     $bkp = $dbConnector; // Make backup
     $dbConnector = Mock::create('DBConnector'); // Override

     assertTrue(foo());

     $dbConnector = $bkp; // Restore
}
```

How do we avoid Globals?

The best way to avoid globals is a philosophy called **Dependency Injection**. This is where we pass the tools we need into the function or class.

```php
function foo(\Bar $bob) {
    $bob->doSomething();
}
```

This is **much** easier to understand and maintain. There's no guessing where `$bob` was set up because the caller is responsible for knowing that (it's passing us what we need to know). Better still, we can use type declarations to restrict what's being passed.

So we know that `$bob` is either an instance of the `Bar` class, or an instance of a child of `Bar`, meaning we know we can use the methods of that class. Combined with a standard autoloader (available since PHP 5.3), we can now go track down where `Bar` is defined. PHP 7.0 or later includes expanded type declarations, where you can also use scalar types (like `int` or `string`).

```
Version = 4.1
```

Superglobal variables

Super globals in PHP are predefined variables, which are always available, can be accessed from any scope throughout the script.

There is no need to do global $variable; to access them within functions/methods, classes or files.

These PHP superglobal variables are listed below:

- $GLOBALS
- $_SERVER
- $_REQUEST
- $_POST
- $_GET
- $_FILES
- $_ENV
- $_COOKIE
- $_SESSION

Section 2.4: Default values of uninitialized variables

Although not necessary in PHP however it is a very good practice to initialize variables. Uninitialized variables have a default value of their type depending on the context in which they are used:

Unset AND unreferenced

```php
var_dump($unset_var); // outputs NULL
```

Boolean

```php
echo($unset_bool ? "true\n" : "false\n"); // outputs 'false'
```

String

```php
$unset_str .= 'abc';
var_dump($unset_str); // outputs 'string(3) "abc"'
```

Integer

```
$unset_int += 25; // 0 + 25 => 25
var_dump($unset_int); // outputs 'int(25)'
```

Float/double

```
$unset_float += 1.25;
var_dump($unset_float); // outputs 'float(1.25)'
```

Array

```
$unset_arr[3] = "def";
var_dump($unset_arr); //  outputs array(1) {  [3]=>  string(3) "def" }
```

Object

```
$unset_obj->foo = 'bar';
var_dump($unset_obj); // Outputs: object(stdClass)#1 (1) {  ["foo"]=>  string(3) "bar" }
```

Relying on the default value of an uninitialized variable is problematic in the case of including one file into another which uses the same variable name.

Section 2.5: Variable Value Truthiness and Identical Operator

In PHP, variable values have an associated "truthiness" so even non-boolean values will equate to `true` or `false`. This allows any variable to be used in a conditional block, e.g.

```
if ($var == true) { /* explicit version */ }
if ($var) { /* $var == true is implicit */ }
```

Here are some fundamental rules for different types of variable values:

- **Strings** with non-zero length equate to `true` including strings containing only whitepace such as `' '`.
- Empty strings `''` equate to `false`.

```
$var = '';
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true

$var = '    ';
$var_is_true = ($var == true); // true
$var_is_false = ($var == false); // false
```

- **Integers** equate to `true` if they are nonzero, while zero equates to `false`.

```
$var = -1;
$var_is_true = ($var == true); // true
$var = 99;
$var_is_true = ($var == true); // true
$var = 0;
$var_is_true = ($var == true); // false
```

- `null` equates to `false`

```
$var = null;
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true
```

- **Empty** strings `''` and string zero `'0'` equate to `false`.

```
$var = '';
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true

$var = '0';
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true
```

- **Floating-point** values equate to `true` if they are nonzero, while zero values equates to `false`.
  - `NAN` (PHP's Not-a-Number) equates to `true`, i.e. `NAN == true` is `true`. This is because `NAN` is a *nonzero* floating-point value.
  - Zero-values include both +0 and -0 as defined by IEEE 754. PHP does not distinguish between +0 and -0 in its double-precision floating-point, i.e. `floatval('0') == floatval('-0')` is `true`.
    - In fact, `floatval('0') === floatval('-0')`.
    - Additionally, both `floatval('0') == false` and `floatval('-0') == false`.

```
$var = NAN;
$var_is_true = ($var == true); // true
$var_is_false = ($var == false); // false

$var = floatval('-0');
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true

$var = floatval('0') == floatval('-0');
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true
```

IDENTICAL OPERATOR

In the PHP Documentation for Comparison Operators, there is an Identical Operator `===`. This operator can be used to check whether a variable is *identical* to a reference value:

```
$var = null;
$var_is_null = $var === null; // true
$var_is_true = $var === true; // false
$var_is_false = $var === false; // false
```

It has a corresponding *not identical* operator `!==`:

```
$var = null;
$var_is_null = $var !== null; // false
$var_is_true = $var !== true; // true
$var_is_false = $var !== false; // true
```

The identical operator can be used as an alternative to language functions like `is_null()`.

USE CASE WITH `strpos()`

The `strpos($haystack, $needle)` language function is used to locate the index at which `$needle` occurs in `$haystack`, or whether it occurs at all. The `strpos()` function is case sensitive; if case-insensitive find is what you need you can go with `stripos($haystack, $needle)`

The `strpos` & `stripos` function also contains third parameter `offset` (int) which if specified, search will start this number of characters counted from the beginning of the string. Unlike strrpos and strripos, the offset cannot be

negative

The function can return:

- `0` if `$needle` is found at the beginning of `$haystack`;
- a non-zero integer specifying the index if `$needle` is found somewhere other than the beginning in `$haystack`;
- and value `false` if `$needle` is *not* found anywhere in `$haystack`.

Because both `0` and `false` have truthiness `false` in PHP but represent distinct situations for `strpos()`, it is important to distinguish between them and use the identical operator `===` to look exactly for `false` and not just a value that equates to `false`.

```
$idx = substr($haystack, $needle);
if ($idx === false)
{
    // logic for when $needle not found in $haystack
}
else
{
    // logic for when $needle found in $haystack
}
```

Alternatively, using the *not identical* operator:

```
$idx = substr($haystack, $needle);
if ($idx !== false)
{
    // logic for when $needle found in $haystack
}
else
{
    // logic for when $needle not found in $haystack
}
```

Chapter 3: Variable Scope

Variable scope refers to the regions of code where a variable may be accessed. This is also referred to as *visibility*. In PHP scope blocks are defined by functions, classes, and a global scope available throughout an application.

Section 3.1: Superglobal variables

Superglobal variables are defined by PHP and can always be used from anywhere without the **`global`** keyword.

```
<?php

function getPostValue($key, $default = NULL) {
    // $_POST is a superglobal and can be used without
    // having to specify 'global $_POST;'
    if (isset($_POST[$key])) {
        return $_POST[$key];
    }

    return $default;
}

// retrieves $_POST['username']
echo getPostValue('username');

// retrieves $_POST['email'] and defaults to empty string
echo getPostValue('email', '');
```

Section 3.2: Static properties and variables

Static class properties that are defined with the **`public`** visibility are functionally the same as global variables. They can be accessed from anywhere the class is defined.

```
class SomeClass {
    public static int $counter = 0;
}

// The static $counter variable can be read/written from anywhere
// and doesn't require an instantiation of the class
SomeClass::$counter += 1;
```

Functions can also define static variables inside their own scope. These static variables persist through multiple function calls, unlike regular variables defined in a function scope. This can be a very easy and simple way to implement the Singleton design pattern:

```
class Singleton {
    public static function getInstance() {
        // Static variable $instance is not deleted when the function ends
        static $instance;

        // Second call to this function will not get into the if-statement,
        // Because an instance of Singleton is now stored in the $instance
        // variable and is persisted through multiple calls
        if (!$instance) {
            // First call to this function will reach this line,
            // because the $instance has only been declared, not initialized
            $instance = new Singleton();
        }
```

```
        return $instance;

    }
}

$instance1 = Singleton::getInstance();
$instance2 = Singleton::getInstance();

// Comparing objects with the '===' operator checks whether they are
// the same instance. Will print 'true', because the static $instance
// variable in the getInstance() method is persisted through multiple calls
var_dump($instance1 === $instance2);
```

Section 3.3: User-defined global variables

The scope outside of any function or class is the global scope. When a PHP script includes another (using `include` or `require`) the scope remains the same. If a script is included outside of any function or class, it's global variables are included in the same global scope, but if a script is included from within a function, the variables in the included script are in the scope of the function.

Within the scope of a function or class method, the **`global`** keyword may be used to create an access user-defined global variables.

```
<?php

$amount_of_log_calls = 0;

function log_message($message) {
    // Accessing global variable from function scope
    // requires this explicit statement
    global $amount_of_log_calls;

    // This change to the global variable is permanent
    $amount_of_log_calls += 1;

    echo $message;
}

// When in the global scope, regular global variables can be used
// without explicitly stating 'global $variable;'
echo $amount_of_log_calls; // 0

log_message("First log message!");
echo $amount_of_log_calls; // 1

log_message("Second log message!");
echo $amount_of_log_calls; // 2
```

A second way to access variables from the global scope is to use the special PHP-defined $GLOBALS array.

The $GLOBALS array is an associative array with the name of the global variable being the key and the contents of that variable being the value of the array element. Notice how $GLOBALS exists in any scope, this is because $GLOBALS is a superglobal.

This means that the `log_message()` function could be rewritten as:

```
function log_message($message) {
    // Access the global $amount_of_log_calls variable via the
    // $GLOBALS array. No need for 'global $GLOBALS;', since it
```

```
    // is a superglobal variable.
    $GLOBALS['amount_of_log_calls'] += 1;

    echo $messsage;
}
```

One might ask, why use the $GLOBALS array when the **`global`** keyword can also be used to get a global variable's value? The main reason is using the **`global`** keyword will bring the variable into scope. You then can't reuse the same variable name in the local scope.

Chapter 4: Superglobal Variables PHP

Superglobals are built-in variables that are always available in all scopes.

Several predefined variables in PHP are "superglobals", which means they are available in all scopes throughout a script. There is no need to do `global $variable;` to access them within functions or methods.

Section 4.1: Suberglobals explained

Introduction

Put simply, these are variables that are available in *all* scope in your scripts.

This means that there is no need to pass them as parameters in your functions, or store them outside a block of code to have them available in different scopes.

What's a superglobal??

If you're thinking that these are like superheroes - they're not.

As of PHP version 7.1.3 there are 9 superglobal variables. They are as follows:

- `$GLOBALS` - References all variables available in global scope
- `$_SERVER` - Server and execution environment information
- `$_GET` - HTTP GET variables
- `$_POST` - HTTP POST variables
- `$_FILES` - HTTP File Upload variables
- `$_COOKIE` - HTTP Cookies
- `$_SESSION` - Session variables
- `$_REQUEST` - HTTP Request variables
- `$_ENV` - Environment variables

See the documentation.

Tell me more, tell me more

I'm sorry for the Grease reference! Link

Time for some explanation on these superheroesglobals.

`$GLOBALS`

> An associative array containing references to all variables which are currently defined in the global scope of the script. The variable names are the keys of the array.

Code

```
$myGlobal = "global"; // declare variable outside of scope

function test()
{
    $myLocal = "local"; // declare variable inside of scope
    // both variables are printed
    var_dump($myLocal);
```

```
    var_dump($GLOBALS["myGlobal"]);
}

test(); // run function
// only $myGlobal is printed since $myLocal is not globally scoped
var_dump($myLocal);
var_dump($myGlobal);
```

Output

```
string 'local' (length=5)
string 'global' (length=6)
null
string 'global' (length=6)
```

In the above example `$myLocal` is not displayed the second time because it is declared inside the `test()` function and then destroyed after the function is closed.

Becoming global

To remedy this there are two options.

Option one: **`global` keyword**

```
function test()
{
    global $myLocal;
    $myLocal = "local";
    var_dump($myLocal);
    var_dump($GLOBALS["myGlobal"]);
}
```

The **`global`** keyword is a prefix on a variable that forces it to be part of the global scope.

Note that you cannot assign a value to a variable in the same statement as the global keyword. Hence, why I had to assign a value underneath. (It is possible if you remove new lines and spaces but I don't think it is neat. **`global`** `$myLocal; $myLocal = "local"`).

Option two: **`$GLOBALS`** **array**

```
function test()
{
    $GLOBALS["myLocal"] = "local";
    $myLocal = $GLOBALS["myLocal"];
    var_dump($myLocal);
    var_dump($GLOBALS["myGlobal"]);
}
```

In this example I reassigned `$myLocal` the value of `$GLOBAL["myLocal"]` since I find it easier writing a variable name rather than the associative array.

`$_SERVER`

> $_SERVER is an array containing information such as headers, paths, and script locations. The entries in this array are created by the web server. There is no guarantee that every web server will provide any of these; servers may omit some, or provide others not listed here. That said, a large number of these

> variables are accounted for in the CGI/1.1 specification, so you should be able to expect those.

An example output of this might be as follows (run on my Windows PC using WAMP)

```
C:\wamp64\www\test.php:2:
array (size=36)
    'HTTP_HOST' => string 'localhost' (length=9)
    'HTTP_CONNECTION' => string 'keep-alive' (length=10)
    'HTTP_CACHE_CONTROL' => string 'max-age=0' (length=9)
    'HTTP_UPGRADE_INSECURE_REQUESTS' => string '1' (length=1)
    'HTTP_USER_AGENT' => string 'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/57.0.2987.133 Safari/537.36' (length=110)
    'HTTP_ACCEPT' => string
'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8' (length=74)
    'HTTP_ACCEPT_ENCODING' => string 'gzip, deflate, sdch, br' (length=23)
    'HTTP_ACCEPT_LANGUAGE' => string 'en-US,en;q=0.8,en-GB;q=0.6' (length=26)
    'HTTP_COOKIE' => string 'PHPSESSID=0gslnvgsci371ete9hg7k9ivc6' (length=36)
    'PATH' => string 'C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files
(x86)\Intel\iCLS Client\;C:\Program Files\Intel\iCLS
Client\;C:\ProgramData\Oracle\Java\javapath;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem
;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;E:\Program Files\ATI Technologies\ATI.ACE\Core-
Static;E:\Program Files\AMD\ATI.ACE\Core-Static;C:\Program Files (x86)\AMD\ATI.ACE\Core-
Static;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Static;C:\Program Files\Intel\Intel(R)
Managemen'... (length=1169)
    'SystemRoot' => string 'C:\WINDOWS' (length=10)
    'COMSPEC' => string 'C:\WINDOWS\system32\cmd.exe' (length=27)
    'PATHEXT' => string '.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.PY' (length=57)
    'WINDIR' => string 'C:\WINDOWS' (length=10)
    'SERVER_SIGNATURE' => string '<address>Apache/2.4.23 (Win64) PHP/7.0.10 Server at localhost
Port 80</address>' (length=80)
    'SERVER_SOFTWARE' => string 'Apache/2.4.23 (Win64) PHP/7.0.10' (length=32)
    'SERVER_NAME' => string 'localhost' (length=9)
    'SERVER_ADDR' => string '::1' (length=3)
    'SERVER_PORT' => string '80' (length=2)
    'REMOTE_ADDR' => string '::1' (length=3)
    'DOCUMENT_ROOT' => string 'C:/wamp64/www' (length=13)
    'REQUEST_SCHEME' => string 'http' (length=4)
    'CONTEXT_PREFIX' => string '' (length=0)
    'CONTEXT_DOCUMENT_ROOT' => string 'C:/wamp64/www' (length=13)
    'SERVER_ADMIN' => string 'wampserver@wampserver.invalid' (length=29)
    'SCRIPT_FILENAME' => string 'C:/wamp64/www/test.php' (length=26)
    'REMOTE_PORT' => string '5359' (length=4)
    'GATEWAY_INTERFACE' => string 'CGI/1.1' (length=7)
    'SERVER_PROTOCOL' => string 'HTTP/1.1' (length=8)
    'REQUEST_METHOD' => string 'GET' (length=3)
    'QUERY_STRING' => string '' (length=0)
    'REQUEST_URI' => string '/test.php' (length=13)
    'SCRIPT_NAME' => string '/test.php' (length=13)
    'PHP_SELF' => string '/test.php' (length=13)
    'REQUEST_TIME_FLOAT' => float 1491068771.413
    'REQUEST_TIME' => int 1491068771
```

There is a lot to take in there so I will pick out some important ones below. If you wish to read about them all then consult the indices section of the documentation.

I might add them all below one day. Or someone can edit and add a ***good*** explanation of them below? *Hint, hint*;)

For all explanations below, assume the URL is http://www.example.com/index.php

- `HTTP_HOST` - The host address.

This would return `www.example.com`
- `HTTP_USER_AGENT` - Contents of the user agent. This is a string which contains all the information about the client's browser, including operating system.
- `HTTP_COOKIE` - All cookies in a concatenated string, with a semi-colon delimiter.
- `SERVER_ADDR` - The IP address of the server, of which the current script is running.
  This would return `93.184.216.34`
- `PHP_SELF` - The file name of the currently executed script, relative to document root.
  This would return `/index.php`
- `REQUEST_TIME_FLOAT` - The timestamp of the start of the request, with microsecond precision. Available since PHP 5.4.0.
- `REQUEST_TIME` - The timestamp of the start of the request. Available since PHP 5.1.0.

`$_GET`

> An associative array of variables passed to the current script via the URL parameters.

`$_GET` is an array that contains all the URL parameters; these are the whatever is after the ? in the URL.

Using http://www.example.com/index.php?myVar=myVal as an example. This information from this URL can be obtained by accessing in this format `$_GET["myVar"]` and the result of this will be `myVal`.

Using some code for those that don't like reading.

```
// URL = http://www.example.com/index.php?myVar=myVal
echo $_GET["myVar"] == "myVal" ? "true" : "false"; // returns "true"
```

The above example makes use of the ternary operator.

This shows how you can access the value from the URL using the `$_GET` superglobal.

Now another example! *gasp*

```
// URL = http://www.example.com/index.php?myVar=myVal&myVar2=myVal2
echo $_GET["myVar"]; // returns "myVal"
echo $_GET["myVar2"]; // returns "myVal2"
```

It is possible to send multiple variables through the URL by separating them with an ampersand (`&`) character.

Security risk
It is very important not to send any sensitive information via the URL as it will stay in history of the computer and will be visible to anyone that can access that browser.

`$_POST`

> An associative array of variables passed to the current script via the HTTP POST method when using application/x-www-form-urlencoded or multipart/form-data as the HTTP Content-Type in the request.

Very similar to `$_GET` in that data is sent from one place to another.

I'll start by going straight into an example. (I have omitted the action attribute as this will send the information to the page that the form is in).

```
<form method="POST">
    <input type="text" name="myVar" value="myVal" />
    <input type="submit" name="submit" value="Submit" />
</form>
```

Above is a basic form for which data can be sent. In an real environment the `value` attribute would not be set meaning the form would be blank. This would then send whatever information is entered by the user.

```
echo $_POST["myVar"]); // returns "myVal"
```

Security risk
Sending data via POST is also not secure. Using HTTPS will ensure that data is kept more secure.

`$_FILES`

> An associative array of items uploaded to the current script via the HTTP POST method. The structure of this array is outlined in the POST method uploads section.

Let's start with a basic form.

```
<form method="POST" enctype="multipart/form-data">
    <input type="file" name="myVar" />
    <input type="submit" name="Submit" />
</form>
```

Note that I omitted the `action` attribute (again!). Also, I added `enctype="multipart/form-data"`, this is important to any form that will be dealing with file uploads.

```
// ensure there isn't an error
if ($_FILES["myVar"]["error"] == UPLOAD_ERR_OK)
{
    $folderLocation = "myFiles"; // a relative path. (could be "path/to/file" for example)

    // if the folder doesn't exist then make it
    if (!file_exists($folderLocation)) mkdir($folderLocation);

    // move the file into the folder
    move_uploaded_file($_FILES["myVar"]["tmp_name"], "$folderLocation/" .
basename($_FILES["myVar"]["name"]));
}
```

This is used to upload one file. Sometimes you may wish to upload more than one file. An attribute exists for that, it's called `multiple`.
There's an attribute for just about *anything*. I'm sorry

Below is an example of a form submitting multiple files.

```
<form method="POST" enctype="multipart/form-data">
    <input type="file" name="myVar[]" multiple="multiple" />
    <input type="submit" name="Submit" />
</form>
```

Note the changes made here; there are only a few.

- The `input` name has square brackets. This is because it is now an array of files and so we are telling the form

to make an array of the files selected. Omitting the square brackets will result in the latter most file being set to `$_FILES["myVar"]`.

- The `multiple="multiple"` attribute. This just tells the browser that users can select more than one file.

```
$total = isset($_FILES["myVar"]) ? count($_FILES["myVar"]["name"]) : 0; // count how many files
were sent
// iterate over each of the files
for ($i = 0; $i < $total; $i++)
{
    // there isn't an error
    if ($_FILES["myVar"]["error"][$i] == UPLOAD_ERR_OK)
    {
        $folderLocation = "myFiles"; // a relative path. (could be "path/to/file" for example)

        // if the folder doesn't exist then make it
        if (!file_exists($folderLocation)) mkdir($folderLocation);

        // move the file into the folder
        move_uploaded_file($_FILES["myVar"]["tmp_name"][$i], "$folderLocation/" .
basename($_FILES["myVar"]["name"][$i]));
    }
    // else report the error
    else switch ($_FILES["myVar"]["error"][$i])
    {
        case UPLOAD_ERR_INI_SIZE:
            echo "Value: 1; The uploaded file exceeds the upload_max_filesize directive in
php.ini.";
            break;
        case UPLOAD_ERR_FORM_SIZE:
            echo "Value: 2; The uploaded file exceeds the MAX_FILE_SIZE directive that was
specified in the HTML form.";
            break;
        case UPLOAD_ERR_PARTIAL:
            echo "Value: 3; The uploaded file was only partially uploaded.";
            break;
        case UPLOAD_ERR_NO_FILE:
            echo "Value: 4; No file was uploaded.";
            break;
        case UPLOAD_ERR_NO_TMP_DIR:
            echo "Value: 6; Missing a temporary folder. Introduced in PHP 5.0.3.";
            break;
        case UPLOAD_ERR_CANT_WRITE:
            echo "Value: 7; Failed to write file to disk. Introduced in PHP 5.1.0.";
            break;
        case UPLOAD_ERR_EXTENSION:
            echo "Value: 8; A PHP extension stopped the file upload. PHP does not provide a way to
ascertain which extension caused the file upload to stop; examining the list of loaded extensions
with phpinfo() may help. Introduced in PHP 5.2.0.";
            break;

        default:
            echo "An unknown error has occurred.";
            break;
    }
}
```

This is a very simple example and doesn't handle problems such as file extensions that aren't allowed or files named with PHP code (like a PHP equivalent of an SQL injection). See the documentation.

The first process is checking if there are any files, and if so, set the total number of them to `$total`.

Using the for loop allows an iteration of the `$_FILES` array and accessing each item one at a time. If that file doesn't encounter a problem then the if statement is true and the code from the single file upload is run.
If an problem is encountered the switch block is executed and an error is presented in accordance with the error for that particular upload.

$_COOKIE

> An associative array of variables passed to the current script via HTTP Cookies.

Cookies are variables that contain data and are stored on the client's computer.

Unlike the aforementioned superglobals, cookies must be created with a function (and not be assigning a value). The convention is below.

```
setcookie("myVar", "myVal", time() + 3600);
```

In this example a name is specified for the cookie (in this example it is "myVar"), a value is given (in this example it is "myVal", but a variable can be passed to assign its value to the cookie), and then an expiration time is given (in this example it is one hour since 3600 seconds is a minute).

Despite the convention for creating a cookie being different, it is accessed in the same way as the others.

```
echo $_COOKIE["myVar"]; // returns "myVal"
```

To destroy a cookie, `setcookie` must be called again, but the expiration time is set to *any* time in the past. See below.

```
setcookie("myVar", "", time() - 1);
var_dump($_COOKIE["myVar"]); // returns null
```

This will unset the cookies and remove it from the clients computer.

$_SESSION

> An associative array containing session variables available to the current script. See the Session functions documentation for more information on how this is used.

Sessions are much like cookies except they are server side.

To use sessions you must include `session_start()` at the top of your scripts to allow sessions to be utilised.

Setting a session variable is the same as setting any other variable. See example below.

```
$_SESSION["myVar"] = "myVal";
```

When starting a session a random ID is set as a cookie and called "PHPSESSID" and will contain the session ID for that current session. This can be accessed by calling the `session_id()` function.

It is possible to destroy session variables using the `unset` function (such that `unset($_SESSION["myVar"])` would destroy that variable).
The alternative is to call `session_destory()`. This will destroy the entire session meaning that **all** session variables will no longer exist.

$_REQUEST

> An associative array that by default contains the contents of $_GET, $_POST and $_COOKIE.

As the PHP documentation states, this is just a collation of $_GET, $_POST, and $_COOKIE all in one variable.

Since it is possible for all three of those arrays to have an index with the same name, there is a setting in the php.ini file called request_order which can specify which of the three has precedence.
For instance, if it was set to "GPC", then the value of $_COOKIE will be used, as it is read from left to right meaning the $_REQUEST will set its value to $_GET, then $_POST, and then $_COOKIE and since $_COOKIE is last that is the value that is in $_REQUEST.
See this question.

$_ENV

> An associative array of variables passed to the current script via the environment method.
>
> These variables are imported into PHP's global namespace from the environment under which the PHP parser is running. Many are provided by the shell under which PHP is running and different systems are likely running different kinds of shells, a definitive list is impossible. Please see your shell's documentation for a list of defined environment variables.
>
> Other environment variables include the CGI variables, placed there regardless of whether PHP is running as a server module or CGI processor.

Anything stored within $_ENV is from the environment from which PHP is running in.

$_ENV is only populated if php.ini allows it.
See this answer for more information on why $_ENV is not populated.

Section 4.2: PHP5 SuperGlobals

Below are the PHP5 SuperGlobals

- $GLOBALS
- $_REQUEST
- $_GET
- $_POST
- $_FILES
- $_SERVER
- $_ENV
- $_COOKIE
- $_SESSION

$GLOBALS: This SuperGlobal Variable is used for accessing globals variables.

```
<?php
$a = 10;
function foo(){
    echo $GLOBALS['a'];
}
//Which will print 10 Global Variable a
```

```
?>
```

$_REQUEST: This SuperGlobal Variable is used to collect data submitted by a HTML Form.

```
<?php
if(isset($_REQUEST['user'])){
    echo $_REQUEST['user'];
}
//This will print value of HTML Field with name=user submitted using POST and/or GET MEthod
?>
```

$_GET: This SuperGlobal Variable is used to collect data submitted by HTML Form with `get` method.

```
<?php
if(isset($_GET['username'])){
    echo $_GET['username'];
}
//This will print value of HTML field with name username submitted using GET Method
?>
```

$_POST: This SuperGlobal Variable is used to collect data submitted by HTML Form with `post` method.

```
<?php
if(isset($_POST['username'])){
    echo $_POST['username'];
}
//This will print value of HTML field with name username submitted using POST Method
?>
```

$_FILES: This SuperGlobal Variable holds the information of uploaded files via HTTP Post method.

```
<?php
if($_FILES['picture']){
    echo "<pre>";
    print_r($_FILES['picture']);
    echo "</pre>";
}
/**
This will print details of the File with name picture uploaded via a form with method='post and with
enctype='multipart/form-data'
Details includes Name of file, Type of File, temporary file location, error code(if any error
occurred while uploading the file) and size of file in Bytes.
Eg.

Array
(
    [picture] => Array
        (
            [0] => Array
                (
                    [name] => 400.png
                    [type] => image/png
                    [tmp_name] => /tmp/php5Wx0aJ
                    [error] => 0
                    [size] => 15726
                )
        )
)
```

```
*/
?>
```

$_SERVER: This SuperGlobal Variable holds information about Scripts, HTTP Headers and Server Paths.

```
<?php
    echo "<pre>";
    print_r($_SERVER);
    echo "</pre>";
    /**
    Will print the following details
    on my local XAMPP
    Array
(
    [MIBDIRS] => C:/xampp/php/extras/mibs
    [MYSQL_HOME] => \xampp\mysql\bin
    [OPENSSL_CONF] => C:/xampp/apache/bin/openssl.cnf
    [PHP_PEAR_SYSCONF_DIR] => \xampp\php
    [PHPRC] => \xampp\php
    [TMP] => \xampp\tmp
    [HTTP_HOST] => localhost
    [HTTP_CONNECTION] => keep-alive
    [HTTP_CACHE_CONTROL] => max-age=0
    [HTTP_UPGRADE_INSECURE_REQUESTS] => 1
    [HTTP_USER_AGENT] => Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/52.0.2743.82 Safari/537.36
    [HTTP_ACCEPT] => text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*;q=0.8
    [HTTP_ACCEPT_ENCODING] => gzip, deflate, sdch
    [HTTP_ACCEPT_LANGUAGE] => en-US,en;q=0.8
    [PATH] => C:\xampp\php;C:\ProgramData\ComposerSetup\bin;
    [SystemRoot] => C:\Windows
    [COMSPEC] => C:\Windows\system32\cmd.exe
    [PATHEXT] => .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
    [WINDIR] => C:\Windows
    [SERVER_SIGNATURE] => Apache/2.4.16 (Win32) OpenSSL/1.0.1p PHP/5.6.12 Server at localhost Port 80
    [SERVER_SOFTWARE] => Apache/2.4.16 (Win32) OpenSSL/1.0.1p PHP/5.6.12
    [SERVER_NAME] => localhost
    [SERVER_ADDR] => ::1
    [SERVER_PORT] => 80
    [REMOTE_ADDR] => ::1
    [DOCUMENT_ROOT] => C:/xampp/htdocs
    [REQUEST_SCHEME] => http
    [CONTEXT_PREFIX] =>
    [CONTEXT_DOCUMENT_ROOT] => C:/xampp/htdocs
    [SERVER_ADMIN] => postmaster@localhost
    [SCRIPT_FILENAME] => C:/xampp/htdocs/abcd.php
    [REMOTE_PORT] => 63822
    [GATEWAY_INTERFACE] => CGI/1.1
    [SERVER_PROTOCOL] => HTTP/1.1
    [REQUEST_METHOD] => GET
    [QUERY_STRING] =>
    [REQUEST_URI] => /abcd.php
    [SCRIPT_NAME] => /abcd.php
    [PHP_SELF] => /abcd.php
    [REQUEST_TIME_FLOAT] => 1469374173.88
    [REQUEST_TIME] => 1469374173
)
*/
?>
```

$_ENV: This SuperGlobal Variable Shell Environment Variable details under which the PHP is running.

$_COOKIE: This SuperGlobal Variable is used to retrieve Cookie value with given Key.

```php
<?php
$cookie_name = "data";
$cookie_value = "Foo Bar";
setcookie($cookie_name, $cookie_value, time() + (86400 * 30), "/"); // 86400 = 1 day
if(!isset($_COOKIE[$cookie_name])) {
    echo "Cookie named '" . $cookie_name . "' is not set!";
}
else {
    echo "Cookie '" . $cookie_name . "' is set!<br>";
    echo "Value is: " . $_COOKIE[$cookie_name];
}

/**
    Output
    Cookie 'data' is set!
    Value is: Foo Bar
*/
?>
```

$_SESSION: This SuperGlobal Variable is used to Set and Retrieve Session Value which is stored on Server.

```php
<?php
//Start the session
session_start();
/**
    Setting the Session Variables
    that can be accessed on different
    pages on save server.
*/
$_SESSION["username"] = "John Doe";
$_SESSION["user_token"] = "d5f1df5b4dfb8b8d5f";
echo "Session is saved successfully";

/**
    Output
    Session is saved successfully
*/
?>
```

Chapter 5: Outputting the Value of a Variable

To build a dynamic and interactive PHP program, it is useful to output variables and their values. The PHP language allows for multiple methods of value output. This topic covers the standard methods of printing a value in PHP and where these methods can be used.

Section 5.1: echo and print

`echo` and `print` are language constructs, not functions. This means that they don't require parentheses around the argument like a function does (although one can always add parentheses around almost any PHP expression and thus `echo("test")` won't do any harm either). They output the string representation of a variable, constant, or expression. They can't be used to print arrays or objects.

- Assign the string `Joel` to the variable `$name`

```
$name = "Joel";
```

- Output the value of $name using `echo` & `print`

```
echo $name;   #> Joel
print $name;  #> Joel
```

- Parentheses are not required, but can be used

```
echo($name);  #> Joel
print($name); #> Joel
```

- Using multiple parameters (only `echo`)

```
echo $name, "Smith";       #> JoelSmith
echo($name, " ", "Smith"); #> Joel Smith
```

- `print`, unlike `echo`, is an expression (it returns `1`), and thus can be used in more places:

```
print("hey") && print(" ") && print("you"); #> you11
```

- The above is equivalent to:

```
print ("hey" && (print (" " && print "you"))); #> you11
```

Shorthand notation for `echo`

When outside of PHP tags, a shorthand notation for `echo` is available by default, using **`<?=`** to begin output and **`?>`** to end it. For example:

```
<p><?=$variable?></p>
<p><?= "This is also PHP" ?></p>
```

Note that there is no terminating `;`. This works because the closing PHP tag acts as the terminator for the single

statement. So, it is conventional to omit the semicolon in this shorthand notation.

Priority of `print`

Although the `print` is language construction it has priority like operator. It places between `=` `+=` `-=` `*=` `**=` `/=` `.=` `%=` `&=` and `and` operators and has left association. Example:

```
echo '1' . print '2' + 3; //output 511
```

Same example with brackets:

```
echo '1' . print ('2' + 3); //output 511
```

Differences between `echo` and `print`

In short, there are two main differences:

- `print` only takes one parameter, while `echo` can have multiple parameters.
- `print` returns a value, so can be used as an expression.

Section 5.2: Outputting a structured view of arrays and objects

`print_r()` - Outputting Arrays and Objects *for debugging*

`print_r` will output a human readable format of an array or object.

You may have a variable that is an array or object. Trying to output it with an `echo` will throw the error: `Notice: Array to string conversion`. You can instead use the `print_r` function to dump a human readable format of this variable.

> You can pass **true** as the second parameter to return the content as a string.

```
$myobject = new stdClass();
$myobject->myvalue = 'Hello World';
$myarray = [ "Hello", "World" ];
$mystring = "Hello World";
$myint = 42;

// Using print_r we can view the data the array holds.
print_r($myobject);
print_r($myarray);
print_r($mystring);
print_r($myint);
```

This outputs the following:

```
stdClass Object
(
    [myvalue] => Hello World
)
Array
(
    [0] => Hello
    [1] => World
)
```

```
Hello World
42
```

Further, the output from `print_r` can be captured as a string, rather than simply echoed. For instance, the following code will dump the formatted version of `$myarray` into a new variable:

```
$formatted_array = print_r($myarray, true);
```

Note that if you are viewing the output of PHP in a browser, and it is interpreted as HTML, then the line breaks will not be shown and the output will be much less legible unless you do something like

```
echo '<pre>' . print_r($myarray, true) . '</pre>';
```

> Opening the source code of a page will also format your variable in the same way without the use of the **`<pre>`** tag.

Alternatively you can tell the browser that what you're outputting is plain text, and not HTML:

```
header('Content-Type: text/plain; charset=utf-8');
print_r($myarray);
```

`var_dump()` - Output human-readable debugging information about content of the argument(s) including its type and value

The output is more detailed as compared to `print_r` because it also outputs the **type** of the variable along with its **value** and other information like object IDs, array sizes, string lengths, reference markers, etc.

You can use `var_dump` to output a more detailed version for debugging.

```
var_dump($myobject, $myarray, $mystring, $myint);
```

Output is more detailed:

```
object(stdClass)#12 (1) {
 ["myvalue"]=>
  string(11) "Hello World"
}
array(2) {
  [0]=>
  string(5) "Hello"
  [1]=>
  string(5) "World"
}
string(11) "Hello World"
int(42)
```

Note: If you are using xDebug in your development environment, the output of var_dump is limited / truncated by default. See the official documentation for more info about the options to change this.

`var_export()` - Output valid `PHP` Code

`var_export()` dumps a `PHP` parseable representation of the item.

> You can pass **true** as the second parameter to return the contents into a variable.

```
var_export($myarray);
var_export($mystring);
var_export($myint);
```

Output is valid PHP code:

```
array (
  0 => 'Hello',
  1 => 'World',
)
'Hello World'
42
```

To put the content into a variable, you can do this:

```
$array_export = var_export($myarray, true);
$string_export = var_export($mystring, true);
$int_export = var_export($myint, 1); // any `Truthy` value
```

After that, you can output it like this:

```
printf('$myarray = %s; %s', $array_export, PHP_EOL);
printf('$mystring = %s; %s', $string_export, PHP_EOL);
printf('$myint = %s; %s', $int_export, PHP_EOL);
```

This will produce the following output:

```
$myarray = array (
  0 => 'Hello',
  1 => 'World',
);
$mystring = 'Hello World';
$myint = 42;
```

Section 5.3: String concatenation with echo

You can use concatenation to join strings "end to end" while outputting them (with `echo` or `print` for example).

You can concatenate variables using a `.` (period/dot).

```
// String variable
$name = 'Joel';

// Concatenate multiple strings (3 in this example) into one and echo it once done.
//      1. ↓        2. ↓            3. ↓    - Three Individual string items
echo '<p>Hello ' . $name . ', Nice to see you.</p>';
//               ↑       ↑                  - Concatenation Operators

#> "<p>Hello Joel, Nice to see you.</p>"
```

Similar to concatenation, `echo` (when used without parentheses) can be used to combine strings and variables together (along with other arbitrary expressions) using a comma (,).

```
$itemCount = 1;
```

```
echo 'You have ordered ', $itemCount, ' item', $itemCount === 1 ? '' : 's';
//                     ↑           ↑        ↑                - Note the commas

#> "You have ordered 1 item"
```

String concatenation vs passing multiple arguments to echo

Passing multiple arguments to the echo command is more advantageous than string concatenation in some circumstances. The arguments are written to the output in the same order as they are passed in.

```
echo "The total is: ", $x + $y;
```

The problem with the concatenation is that the period `.` takes precedence in the expression. If concatenated, the above expression needs extra parentheses for the correct behavior. The precedence of the period affects ternary operators too.

```
echo "The total is: " . ($x + $y);
```

Section 5.4: printf vs sprintf

`printf` will **output** a formatted string using placeholders

`sprintf` will **return** the formatted string

```
$name = 'Jeff';

// The `%s` tells PHP to expect a string
//            ↓   `%s` is replaced by  ↓
printf("Hello %s, How's it going?", $name);
#> Hello Jeff, How's it going?

// Instead of outputting it directly, place it into a variable ($greeting)
$greeting = sprintf("Hello %s, How's it going?", $name);
echo $greeting;
#> Hello Jeff, How's it going?
```

It is also possible to format a number with these 2 functions. This can be used to format a decimal value used to represent money so that it always has 2 decimal digits.

```
$money = 25.2;
printf('%01.2f', $money);
#> 25.20
```

The two functions `vprintf` and `vsprintf` operate as `printf` and `sprintf`, but accept a format string and an array of values, instead of individual variables.

Section 5.5: Outputting large integers

On 32-bits systems, integers larger than `PHP_INT_MAX` are automatically converted to float. Outputting these as integer values (i.e. non-scientific notation) can be done with `printf`, using the `float` representation, as illustrated below:

```
foreach ([1, 2, 3, 4, 5, 6, 9, 12] as $p) {
    $i = pow(1024, $p);
    printf("pow(1024, %d) > (%7s) %20s %38.0F", $p, gettype($i), $i, $i);
```

```
    echo "   ", $i, "\n";
}
// outputs:
pow(1024,  1)  integer                 1024                                  1024  1024
pow(1024,  2)  integer              1048576                               1048576  1048576
pow(1024,  3)  integer           1073741824                            1073741824  1073741824
pow(1024,  4)   double        1099511627776                         1099511627776  1099511627776
pow(1024,  5)   double  1.1258999068426E+15                      1125899906842624
 1.1258999068426E+15
pow(1024,  6)   double  1.1529215046068E+18                   1152921504606846976
 1.1529215046068E+18
pow(1024,  9)   double  1.2379400392854E+27           1237940039285380274899124224
 1.2379400392854E+27
pow(1024, 12)   double  1.3292279957849E+36  1329227995784915872903807060280344576
 1.3292279957849E+36
```

> Note: watch out for float precision, which is not infinite!

While this looks nice, in this contrived example the numbers can all be represented as a binary number since they are all powers of 1024 (and thus 2). See for example:

```
$n = pow(10, 27);
printf("%s %.0F\n", $n, $n);
// 1.0E+27 1000000000000000013287555072
```

Section 5.6: Output a Multidimensional Array with index and value and print into the table

```
Array
(
    [0] => Array
        (
            [id] => 13
            [category_id] => 7
            [name] => Leaving Of Liverpool
            [description] => Leaving Of Liverpool
            [price] => 1.00
            [virtual] => 1
            [active] => 1
            [sort_order] => 13
            [created] => 2007-06-24 14:08:03
            [modified] => 2007-06-24 14:08:03
            [image] => NONE
        )

    [1] => Array
        (
            [id] => 16
            [category_id] => 7
            [name] => Yellow Submarine
            [description] => Yellow Submarine
            [price] => 1.00
            [virtual] => 1
            [active] => 1
            [sort_order] => 16
            [created] => 2007-06-24 14:10:02
            [modified] => 2007-06-24 14:10:02
```

```
            [image] => NONE
        )

)
```

Output Multidimensional Array with index and value in table

```
<table>
<?php
foreach ($products as $key => $value) {
    foreach ($value as $k => $v) {
        echo "<tr>";
        echo "<td>$k</td>"; // Get index.
        echo "<td>$v</td>"; // Get value.
        echo "</tr>";
    }
}
?>
</table>
```

Chapter 6: Constants

Section 6.1: Defining constants

Constants are created using the `const` statement or the `define` function. The convention is to use UPPERCASE letters for constant names.

Define constant using explicit values

```
const PI = 3.14; // float
define("EARTH_IS_FLAT", false); // boolean
const "UNKNOWN" = null; // null
define("APP_ENV", "dev"); // string
const MAX_SESSION_TIME = 60 * 60; // integer, using (scalar) expressions is ok

const APP_LANGUAGES = ["de", "en"]; // arrays


define("BETTER_APP_LANGUAGES", ["lu", "de"]); // arrays
```

Define constant using another constant

if you have one constant you can define another one based on it:

```
const TAU = PI * 2;
define("EARTH_IS_ROUND", !EARTH_IS_FLAT);
define("MORE_UNKNOWN", UNKNOWN);
define("APP_ENV_UPPERCASE", strtoupper(APP_ENV)); // string manipulation is ok too
// the above example (a function call) does not work with const:
// const TIME = time(); # fails with a fatal error! Not a constant scalar expression
define("MAX_SESSION_TIME_IN_MINUTES", MAX_SESSION_TIME / 60);

const APP_FUTURE_LANGUAGES = [-1 => "es"] + APP_LANGUAGES; // array manipulations


define("APP_BETTER_FUTURE_LANGUAGES", array_merge(["fr"], APP_BETTER_LANGUAGES));
```

Reserved constants

Some constant names are reserved by PHP and cannot be redefined. All these examples will fail:

```
define("true", false); // internal constant
define("false", true); // internal constant
define("CURLOPT_AUTOREFERER", "something"); // will fail if curl extension is loaded
```

And a Notice will be issued:

```
Constant ... already defined in ...
```

Conditional defines

If you have several files where you may define the same variable (for example, your main config then your local config) then following syntax may help avoiding conflicts:

```
defined("PI") || define("PI", 3.1415); // "define PI if it's not yet defined"
```

`const` vs `define`

`define` is a runtime expression while `const` a compile time one.

Thus `define` allows for dynamic values (i.e. function calls, variables etc.) and even dynamic names and conditional definition. It however is always defining relative to the root namespace.

`const` is static (as in allows only operations with other constants, scalars or arrays, and only a restricted set of them, the so called *constant scalar expressions*, i.e. arithmetic, logical and comparison operators as well as array dereferencing), but are automatically namespace prefixed with the currently active namespace.

`const` only supports other constants and scalars as values, and no operations.

Section 6.2: Class Constants

Constants can be defined inside classes using a **`const`** keyword.

```
class Foo {
    const BAR_TYPE = "bar";

    // reference from inside the class using self::
    public function myMethod() {
        return self::BAR_TYPE;
    }
}

// reference from outside the class using <ClassName>::
echo Foo::BAR_TYPE;
```

This is useful to store types of items.

```
<?php

class Logger {
    const LEVEL_INFO = 1;
    const LEVEL_WARNING = 2;
    const LEVEL_ERROR = 3;

    // we can even assign the constant as a default value
    public function log($message, $level = self::LEVEL_INFO) {
        echo "Message level " . $level . ": " . $message;
    }
}

$logger = new Logger();
$logger->log("Info"); // Using default value
$logger->log("Warning", $logger::LEVEL_WARNING); // Using var
$logger->log("Error", Logger::LEVEL_ERROR); // using class
```

Section 6.3: Checking if constant is defined

Simple check

To check if constant is defined use the `defined` function. Note that this function doesn't care about constant's value, it only cares if the constant exists or not. Even if the value of the constant is **`null`** or **`false`** the function will still return **`true`**.

```
<?php

define("GOOD", false);

if (defined("GOOD")) {
```

```
    print "GOOD is defined" ; // prints "GOOD is defined"

    if (GOOD) {
        print "GOOD is true" ; // does not print anything, since GOOD is false
    }
}

if (!defined("AWESOME")) {
   define("AWESOME", true); // awesome was not defined. Now we have defined it
}
```

Note that constant becomes "visible" in your code only **after** the line where you have defined it:

```
<?php

if (defined("GOOD")) {
   print "GOOD is defined"; // doesn't print anyhting, GOOD is not defined yet.
}

define("GOOD", false);

if (defined("GOOD")) {
   print "GOOD is defined"; // prints "GOOD is defined"
}
```

Getting all defined constants

To get all defined constants including those created by PHP use the `get_defined_constants` function:

```
<?php

$constants = get_defined_constants();
var_dump($constants); // pretty large list
```

To get only those constants that were defined by your app call the function at the beginning and at the end of your script (normally after the bootstrap process):

```
<?php

$constants = get_defined_constants();

define("HELLO", "hello");
define("WORLD", "world");

$new_constants = get_defined_constants();

$myconstants = array_diff_assoc($new_constants, $constants);
var_export($myconstants);

/*
Output:

array (
  'HELLO' => 'hello',
  'WORLD' => 'world',
)
*/
```

It's sometimes useful for debugging

Section 6.4: Using constants

To use the constant simply use its name:

```
if (EARTH_IS_FLAT) {
    print "Earth is flat";
}

print APP_ENV_UPPERCASE;
```

or if you don't know the name of the constant in advance, use the `constant` function:

```
// this code is equivalent to the above code
$const1 = "EARTH_IS_FLAT";
$const2 = "APP_ENV_UPPERCASE";

if (constant($const1)) {
    print "Earth is flat";
}

print constant($const2);
```

Section 6.5: Constant arrays

Arrays can be used as plain constants and class constants from version PHP 5.6 onwards:

Class constant example

```
class Answer {
    const C = [2,4];
}

print Answer::C[1] . Answer::C[0]; // 42
```

Plain constant example

```
const ANSWER = [2,4];
print ANSWER[1] . ANSWER[0]; // 42
```

Also from version PHP 7.0 this functionality was ported to the `define` function for plain constants.

```
define('VALUES', [2, 3]);
define('MY_ARRAY', [
    1,
    VALUES,
]);

print MY_ARRAY[1][1]; // 3
```

Chapter 7: Magic Constants

Section 7.1: Difference between __FUNCTION__ and __METHOD__

`__FUNCTION__` returns only the name of the function whereas `__METHOD__` returns the name of the class along with the name of the function:

```
<?php

class trick
{
    public function doit()
    {
        echo __FUNCTION__;
    }

    public function doitagain()
    {
        echo __METHOD__;
    }
}

$obj = new trick();
$obj->doit(); // Outputs: doit
$obj->doitagain();  // Outputs: trick::doitagain
```

Section 7.2: Difference between __CLASS__, get_class() and get_called_class()

`__CLASS__` magic constant returns the same result as `get_class()` function called without parameters and they both return the name of the class where it was defined (i.e. where you wrote the function call/constant name).

In contrast, `get_class($this)` and `get_called_class()` functions call, will both return the name of the actual class which was instantiated:

```
<?php

class Definition_Class {

  public function say(){
     echo '__CLASS__ value: ' . __CLASS__ . "\n";
     echo 'get_called_class() value: ' . get_called_class() . "\n";
     echo 'get_class($this) value: ' . get_class($this) . "\n";
     echo 'get_class() value: ' . get_class() . "\n";
  }

}

class Actual_Class extends Definition_Class {}

$c = new Actual_Class();
$c->say();
// Output:
// __CLASS__ value: Definition_Class
// get_called_class() value: Actual_Class
// get_class($this) value: Actual_Class
```

```
// get_class() value: Definition_Class
```

Section 7.3: File & Directory Constants

Current file

You can get the name of the current PHP file (with the absolute path) using the `__FILE__` magic constant. This is most often used as a logging/debugging technique.

```
echo "We are in the file:" , __FILE__ , "\n";
```

Current directory

To get the absolute path to the directory where the current file is located use the `__DIR__` magic constant.

```
echo "Our script is located in the:" , __DIR__ , "\n";
```

To get the absolute path to the directory where the current file is located, use `dirname(__FILE__)`.

```
echo "Our script is located in the:" , dirname(__FILE__) , "\n";
```

Getting current directory is often used by PHP frameworks to set a base directory:

```
// index.php of the framework

define(BASEDIR, __DIR__); // using magic constant to define normal constant
```

```
// somefile.php looks for views:

$view = 'page';
$viewFile = BASEDIR . '/views/' . $view;
```

Separators

> Windows system perfectly understands the `/` in paths so the `DIRECTORY_SEPARATOR` is used mainly when parsing paths.

Besides magic constants PHP also adds some fixed constants for working with paths:

- `DIRECTORY_SEPARATOR` constant for separating directories in a path. Takes value `/` on *nix, and `\` on Windows. The example with views can be rewritten with:

```
$view = 'page';
$viewFile = BASEDIR . DIRECTORY_SEPARATOR .'views' . DIRECTORY_SEPARATOR . $view;
```

- Rarely used `PATH_SEPARATOR` constant for separating paths in the `$PATH` environment variable. It is `;` on Windows, `:` otherwise

Chapter 8: Comments

Section 8.1: Single Line Comments

The single line comment begins with "//" or "#". When encountered, all text to the right will be ignored by the PHP interpreter.

```
// This is a comment

# This is also a comment

echo "Hello World!"; // This is also a comment, beginning where we see "//"
```

Section 8.2: Multi Line Comments

The multi-line comment can be used to comment out large blocks of code. It begins with `/*` and ends with `*/`.

```
/* This is a multi-line comment.
   It spans multiple lines.
   This is still part of the comment.
*/
```

Chapter 9: Types

Section 9.1: Type Comparison

There are two types of comparison: **loose comparison** with `==` and **strict comparison** with `===`. Strict comparison ensures both the type and value of both sides of the operator are the same.

```
// Loose comparisons
var_dump(1 == 1); // true
var_dump(1 == "1"); // true
var_dump(1 == true); // true
var_dump(0 == false); // true

// Strict comparisons
var_dump(1 === 1); // true
var_dump(1 === "1"); // false
var_dump(1 === true); // false
var_dump(0 === false); // false

// Notable exception: NAN — it never is equal to anything
var_dump(NAN == NAN); // false
var_dump(NAN === NAN); // false
```

You can also use strong comparison to check if type and value **don't** match using `!==`.

A typical example where the `==` operator is not enough, are functions that can return different types, like `strpos`, which returns `false` if the `searchword` is not found, and the match position (`int`) otherwise:

```
if(strpos('text', 'searchword') == false)
  // strpos returns false, so == comparison works as expected here, BUT:
if(strpos('text bla', 'text') == false)
  // strpos returns 0 (found match at position 0) and 0==false is true.
  // This is probably not what you expect!
if(strpos('text','text') === false)
  // strpos returns 0, and 0===false is false, so this works as expected.
```

Section 9.2: Boolean

Boolean is a type, having two values, denoted as `true` or `false`.

This code sets the value of `$foo` as `true` and `$bar` as `false`:

```
$foo = true;
$bar = false;
```

`true` and `false` are not case sensitive, so `TRUE` and `FALSE` can be used as well, even `FaLsE` is possible. Using lower case is most common and recommended in most code style guides, e.g. PSR-2.

Booleans can be used in if statements like this:

```
if ($foo) { //same as evaluating if($foo == true)
    echo "true";
}
```

Due to the fact that PHP is weakly typed, if `$foo` above is other than `true` or `false`, it's automatically coerced to a boolean value.

The following values result in `false`:

- a zero value: `0` (integer), `0.0` (float), or `'0'` (string)
- an empty string `''` or array [ ]
- `null` (the content of an unset variable, or assigned to a variable)

Any other value results in `true`.

To avoid this loose comparison, you can enforce strong comparison using `===`, which compares value *and type*. See Type Comparison for details.

To convert a type into boolean, you can use the `(bool)` or `(boolean)` cast before the type.

```
var_dump((bool) "1"); //evaluates to true
```

or call the `boolval` function:

```
var_dump( boolval("1") ); //evaluates to true
```

Boolean conversion to a string (note that `false` yields an empty string):

```
var_dump( (string) true ); // string(1) "1"
var_dump( (string) false ); // string(0) ""
```

Boolean conversion to an integer:

```
var_dump( (int) true ); // int(1)
var_dump( (int) false ); // int(0)
```

Note that the opposite is also possible:

```
var_dump((bool) "");        // bool(false)
var_dump((bool) 1);         // bool(true)
```

Also all non-zero will return true:

```
var_dump((bool) -2);        // bool(true)
var_dump((bool) "foo");     // bool(true)
var_dump((bool) 2.3e5);     // bool(true)
var_dump((bool) array(12)); // bool(true)
var_dump((bool) array());   // bool(false)
var_dump((bool) "false");   // bool(true)
```

Section 9.3: Float

```
$float = 0.123;
```

> For historical reasons "double" is returned by `gettype()` in case of a float, and not simply "float"

Floats are floating point numbers, which allow more output precision than plain integers.

Floats and integers can be used together due to PHP's loose casting of variable types:

```php
$sum = 3 + 0.14;

echo $sum; // 3.14
```

php does not show float as float number like other languages, for example:

```php
$var = 1;
echo ((float) $var); //returns 1 not 1.0
```

Warning

Floating point precision

(From the PHP manual page)

> Floating point numbers have limited precision. Although it depends on the system, PHP typically give a maximum relative error due to rounding in the order of 1.11e-16. Non elementary arithmetic operations may give larger errors, and error *propagation* must be considered when several operations are compounded.

> Additionally, rational numbers that are exactly representable as floating point numbers in base 10, like 0.1 or 0.7, do not have an exact representation as floating point numbers in base 2 (binary), which is used internally, no matter the size of the mantissa. Hence, they cannot be converted into their internal binary counterparts without a small loss of precision. This can lead to confusing results: for example, floor((0.1+0.7)*10) will usually return 7 instead of the expected 8, since the internal representation will be something like 7.99999999999991118....

> So never trust floating number results to the last digit, and do not compare floating point numbers directly for equality. If higher precision is necessary, the arbitrary precision math functions and gmp functions are available.

Section 9.4: Strings

A string in PHP is a series of single-byte characters (i.e. there is no native Unicode support) that can be specified in four ways:

Single Quoted

Displays things almost completely "as is". Variables and most escape sequences will not be interpreted. The exception is that to display a literal single quote, one can escape it with a back slash ', and to display a back slash, one can escape it with another backslash \

```php
$my_string = 'Nothing is parsed, except an escap\'d apostrophe or backslash. $foo\n';
var_dump($my_string);

/*
string(68) "Nothing is parsed, except an escap'd apostrophe or backslash. $foo\n"
*/
```

Double Quoted

Unlike a single-quoted string, simple variable names and escape sequences in the strings will be evaluated. Curly braces (as in the last example) can be used to isolate complex variable names.

```
$variable1 = "Testing!";
$variable2 = [ "Testing?", [ "Failure", "Success" ] ];
$my_string = "Variables and escape characters are parsed:\n\n";
$my_string .= "$variable1\n\n$variable2[0]\n\n";
$my_string .= "There are limits: $variable2[1][0]";
$my_string .= "But we can get around them by wrapping the whole variable in braces:
{$variable2[1][1]}";
var_dump($my_string);

/*
string(98) "Variables and escape characters are parsed:

Testing!

Testing?

There are limits: Array[0]"

But we can get around them by wrapping the whole variable in braces: Success

*/
```

Heredoc

In a heredoc string, variable names and escape sequences are parsed in a similar manner to double-quoted strings, though braces are not available for complex variable names. The start of the string is delimited by `<<<identifier`, and the end by `identifier`, where `identifier` is any valid PHP name. The ending identifier must appear on a line by itself. No whitespace is allowed before or after the identifier, although like any line in PHP, it must also be terminated by a semicolon.

```
$variable1 = "Including text blocks is easier";
$my_string = <<< EOF
Everything is parsed in the same fashion as a double-quoted string,
but there are advantages. $variable1; database queries and HTML output
can benefit from this formatting.
Once we hit a line containing nothing but the identifier, the string ends.
EOF;
var_dump($my_string);

/*
string(268) "Everything is parsed in the same fashion as a double-quoted string,
but there are advantages. Including text blocks is easier; database queries and HTML output
can benefit from this formatting.
Once we hit a line containing nothing but the identifier, the string ends."
*/
```

Nowdoc

A nowdoc string is like the single-quoted version of heredoc, although not even the most basic escape sequences are evaluated. The identifier at the beginning of the string is wrapped in single quotes.

PHP 5.x `Version ≥ 5.3`

```
$my_string = <<< 'EOF'
A similar syntax to heredoc but, similar to single quoted strings,
nothing is parsed (not even escaped apostrophes \' and backslashes \\.)
```

```
EOF;
var_dump($my_string);

/*
string(116) "A similar syntax to heredoc but, similar to single quoted strings,
nothing is parsed (not even escaped apostrophes \' and backslashes \\.)"
*/
```

Section 9.5: Callable

Callables are anything which can be called as a callback. Things that can be termed a "callback" are as follows:

- Anonymous functions
- Standard PHP functions (note: *not language constructs*)
- Static Classes
- non-static Classes (*using an alternate syntax*)
- Specific Object/Class Methods
- Objects themselves, as long as the object is found in key `0` of an array

  Example Of referencing an object as an array element:

```
$obj = new MyClass();
call_user_func([$obj, 'myCallbackMethod']);
```

Callbacks can be denoted by `callable` type hint as of PHP 5.4.

```
$callable = function () {
    return 'value';
};

function call_something(callable $fn) {
    call_user_func($fn);
}

call_something($callable);
```

Section 9.6: Resources

A *resource* is a special type of variable that references an external resource, such as a file, socket, stream, document, or connection.

```
$file = fopen('/etc/passwd', 'r');

echo gettype($file);
# Out: resource

echo $file;
# Out: Resource id #2
```

There are different (sub-)types of resource. You can check the resource type using `get_resource_type()`:

```
$file = fopen('/etc/passwd', 'r');
```

```php
echo get_resource_type($file);
#Out: stream

$sock = fsockopen('www.google.com', 80);
echo get_resource_type($sock);
#Out: stream
```

You can find a complete list of built-in resource types here.

Section 9.7: Type Casting

PHP will generally correctly guess the data type you intend to use from the context it's used in, however sometimes it is useful to manually force a type. This can be accomplished by prefixing the declaration with the name of the required type in parenthesis:

```php
$bool = true;
var_dump($bool); // bool(true)

$int = (int) true;
var_dump($int); // int(1)

$string = (string) true;
var_dump($string); // string(1) "1"
$string = (string) false;
var_dump($string); // string(0) ""

$float = (float) true;
var_dump($float); // float(1)

$array = ['x' => 'y'];
var_dump((object) $array); // object(stdClass)#1 (1) { ["x"]=> string(1) "y" }

$object = new stdClass();
$object->x = 'y';
var_dump((array) $object); // array(1) { ["x"]=> string(1) "y" }

$string = "asdf";
var_dump((unset)$string); // NULL
```

But be careful: not all type casts work as one might expect:

```php
// below 3 statements hold for 32-bits systems (PHP_INT_MAX=2147483647)
// an integer value bigger than PHP_INT_MAX is automatically converted to float:
var_dump(       999888777666 ); // float(999888777666)
// forcing to (int) gives overflow:
var_dump((int)  999888777666 ); // int(-838602302)
// but in a string it just returns PHP_INT_MAX
var_dump((int) "999888777666"); // int(2147483647)

var_dump((bool) []);      // bool(false) (empty array)
var_dump((bool) [false]); // bool(true)  (non-empty array)
```

Section 9.8: Type Juggling

PHP is a weakly-typed language. It does not require explicit declaration of data types. The context in which the variable is used determines its data type; conversion is done automatically:

```php
$a = "2";             // string
```

```
$a = $a + 2;          // integer (4)
$a = $a + 0.5;        // float (4.5)
$a = 1 + "2 oranges"; // integer (3)
```

Section 9.9: Null

PHP represents "no value" with the `null` keyword. It's somewhat similar to the null pointer in C-language and to the NULL value in SQL.

Setting the variable to null:

```
$nullvar = null; // directly

function doSomething() {} // this function does not return anything
$nullvar = doSomething(); // so the null is assigned to $nullvar
```

Checking if the variable was set to null:

```
if (is_null($nullvar)) { /* variable is null */ }

if ($nullvar === null) {  /* variable is null */ }
```

Null vs undefined variable

If the variable was not defined or was unset then any tests against the null will be successful but they will also generate a `Notice: Undefined variable: nullvar`:

```
$nullvar = null;
unset($nullvar);
if ($nullvar === null) {  /* true but also a Notice is printed */ }
if (is_null($nullvar)) {  /* true but also a Notice is printed */ }
```

Therefore undefined values must be checked with `isset`:

```
if (!isset($nullvar)) {  /* variable is null or is not even defined */  }
```

Section 9.10: Integers

Integers in PHP can be natively specified in base 2 (binary), base 8 (octal), base 10 (decimal), or base 16 (hexadecimal.)

```
$my_decimal = 42;
$my_binary = 0b101010;
$my_octal = 052;
$my_hexadecimal = 0x2a;

echo ($my_binary + $my_octal) / 2;
// Output is always in decimal: 42
```

Integers are 32 or 64 bits long, depending on the platform. The constant `PHP_INT_SIZE` holds integer size in bytes. `PHP_INT_MAX` and (since PHP 7.0) `PHP_INT_MIN` are also available.

```
printf("Integers are %d bits long" . PHP_EOL, PHP_INT_SIZE * 8);
printf("They go up to %d" . PHP_EOL, PHP_INT_MAX);
```

Integer values are automatically created as needed from floats, booleans, and strings. If an explicit typecast is

needed, it can be done with the `(int)` or `(integer)` cast:

```
$my_numeric_string = "123";
var_dump($my_numeric_string);
// Output: string(3) "123"
$my_integer = (int)$my_numeric_string;
var_dump($my_integer);
// Output: int(123)
```

Integer overflow will be handled by conversion to a float:

```
$too_big_integer = PHP_INT_MAX + 7;
var_dump($too_big_integer);
// Output: float(9.2233720368548E+18)
```

There is no integer division operator in PHP, but it can be simulated using an implicit cast, which always 'rounds' by just discarding the float-part. As of PHP version 7, an integer division function was added.

```
$not_an_integer = 25 / 4;
var_dump($not_an_integer);
// Output: float(6.25)
var_dump((int) (25 / 4)); // (see note below)
// Output: int(6)
var_dump(intdiv(25 / 4)); // as of PHP7
// Output: int(6)
```

(Note that the extra parentheses around `(25 / 4)` are needed because the `(int)` cast has higher precedence than the division)

Chapter 10: Operators

An operator is something that takes one or more values (or expressions, in programming jargon) and yields another value (so that the construction itself becomes an expression).

Operators can be grouped according to the number of values they take.

Section 10.1: Null Coalescing Operator (??)

Null coalescing is a new operator introduced in PHP 7. This operator returns its first operand if it is set and not `NULL`. Otherwise it will return its second operand.

The following example:

```
$name = $_POST['name'] ?? 'nobody';
```

is equivalent to both:

```
if (isset($_POST['name'])) {
    $name = $_POST['name'];
} else {
    $name = 'nobody';
}
```

and:

```
$name = isset($_POST['name']) ? $_POST['name'] : 'nobody';
```

This operator can also be chained (with right-associative semantics):

```
$name = $_GET['name'] ?? $_POST['name'] ?? 'nobody';
```

which is an equivalent to:

```
if (isset($_GET['name'])) {
    $name = $_GET['name'];
} elseif (isset($_POST['name'])) {
    $name = $_POST['name'];
} else {
    $name = 'nobody';
}
```

Note:
When using coalescing operator on string concatenation don't forget to use parentheses `()`

```
$firstName = "John";
$lastName = "Doe";
echo $firstName ?? "Unknown" . " " . $lastName ?? "";
```

This will output `John` only, and if its $firstName is null and $lastName is `Doe` it will output `Unknown Doe`. In order to output `John Doe`, we must use parentheses like this.

```
$firstName = "John";
$lastName = "Doe";
```

```
echo ($firstName ?? "Unknown") . " " . ($lastName ?? "");
```

This will output `John Doe` instead of `John` only.

Section 10.2: Spaceship Operator (<=>)

PHP 7 introduces a new kind of operator, which can be used to compare expressions. This operator will return -1, 0 or 1 if the first expression is less than, equal to, or greater than the second expression.

```
// Integers
print (1 <=> 1); // 0
print (1 <=> 2); // -1
print (2 <=> 1); // 1

// Floats
print (1.5 <=> 1.5); // 0
print (1.5 <=> 2.5); // -1
print (2.5 <=> 1.5); // 1

// Strings
print ("a" <=> "a"); // 0
print ("a" <=> "b"); // -1
print ("b" <=> "a"); // 1
```

Objects are not comparable, and so doing so will result in undefined behaviour.

This operator is particularly useful when writing a user-defined comparison function using `usort`, `uasort`, or `uksort`. Given an array of objects to be sorted by their `weight` property, for example, an anonymous function can use `<=>` to return the value expected by the sorting functions.

```
usort($list, function($a, $b) { return $a->weight <=> $b->weight; });
```

In PHP 5 this would have required a rather more elaborate expression.

```
usort($list, function($a, $b) {
    return $a->weight < $b->weight ? -1 : ($a->weight == $b->weight ? 0 : 1);
});
```

Section 10.3: Execution Operator (``)

The PHP execution operator consists of backticks (` `) and is used to run shell commands. The output of the command will be returned, and may, therefore, be stored in a variable.

```
// List files
$output = `ls`;
echo "<pre>$output</pre>";
```

Note that the execute operator and `shell_exec()` will give the same result.

Section 10.4: Incrementing (++) and Decrementing Operators (--)

Variables can be incremented or decremented by 1 with `++` or `--`, respectively. They can either precede or succeed variables and slightly vary semantically, as shown below.

```php
$i = 1;
echo $i; // Prints 1

// Pre-increment operator increments $i by one, then returns $i
echo ++$i; // Prints 2

// Pre-decrement operator decrements $i by one, then returns $i
echo --$i; // Prints 1

// Post-increment operator returns $i, then increments $i by one
echo $i++; // Prints 1 (but $i value is now 2)

// Post-decrement operator returns $i, then decrements $i by one
echo $i--; // Prints 2 (but $i value is now 1)
```

More information about incrementing and decrementing operators can be found in the official documentation.

Section 10.5: Ternary Operator (?:)

The ternary operator can be thought of as an inline `if` statement. It consists of three parts. The `operator`, and two outcomes. The syntax is as follows:

```
$value = <operator> ? <true value> : <false value>
```

If the `operator` is evaluated as **`true`**, the value in the first block will be returned (**`<true value>`**), else the value in the second block will be returned (**`<false value>`**). Since we are setting `$value` to the result of our ternary operator it will store the returned value.

Example:

```php
$action = empty($_POST['action']) ? 'default' : $_POST['action'];
```

`$action` would contain the string `'default'` if `empty($_POST['action'])` evaluates to true. Otherwise it would contain the value of `$_POST['action']`.

The expression `(expr1) ? (expr2) : (expr3)` evaluates to `expr2` if `expr1`evaluates to **`true`**, and `expr3` if `expr1` evaluates to **`false`**.

It is possible to leave out the middle part of the ternary operator. Expression `expr1 ?: expr3` returns `expr1` if `expr1` evaluates to TRUE, and `expr3` otherwise. `?:` is often referred to as *Elvis* operator.

This behaves like the Null Coalescing operator `??`, except that `??` requires the left operand to be exactly **`null`** while `?:` tries to resolve the left operand into a boolean and check if it resolves to boolean **`false`**.

Example:

```php
function setWidth(int $width = 0){
    $_SESSION["width"] = $width ?: getDefaultWidth();
}
```

In this example, `setWidth` accepts a width parameter, or default 0, to change the width session value. If `$width` is 0 (if `$width` is not provided), which will resolve to boolean false, the value of `getDefaultWidth()` is used instead. The `getDefaultWidth()` function will not be called if `$width` did not resolve to boolean false.

Refer to Types for more information about conversion of variables to boolean.

Section 10.6: Logical Operators (&&/AND and ||/OR)

In PHP, there are two versions of logical AND and OR operators.

Operator	True if
`$a and $b`	Both `$a` and `$b` are true
`$a && $b`	Both `$a` and `$b` are true
`$a or $b`	Either `$a` or `$b` is true
`$a || $b`	Either `$a` or `$b` is true

Note that the `&&` and `||` opererators have higher precedence than `and` and `or`. See table below:

Evaluation	Result of `$e`	Evaluated as
`$e = false || true`	True	`$e = (false || true)`
`$e = false or true`	False	`($e = false) or true`

Because of this it's safer to use `&&` and `||` instead of `and` and `or`.

Section 10.7: String Operators (. and .=)

There are only two string operators:

- Concatenation of two strings (dot):

```
$a = "a";
$b = "b";
$c = $a . $b; // $c => "ab"
```

- Concatenating assignment (dot=):

```
$a = "a";
$a .= "b"; // $a => "ab"
```

Section 10.8: Object and Class Operators

Members of objects or classes can be accessed using the object operator (`->`) and the class operator (`::`).

```
class MyClass {
    public $a = 1;
    public static $b = 2;
    const C = 3;
    public function d() { return 4; }
    public static function e() { return 5; }
}

$object = new MyClass();
var_dump($object->a);   // int(1)
var_dump($object::$b);  // int(2)
var_dump($object::C);   // int(3)
var_dump(MyClass::$b);  // int(2)
var_dump(MyClass::C);   // int(3)
var_dump($object->d()); // int(4)
var_dump($object::d()); // int(4)
var_dump(MyClass::e()); // int(5)
$classname = "MyClass";
```

```
var_dump($classname::e()); // also works! int(5)
```

Note that after the object operator, the $ should not be written (`$object->a` instead of `$object->$a`). For the class operator, this is not the case and the $ is necessary. For a constant defined in the class, the $ is never used.

Also note that `var_dump(MyClass::d());` is only allowed if the function `d()` does *not* reference the object:

```
class MyClass {
    private $a = 1;
    public function d() {
        return $this->a;
    }
}

$object = new MyClass();
var_dump(MyClass::d());   // Error!
```

This causes a 'PHP Fatal error: Uncaught Error: Using $this when not in object context'

These operators have *left* associativity, which can be used for 'chaining':

```
class MyClass {
    private $a = 1;

    public function add(int $a) {
        $this->a += $a;
        return $this;
    }

    public function get() {
        return $this->a;
    }
}

$object = new MyClass();
var_dump($object->add(4)->get());  // int(5)
```

These operators have the highest precedence (they are not even mentioned in the manual), even higher that `clone`. Thus:

```
class MyClass {
    private $a = 0;
    public function add(int $a) {
        $this->a += $a;
        return $this;
    }
    public function get() {
        return $this->a;
    }
}

$o1 = new MyClass();
$o2 = clone $o1->add(2);
var_dump($o1->get()); // int(2)
var_dump($o2->get()); // int(2)
```

The value of `$o1` is added to *before* the object is cloned!

Note that using parentheses to influence precedence did not work in PHP version 5 and older (it does in PHP 7):

```php
// using the class MyClass from the previous code
$o1 = new MyClass();
$o2 = (clone $o1)->add(2);  // Error in PHP 5 and before, fine in PHP 7
var_dump($o1->get()); // int(0) in PHP 7
var_dump($o2->get()); // int(2) in PHP 7
```

Section 10.9: Combined Assignment (+= etc)

The combined assignment operators are a shortcut for an operation on some variable and subsequently assigning this new value to that variable.

Arithmetic:

```php
$a = 1;   // basic assignment
$a += 2; // read as '$a = $a + 2'; $a now is (1 + 2) => 3
$a -= 1; // $a now is (3 - 1) => 2
$a *= 2; // $a now is (2 * 2) => 4
$a /= 2; // $a now is (16 / 2) => 8
$a %= 5; // $a now is (8 % 5) => 3 (modulus or remainder)

// array +
$arrOne = array(1);
$arrTwo = array(2);
$arrOne += $arrTwo;
```

Processing Multiple Arrays Together

```php
$a **= 2; // $a now is (4 ** 2) => 16 (4 raised to the power of 2)
```

Combined concatenation and assignment of a string:

```php
$a = "a";
$a .= "b"; // $a => "ab"
```

Combined binary bitwise assignment operators:

```php
$a = 0b00101010;  // $a now is 42
$a &= 0b00001111; // $a now is (00101010 & 00001111) => 00001010 (bitwise and)
$a |= 0b00100010; // $a now is (00001010 | 00100010) => 00101010 (bitwise or)
$a ^= 0b10000010; // $a now is (00101010 ^ 10000010) => 10101000 (bitwise xor)
$a >>= 3;         // $a now is (10101000 >> 3) => 00010101 (shift right by 3)
$a <<= 1;         // $a now is (00010101 << 1) => 00101010 (shift left by 1)
```

Section 10.10: Altering operator precedence (with parentheses)

The order in which operators are evaluated is determined by the *operator precedence* (see also the Remarks section).

In

```php
$a = 2 * 3 + 4;
```

`$a` gets a value of 10 because `2 * 3` is evaluated first (multiplication has a higher precedence than addition) yielding a sub-result of `6 + 4`, which equals to 10.

The precedence can be altered using parentheses: in

```
$a = 2 * (3 + 4);
```

`$a` gets a value of 14 because `(3 + 4)` is evaluated first.

Section 10.11: Basic Assignment (=)

```
$a = "some string";
```

results in `$a` having the value `some string`.

The result of an assignment expression is the value being assigned. **Note that a single equal sign `=` is NOT for comparison!**

```
$a = 3;
$b = ($a = 5);
```

does the following:

1. Line 1 assigns `3` to `$a`.
2. Line 2 assigns `5` to `$a`. This expression yields value `5` as well.
3. Line 2 then assigns the result of the expression in parentheses (`5`) to `$b`.

Thus: both `$a` and `$b` now have value `5`.

Section 10.12: Association

Left association

If the preceedence of two operators is equal, the associativity determines the grouping (see also the Remarks section):

```
$a = 5 * 3 % 2; // $a now is (5 * 3) % 2 => (15 % 2) => 1
```

`*` and `%` have equal precedence and **left** associativity. Because the multiplication occurs first (left), it is grouped.

```
$a = 5 % 3 * 2; // $a now is (5 % 3) * 2 => (2 * 2) => 4
```

Now, the modulus operator occurs first (left) and is thus grouped.

Right association

```
$a = 1;
$b = 1;
$a = $b += 1;
```

Both `$a` and `$b` now have value `2` because `$b += 1` is grouped and then the result (`$b` is `2`) is assigned to `$a`.

Section 10.13: Comparison Operators

Equality

For basic equality testing, the equal operator `==` is used. For more comprehensive checks, use the identical operator `===`.

The identical operator works the same as the equal operator, requiring its operands have the same value, but also requires them to have the same data type.

For example, the sample below will display 'a and b are equal', but not 'a and b are identical'.

```
$a = 4;
$b = '4';
if ($a == $b) {
    echo 'a and b are equal'; // this will be printed
}
if ($a === $b) {
    echo 'a and b are identical'; // this won't be printed
}
```

When using the equal operator, numeric strings are cast to integers.

Comparison of objects

`===` compares two objects by checking if they are exactly the **same instance**. This means that `new stdClass() === new stdClass()` resolves to false, even if they are created in the same way (and have the exactly same values).

`==` compares two objects by recursively checking if they are equal (*deep equals*). That means, for `$a == $b`, if `$a` and `$b` are:

1. of the same class
2. have the same properties set, including dynamic properties
3. for each property `$property` set, `$a->property == $b->property` is true (hence recursively checked).

Other commonly used operators

They include:

1. Greater Than (`>`)
2. Lesser Than (`<`)
3. Greater Than Or Equal To (`>=`)
4. Lesser Than Or Equal To (`<=`)
5. Not Equal To (`!=`)
6. Not Identically Equal To (`!==`)

1. **Greater Than**: `$a > $b`, returns `true` if $a's value is greater than of $b, otherwise returns false.

Example:

```
var_dump(5 > 2); // prints bool(true)
var_dump(2 > 7); // prints bool(false)
```

2. **Lesser Than**: `$a < $b`, returns `true` if $a's value is smaller that of $b, otherwise returns false.

Example:

```
var_dump(5 < 2); // prints bool(false)
var_dump(1 < 10); // prints bool(true)
```

3. **Greater Than Or Equal To**: `$a >= $b`, returns `true` if $a's value is either greater than of $b or equal to $b, otherwise returns `false`.

Example:

```
var_dump(2 >= 2); // prints bool(true)
var_dump(6 >= 1); // prints bool(true)
var_dump(1 >= 7); // prints bool(false)
```

4. **Smaller Than Or Equal To**: `$a <= $b`, returns `true` if $a's value is either smaller than of $b or equal to $b, otherwise returns `false`.

Example:

```
var_dump(5 <= 5); // prints bool(true)
var_dump(5 <= 8); // prints bool(true)
var_dump(9 <= 1); // prints bool(false)
```

5/6. **Not Equal/Identical To:** To rehash the earlier example on equality, the sample below will display 'a and b are not identical', but not 'a and b are not equal'.

```
$a = 4;
$b = '4';
if ($a != $b) {
    echo 'a and b are not equal'; // this won't be printed
}
if ($a !== $b) {
    echo 'a and b are not identical'; // this will be printed
}
```

Section 10.14: Bitwise Operators

Prefix bitwise operators

Bitwise operators are like logical operators but executed per bit rather than per boolean value.

```
// bitwise NOT ~: sets all unset bits and unsets all set bits
printf("%'06b", ~0b110110); // 001001
```

Bitmask-bitmask operators

Bitwise AND `&`: a bit is set only if it is set in both operands

```
printf("%'06b", 0b110101 & 0b011001); // 010001
```

Bitwise OR `|`: a bit is set if it is set in either or both operands

```
printf("%'06b", 0b110101 | 0b011001); // 111101
```

Bitwise XOR `^`: a bit is set if it is set in one operand and not set in another operand, i.e. only if that bit is in different state in the two operands

```
printf("%'06b", 0b110101 ^ 0b011001); // 101100
```

Example uses of bitmasks

These operators can be used to manipulate bitmasks. For example:

```
file_put_contents("file.log", LOCK_EX | FILE_APPEND);
```

Here, the `|` operator is used to combine the two bitmasks. Although `+` has the same effect, `|` emphasizes that you

are combining bitmasks, not adding two normal scalar integers.

```
class Foo{
    const OPTION_A = 1;
    const OPTION_B = 2;
    const OPTION_C = 4;
    const OPTION_A = 8;

    private $options = self::OPTION_A | self::OPTION_C;

    public function toggleOption(int $option){
        $this->options ^= $option;
    }

    public function enable(int $option){
        $this->options |= $option; // enable $option regardless of its original state
    }

    public function disable(int $option){
        $this->options &= ~$option; // disable $option regardless of its original state,
                                    // without affecting other bits
    }

    /** returns whether at least one of the options is enabled */
    public function isOneEnabled(int $options) : bool{
        return $this->options & $option !== 0;
        // Use !== rather than >, because
        // if $options is about a high bit, we may be handling a negative integer
    }

    /** returns whether all of the options are enabled */
    public function areAllEnabled(int $options) : bool{
        return ($this->options & $options) === $options;
        // note the parentheses; beware the operator precedence
    }
}
```

This example (assuming `$option` always only contain one bit) uses:

- the `^` operator to conveniently toggle bitmasks.
- the `|` operator to set a bit neglecting its original state or other bits
- the `~` operator to convert an integer with only one bit set into an integer with only one bit not set
- the `&` operator to unset a bit, using these properties of `&`:
    - Since `&=` with a set bit will not do anything (`(1 & 1) === 1`, `(0 & 1) === 0`), doing `&=` with an integer with only one bit not set will only unset that bit, not affecting other bits.
    - `&=` with an unset bit will unset that bit (`(1 & 0) === 0`, `(0 & 0) === 0`)
- Using the `&` operator with another bitmask will filter away all other bits not set in that bitmask.
    - If the output has any bits set, it means that any one of the options are enabled.
    - If the output has all bits of the bitmask set, it means that all of the options in the bitmask are enabled.

Bear in mind that these comparison operators: (`< > <= >= == === != !== <> <=>`) have higher precedence than these bitmask-bitmask operators: (`| ^ &`). As bitwise results are often compared using these comparison operators, this is a common pitfall to be aware of.

Bit-shifting operators

Bitwise left shift `<<`: shift all bits to the left (more significant) by the given number of steps and discard the bits exceeding the int size

`<< $x` is equivalent to unsetting the highest $x bits and multiplying by the $xth power of 2

```
printf("%'08b", 0b00001011<< 2); // 00101100

assert(PHP_INT_SIZE === 4); // a 32-bit system
printf("%x, %x", 0x5FFFFFFF << 2, 0x1FFFFFFF << 4); // 7FFFFFFC, FFFFFFFF
```

Bitwise right shift `>>`: discard the lowest shift and shift the remaining bits to the right (less significant)

`>> $x` is equivalent to dividing by the $xth power of 2 and discard the non-integer part

```
printf("%x", 0xFFFFFFFF >> 3); // 1FFFFFFF
```

Example uses of bit shifting:

Fast division by 16 (better performance than `/= 16`)

```
$x >>= 4;
```

On 32-bit systems, this discards all bits in the integer, setting the value to 0. On 64-bit systems, this unsets the most significant 32 bits and keep the least

```
$x = $x << 32 >> 32;
```

significant 32 bits, equivalent to `$x & 0xFFFFFFFF`

Note: In this example, `printf("%'06b")` is used. It outputs the value in 6 binary digits.

Section 10.15: instanceof (type operator)

For checking whether some object is of a certain class, the (binary) `instanceof` operator can be used since PHP version 5.

The first (left) parameter is the object to test. If this variable is not an object, `instanceof` always returns `false`. If a constant expression is used, an error is thrown.

The second (right) parameter is the class to compare with. The class can be provided as the class name itself, a string variable containing the class name (not a string constant!) or an object of that class.

```
class MyClass {
}

$o1 = new MyClass();
$o2 = new MyClass();
$name = 'MyClass';

// in the cases below, $a gets boolean value true
$a = $o1 instanceof MyClass;
$a = $o1 instanceof $name;
$a = $o1 instanceof $o2;

// counter examples:
$b = 'b';
$a = $o1 instanceof 'MyClass'; // parse error: constant not allowed
$a = false instanceof MyClass; // fatal error: constant not allowed
$a = $b instanceof MyClass;    // false ($b is not an object)
```

`instanceof` can also be used to check whether an object is of some class which extends another class or implements some interface:

```
interface MyInterface {
}

class MySuperClass implements MyInterface {
}

class MySubClass extends MySuperClass {
}

$o = new MySubClass();

// in the cases below, $a gets boolean value true
$a = $o instanceof MySubClass;
$a = $o instanceof MySuperClass;
$a = $o instanceof MyInterface;
```

To check whether an object is *not* of some class, the not operator (`!`) can be used:

```
class MyClass {
}

class OtherClass {
}

$o = new MyClass();
$a = !$o instanceof OtherClass; // true
```

Note that parentheses around `$o instanceof MyClass` are not needed because `instanceof` has higher precedence than `!`, although it may make the code better readable *with* parentheses.

Caveats

If a class does not exist, the registered autoload functions are called to try to define the class (this is a topic outside the scope of this part of the Documentation!). In PHP versions before 5.1.0, the `instanceof` operator would also trigger these calls, thus actually defining the class (and if the class could not be defined, a fatal error would occur). To avoid this, use a string:

```
// only PHP versions before 5.1.0!
class MyClass {
}

$o = new MyClass();
$a = $o instanceof OtherClass; // OtherClass is not defined!
// if OtherClass can be defined in a registered autoloader, it is actually
// loaded and $a gets boolean value false ($o is not a OtherClass)
// if OtherClass can not be defined in a registered autoloader, a fatal
// error occurs.

$name = 'YetAnotherClass';
$a = $o instanceof $name; // YetAnotherClass is not defined!
// $a simply gets boolean value false, YetAnotherClass remains undefined.
```

As of PHP version 5.1.0, the registered autoloaders are not called anymore in these situations.

Older versions of PHP (before 5.0)

In older versions of PHP (before 5.0), the `is_a` function can be used to determine wether an object is of some class. This function was deprecated in PHP version 5 and undeprecated in PHP version 5.3.0.

Chapter 11: References

Section 11.1: Assign by Reference

This is the first phase of referencing. Essentially when you assign by reference, you're allowing two variables to share the same value as such.

```
$foo = &$bar;
```

`$foo` and `$bar` are equal here. They **do not** point to one another. They point to the same place (*the "value"*).

You can also assign by reference within the `array()` language construct. While not strictly being an assignment by reference.

```
$foo = 'hi';
$bar = array(1, 2);
$array = array(&$foo, &$bar[0]);
```

> **Note**, however, that references inside arrays are potentially dangerous. Doing a normal (not by reference) assignment with a reference on the right side does not turn the left side into a reference, but references inside arrays are preserved in these normal assignments. This also applies to function calls where the array is passed by value.

Assigning by reference is not only limited to variables and arrays, they are also present for functions and all "pass-by-reference" associations.

```
function incrementArray(&$arr) {
    foreach ($arr as &$val) {
        $val++;
    }
}

function &getArray() {
    static $arr = [1, 2, 3];
    return $arr;
}

incrementArray(getArray());
var_dump(getArray()); // prints an array [2, 3, 4]
```

Assignment is key within the function definition as above. You **can not** pass an expression by reference, only a value/variable. Hence the instantiation of `$a` in `bar()`.

Section 11.2: Return by Reference

Occasionally there comes time for you to implicitly return-by-reference.

> Returning by reference is useful when you want to use a function to find to which variable a reference should be bound. Do not use return-by-reference to increase performance. The engine will automatically optimize this on its own. Only return references when you have a valid technical reason to do so.

Taken from the PHP Documentation for Returning By Reference.

There are many different forms return by reference can take, including the following example:

```
function parent(&$var) {
    echo $var;
    $var = "updated";
}

function &child() {
    static $a = "test";
    return $a;
}

parent(child()); // returns "test"
parent(child()); // returns "updated"
```

Return by reference is not only limited to function references. You also have the ability to implicitly call the function:

```
function &myFunction() {
    static $a = 'foo';
    return $a;
}

$bar = &myFunction();
$bar = "updated"
echo myFunction();
```

You cannot directly *reference* a function call, it has to be assigned to a variable before harnessing it. To see how that works, simply try `echo &myFunction();`.

Notes

- You are required to specify a reference (`&`) in both places you intend on using it. That means, for your function definition (`function &myFunction() {...}`) and in the calling reference (`function callFunction(&$variable) {...` or `&myFunction();`).
- You can only return a variable by reference. Hence the instantiation of `$a` in the example above. This means you can not return an expression, otherwise an `E_NOTICE` PHP error will be generated (*`Notice: Only variable references should be returned by reference in ......`*).
- Return by reference does have legitimate use cases, but I should warn that they should be used sparingly, only after exploring all other potential options of achieving the same goal.

Section 11.3: Pass by Reference

This allows you to pass a variable by reference to a function or element that allows you to modify the original variable.

Passing-by-reference is not limited to variables only, the following can also be passed by reference:

- New statements, e.g. `foo(new SomeClass)`
- References returned from functions

Arrays

A common use of "passing-by-reference" is to modify initial values within an array without going to the extent of

creating new arrays or littering your namespace. Passing-by-reference is as simple as preceding/prefixing the variable with an `&` => `&$myElement`.

Below is an example of harnessing an element from an array and simply adding 1 to its initial value.

```
$arr = array(1, 2, 3, 4, 5);

foreach($arr as &$num) {
    $num++;
}
```

Now when you harness any element within `$arr`, the original element will be updated as the reference was increased. You can verify this by:

```
print_r($arr);
```

> **Note**
>
> You should take note when harnessing pass by reference within loops. At the end of the above loop, `$num` still holds a reference to the last element of the array. Assigning it post loop will end up manipulating the last array element! You can ensure this doesn't happen by `unset()`'ing it post-loop:
>
> ```
> $myArray = array(1, 2, 3, 4, 5);
>
> foreach($myArray as &$num) {
>    $num++;
> }
> unset($num);
> ```
>
> The above will ensure you don't run into any issues. An example of issues that could relate from this is present in this question on StackOverflow.

Functions

Another common usage for passing-by-reference is within functions. Modifying the original variable is as simple as:

```
$var = 5;
// define
function add(&$var) {
    $var++;
}
// call
add($var);
```

Which can be verified by `echo`'ing the original variable.

```
echo $var;
```

There are various restrictions around functions, as noted below from the PHP docs:

> **Note:** There is no reference sign on a function call - only on function definitions. Function definitions alone are enough to correctly pass the argument by reference. As of PHP 5.3.0, you will get a warning

> saying that "call-time pass-by-reference" is deprecated when you use & in foo(&$a);. And as of PHP 5.4.0, call-time pass-by-reference was removed, so using it will raise a fatal error.

Chapter 12: Arrays

Parameter	Detail
Key	The key is the unique identifier and index of an array. It may be a `string` or an `integer`. Therefore, valid keys would be `'foo'`, `'5'`, `10`, `'a2b'`, `...`
Value	For each `key` there is a corresponding value (`null` otherwise *and a notice is emitted upon access*). The value has no restrictions on the input type.

An array is a data structure that stores an arbitrary number of values in a single value. An array in PHP is actually an ordered map, where map is a type that associates values to keys.

Section 12.1: Initializing an Array

An array can be initialized empty:

```
// An empty array
$foo = array();

// Shorthand notation available since PHP 5.4
$foo = [];
```

An array can be initialized and preset with values:

```
// Creates a simple array with three strings
$fruit = array('apples', 'pears', 'oranges');

// Shorthand notation available since PHP 5.4
$fruit = ['apples', 'pears', 'oranges'];
```

An array can also be initialized with custom indexes *(also called an associative array)*:

```
// A simple associative array
$fruit = array(
   'first'  => 'apples',
   'second' => 'pears',
   'third'  => 'oranges'
);

// Key and value can also be set as follows
$fruit['first'] = 'apples';

// Shorthand notation available since PHP 5.4
$fruit = [
    'first'  => 'apples',
    'second' => 'pears',
    'third'  => 'oranges'
];
```

If the variable hasn't been used before, PHP will create it automatically. While convenient, this might make the code harder to read:

```
$foo[] = 1;     // Array( [0] => 1 )
```

```
$bar[][] = 2;   // Array( [0] => Array( [0] => 2 ) )
```

The index will usually continue where you left off. PHP will try to use numeric strings as integers:

```
$foo = [2 => 'apple', 'melon'];  // Array( [2] => apple, [3] => melon )
$foo = ['2' => 'apple', 'melon']; // same as above
$foo = [2 => 'apple', 'this is index 3 temporarily', '3' => 'melon']; // same as above! The last
entry will overwrite the second!
```

To initialize an array with fixed size you can use SplFixedArray:

```
$array = new SplFixedArray(3);

$array[0] = 1;
$array[1] = 2;
$array[2] = 3;
$array[3] = 4; // RuntimeException

// Increase the size of the array to 10
$array->setSize(10);
```

Note: An array created using `SplFixedArray` has a reduced memory footprint for large sets of data, but the keys must be integers.

To initialize an array with a dynamic size but with `n` non empty elements (e.g. a placeholder) you can use a loop as follows:

```
$myArray = array();
$sizeOfMyArray = 5;
$fill = 'placeholder';

for ($i = 0; $i < $sizeOfMyArray; $i++) {
    $myArray[] = $fill;
}

// print_r($myArray); results in the following:
// Array ( [0] => placeholder [1] => placeholder [2] => placeholder [3] => placeholder [4] =>
placeholder )
```

If all your placeholders are the same then you can also create it using the function array_fill():

> array array_fill (int $start_index , int $num , mixed $value)

This creates and returns an array with `num` entries of `value`, keys starting at `start_index`.

Note: If the `start_index` is negative it will start with the negative index and continue from 0 for the following elements.

```
$a = array_fill(5, 6, 'banana'); // Array ( [5] => banana, [6] => banana, ..., [10] => banana)
$b = array_fill(-2, 4, 'pear'); // Array ( [-2] => pear, [0] => pear, ..., [2] => pear)
```

Conclusion: With array_fill() you are more limited for what you can actually do. The loop is more flexible and

opens you a wider range of opportunities.

Whenever you want an array filled with a range of numbers (e.g. 1-4) you could either append every single element to an array or use the `range()` function:

> array range (mixed $start , mixed $end [, number $step = 1])

This function creates an array containing a range of elements. The first two parameters are required, where they set the start and end points of the (inclusive) range. The third parameter is optional and defines the size of the steps being taken. Creating a `range` from `0` to `4` with a `stepsize` of `1`, the resulting array would consist of the following elements: `0`, `1`, `2`, `3`, and `4`. If the step size is increased to `2` (i.e. `range(0, 4, 2)`) then the resulting array would be: `0`, `2`, and `4`.

```
$array = [];
$array_with_range = range(1, 4);

for ($i = 1; $i <= 4; $i++) {
    $array[] = $i;
}

print_r($array); // Array ( [0] => 1 [1] => 2 [2] => 3 [3] => 4 )
print_r($array_with_range); // Array ( [0] => 1 [1] => 2 [2] => 3 [3] => 4 )
```

`range` can work with integers, floats, booleans (which become casted to integers), and strings. Caution should be taken, however, when using floats as arguments due to the floating point precision problem.

Section 12.2: Check if key exists

Use `array_key_exists()` or `isset()` or `!empty()`:

```
$map = [
    'foo' => 1,
    'bar' => null,
    'foobar' => '',
];

array_key_exists('foo', $map); // true
isset($map['foo']); // true
!empty($map['foo']); // true

array_key_exists('bar', $map); // true
isset($map['bar']); // false
!empty($map['bar']); // false
```

Note that `isset()` treats a **`null`** valued element as non-existent. Whereas `!empty()` does the same for any element that equals **`false`** (using a weak comparision; for example, **`null`**, `''` and `0` are all treated as false by `!empty()`). While `isset($map['foobar']);` is **`true`**, `!empty($map['foobar'])` is **`false`**. This can lead to mistakes (for example, it is easy to forget that the string `'0'` is treated as false) so use of `!empty()` is often frowned upon.

Note also that `isset()` and `!empty()` will work (and return false) if `$map` is not defined at all. This makes them somewhat error-prone to use:

```
// Note "long" vs "lang", a tiny typo in the variable name.
$my_array_with_a_long_name = ['foo' => true];
array_key_exists('foo', $my_array_with_a_lang_name); // shows a warning
isset($my_array_with_a_lang_name['foo']); // returns false
```

You can also check for ordinal arrays:

```
$ord = ['a', 'b']; // equivalent to [0 => 'a', 1 => 'b']

array_key_exists(0, $ord); // true
array_key_exists(2, $ord); // false
```

Note that `isset()` has better performance than `array_key_exists()` as the latter is a function and the former a language construct.

You can also use `key_exists()`, which is an alias for `array_key_exists()`.

Section 12.3: Validating the array type

The function `is_array()` returns true if a variable is an array.

```
$integer = 1337;
$array = [1337, 42];

is_array($integer); // false
is_array($array); // true
```

You can type hint the array type in a function to enforce a parameter type; passing anything else will result in a fatal error.

```
function foo (array $array) { /* $array is an array */ }
```

You can also use the `gettype()` function.

```
$integer = 1337;
$array = [1337, 42];

gettype($integer) === 'array'; // false
gettype($array) === 'array'; // true
```

Section 12.4: Creating an array of variables

```
$username = 'Hadibut';
$email = 'hadibut@example.org';

$variables = compact('username', 'email');
// $variables is now ['username' => 'Hadibut', 'email' => 'hadibut@example.org']
```

This method is often used in frameworks to pass an array of variables between two components.

Section 12.5: Checking if a value exists in array

The function `in_array()` returns true if an item exists in an array.

```
$fruits = ['banana', 'apple'];

$foo = in_array('banana', $fruits);
// $foo value is true

$bar = in_array('orange', $fruits);
```

```
// $bar value is false
```

You can also use the function `array_search()` to get the key of a specific item in an array.

```
$userdb = ['Sandra Shush', 'Stefanie Mcmohn', 'Michael'];
$pos = array_search('Stefanie Mcmohn', $userdb);
if ($pos !== false) {
    echo "Stefanie Mcmohn found at $pos";
}
```

PHP 5.x `Version ≥ 5.5`

In PHP 5.5 and later you can use `array_column()` in conjunction with `array_search()`.

This is particularly useful for checking if a value exists in an associative array:

```
$userdb = [
    [
        "uid" => '100',
        "name" => 'Sandra Shush',
        "url" => 'urlof100',
    ],
    [
        "uid" => '5465',
        "name" => 'Stefanie Mcmohn',
        "pic_square" => 'urlof100',
    ],
    [
        "uid" => '40489',
        "name" => 'Michael',
        "pic_square" => 'urlof40489',
    ]
];

$key = array_search(40489, array_column($userdb, 'uid'));
```

Section 12.6: ArrayAccess and Iterator Interfaces

Another useful feature is accessing your custom object collections as arrays in PHP. There are two interfaces available in PHP (>=5.0.0) core to support this: `ArrayAccess` and `Iterator`. The former allows you to access your custom objects as array.

ArrayAccess

Assume we have a user class and a database table storing all the users. We would like to create a `UserCollection` class that will:

1. allow us to address certain user by their username unique identifier
2. perform basic (not all CRUD, but at least Create, Retrieve and Delete) operations on our users collection

Consider the following source (hereinafter we're using short array creation syntax `[]` available since version 5.4):

```
class UserCollection implements ArrayAccess {
    protected $_conn;

    protected $_requiredParams = ['username','password','email'];

    public function __construct() {
        $config = new Configuration();
```

```
        $connectionParams = [
            //your connection to the database
        ];

        $this->_conn = DriverManager::getConnection($connectionParams, $config);
    }

    protected function _getByUsername($username) {
        $ret = $this->_conn->executeQuery('SELECT * FROM `User` WHERE `username` IN (?)',
            [$username]
        )->fetch();

        return $ret;
    }

    // START of methods required by ArrayAccess interface
    public function offsetExists($offset) {
        return (bool) $this->_getByUsername($offset);
    }

    public function offsetGet($offset) {
        return $this->_getByUsername($offset);
    }

    public function offsetSet($offset, $value) {
        if (!is_array($value)) {
            throw new \Exception('value must be an Array');
        }

        $passed = array_intersect(array_values($this->_requiredParams), array_keys($value));
        if (count($passed) < count($this->_requiredParams)) {
            throw new \Exception('value must contain at least the following params: ' .
implode(',', $this->_requiredParams));
        }
        $this->_conn->insert('User', $value);
    }

    public function offsetUnset($offset) {
        if (!is_string($offset)) {
            throw new \Exception('value must be the username to delete');
        }
        if (!$this->offsetGet($offset)) {
            throw new \Exception('user not found');
        }
        $this->_conn->delete('User', ['username' => $offset]);
    }
    // END of methods required by ArrayAccess interface
}
```

then we can:

```
$users = new UserCollection();

var_dump(empty($users['testuser']),isset($users['testuser']));
$users['testuser'] = ['username' => 'testuser',
                      'password' => 'testpassword',
                      'email'    => 'test@test.com'];
var_dump(empty($users['testuser']), isset($users['testuser']), $users['testuser']);
unset($users['testuser']);
var_dump(empty($users['testuser']), isset($users['testuser']));
```

which will output the following, assuming there was no `testuser` before we launched the code:

```
bool(true)
bool(false)
bool(false)
bool(true)
array(17) {
  ["username"]=>
  string(8) "testuser"
  ["password"]=>
  string(12) "testpassword"
  ["email"]=>
  string(13) "test@test.com"
}
bool(true)
bool(false)
```

IMPORTANT: `offsetExists` is not called when you check existence of a key with `array_key_exists` function. So the following code will output `false` twice:

```
var_dump(array_key_exists('testuser', $users));
$users['testuser'] = ['username' => 'testuser',
                      'password' => 'testpassword',
                      'email'    => 'test@test.com'];
var_dump(array_key_exists('testuser', $users));
```

Iterator

Let's extend our class from above with a few functions from `Iterator` interface to allow iterating over it with `foreach` and `while`.

First, we need to add a property holding our current index of iterator, let's add it to the class properties as `$_position`:

```
// iterator current position, required by Iterator interface methods
protected $_position = 1;
```

Second, let's add `Iterator` interface to the list of interfaces being implemented by our class:

```
class UserCollection implements ArrayAccess, Iterator {
```

then add the required by the interface functions themselves:

```
// START of methods required by Iterator interface
public function current () {
    return $this->_getById($this->_position);
}
public function key () {
    return $this->_position;
}
public function next () {
    $this->_position++;
}
public function rewind () {
    $this->_position = 1;
}
public function valid () {
    return null !== $this->_getById($this->_position);
```

```
}
// END of methods required by Iterator interface
```

So all in all here is complete source of the class implementing both interfaces. Note that this example is not perfect, because the IDs in the database may not be sequential, but this was written just to give you the main idea: you can address your objects collections in any possible way by implementing `ArrayAccess` and `Iterator` interfaces:

```
class UserCollection implements ArrayAccess, Iterator {
    // iterator current position, required by Iterator interface methods
    protected $_position = 1;

    // <add the old methods from the last code snippet here>

    // START of methods required by Iterator interface
    public function current () {
        return $this->_getById($this->_position);
    }
    public function key () {
        return $this->_position;
    }
    public function next () {
        $this->_position++;
    }
    public function rewind () {
        $this->_position = 1;
    }
    public function valid () {
        return null !== $this->_getById($this->_position);
    }
    // END of methods required by Iterator interface
}
```

and a foreach looping through all user objects:

```
foreach ($users as $user) {
    var_dump($user['id']);
}
```

which will output something like

```
string(2) "1"
string(2) "2"
string(2) "3"
string(2) "4"
...
```

Chapter 13: Array iteration

Section 13.1: Iterating multiple arrays together

Sometimes two arrays of the same length need to be iterated together, for example:

```
$people = ['Tim', 'Tony', 'Turanga'];
$foods = ['chicken', 'beef', 'slurm'];
```

`array_map` is the simplest way to accomplish this:

```
array_map(function($person, $food) {
    return "$person likes $food\n";
}, $people, $foods);
```

which will output:

```
Tim likes chicken
Tony likes beef
Turanga likes slurm
```

This can be done through a common index:

```
assert(count($people) === count($foods));
for ($i = 0; $i < count($people); $i++) {
    echo "$people[$i] likes $foods[$i]\n";
}
```

If the two arrays don't have the incremental keys, `array_values($array)[$i]` can be used to replace `$array[$i]`.

If both arrays have the same order of keys, you can also use a foreach-with-key loop on one of the arrays:

```
foreach ($people as $index => $person) {
    $food = $foods[$index];
    echo "$person likes $food\n";
}
```

Separate arrays can only be looped through if they are the same length and also have the same key name. This means if you don't supply a key and they are numbered, you will be fine, or if you name the keys and put them in the same order in each array.

You can also use `array_combine`.

```
$combinedArray = array_combine($people, $foods);
// $combinedArray = ['Tim' => 'chicken', 'Tony' => 'beef', 'Turanga' => 'slurm'];
```

Then you can loop through this by doing the same as before:

```
foreach ($combinedArray as $person => $meal) {
    echo "$person likes $meal\n";
}
```

Section 13.2: Using an incremental index

This method works by incrementing an integer from 0 to the greatest index in the array.

```
$colors = ['red', 'yellow', 'blue', 'green'];
for ($i = 0; $i < count($colors); $i++) {
    echo 'I am the color ' . $colors[$i] . '<br>';
}
```

This also allows iterating an array in reverse order without using `array_reverse`, which may result in overhead if the array is large.

```
$colors = ['red', 'yellow', 'blue', 'green'];
for ($i = count($colors) - 1; $i >= 0; $i--) {
    echo 'I am the color ' . $colors[$i] . '<br>';
}
```

You can skip or rewind the index easily using this method.

```
$array = ["alpha", "beta", "gamma", "delta", "epsilon"];
for ($i = 0; $i < count($array); $i++) {
    echo $array[$i], PHP_EOL;
    if ($array[$i] === "gamma") {
        $array[$i] = "zeta";
        $i -= 2;
    } elseif ($array[$i] === "zeta") {
        $i++;
    }
}
```

Output:

```
alpha
beta
gamma
beta
zeta
epsilon
```

For arrays that do not have incremental indices (including arrays with indices in reverse order, e.g. `[1 => "foo", 0 => "bar"]`, `["foo" => "f", "bar" => "b"]`), this cannot be done directly. `array_values` or `array_keys` can be used instead:

```
$array = ["a" => "alpha", "b" => "beta", "c" => "gamma", "d" => "delta"];
$keys = array_keys($array);
for ($i = 0; $i < count($array); $i++) {
    $key = $keys[$i];
    $value = $array[$key];
    echo "$value is $key\n";
}
```

Section 13.3: Using internal array pointers

Each array instance contains an internal pointer. By manipulating this pointer, different elements of an array can be retrieved from the same call at different times.

Using `each`

Each call to `each()` returns the key and value of the current array element, and increments the internal array pointer.

```
$array = ["f" => "foo", "b" => "bar"];
while (list($key, $value) = each($array)) {
    echo "$value begins with $key";
}
```

Using `next`

```
$array = ["Alpha", "Beta", "Gamma", "Delta"];
while (($value = next($array)) !== false) {
    echo "$value\n";
}
```

Note that this example assumes no elements in the array are identical to boolean `false`. To prevent such assumption, use `key` to check if the internal pointer has reached the end of the array:

```
$array = ["Alpha", "Beta", "Gamma", "Delta"];
while (key($array) !== null) {
    echo current($array) . PHP_EOL;
    next($array);
}
```

This also facilitates iterating an array without a direct loop:

```
class ColorPicker {
    private $colors = ["#FF0064", "#0064FF", "#64FF00", "#FF6400", "#00FF64", "#6400FF"];
    public function nextColor() : string {
        $result = next($colors);
        // if end of array reached
        if (key($colors) === null) {
            reset($colors);
        }
        return $result;
    }
}
```

Section 13.4: Using foreach

Direct loop

```
foreach ($colors as $color) {
    echo "I am the color $color<br>";
}
```

Loop with keys

```
$foods = ['healthy' => 'Apples', 'bad' => 'Ice Cream'];
foreach ($foods as $key => $food) {
    echo "Eating $food is $key";
}
```

Loop by reference

In the `foreach` loops in the above examples, modifying the value (`$color` or `$food`) directly doesn't change its value in the array. The `&` operator is required so that the value is a reference pointer to the element in the array.

```
$years = [2001, 2002, 3, 4];
foreach ($years as &$year) {
    if ($year < 2000) $year += 2000;
```

```
}
```

This is similar to:

```
$years = [2001, 2002, 3, 4];
for($i = 0; $i < count($years); $i++) { // these two lines
    $year = &$years[$i];                // are changed to foreach by reference
    if($year < 2000) $year += 2000;
}
```

Concurrency

PHP arrays can be modified in any ways during iteration without concurrency problems (unlike e.g. Java `List`s). If the array is iterated by reference, later iterations will be affected by changes to the array. Otherwise, the changes to the array will not affect later iterations (as if you are iterating a copy of the array instead). Compare looping by value:

```
$array = [0 => 1, 2 => 3, 4 => 5, 6 => 7];
foreach ($array as $key => $value) {
    if ($key === 0) {
        $array[6] = 17;
        unset($array[4]);
    }
    echo "$key => $value\n";
}
```

Output:

```
0 => 1
2 => 3
4 => 5
6 => 7
```

But if the array is iterated with reference,

```
$array = [0 => 1, 2 => 3, 4 => 5, 6 => 7];
foreach ($array as $key => &$value) {
    if ($key === 0) {
        $array[6] = 17;
        unset($array[4]);
    }
    echo "$key => $value\n";
}
```

Output:

```
0 => 1
2 => 3
6 => 17
```

The key-value set of `4 => 5` is no longer iterated, and `6 => 7` is changed to `6 => 17`.

Section 13.5: Using ArrayObject Iterator

Php arrayiterator allows you to modify and unset the values while iterating over arrays and objects.

Example:

```
$array = ['1' => 'apple', '2' => 'banana', '3' => 'cherry'];

$arrayObject = new ArrayObject($array);

$iterator = $arrayObject->getIterator();

for($iterator; $iterator->valid(); $iterator->next()) {
    echo $iterator->key() . ' => ' . $iterator->current() . "</br>";
}
```

Output:

```
1 => apple
2 => banana
3 => cherry
```

Chapter 14: Executing Upon an Array

Section 14.1: Applying a function to each element of an array

To apply a function to every item in an array, use `array_map()`. This will return a new array.

```
$array = array(1,2,3,4,5);
//each array item is iterated over and gets stored in the function parameter.
$newArray = array_map(function($item) {
    return $item + 1;
}, $array);
```

`$newArray` now is `array(2,3,4,5,6);`.

Instead of using an anonymous function, you could use a named function. The above could be written like:

```
function addOne($item) {
    return $item + 1;
}

$array = array(1, 2, 3, 4, 5);
$newArray = array_map('addOne', $array);
```

If the named function is a class method the call of the function has to include a reference to a class object the method belongs to:

```
class Example {
    public function addOne($item) {
        return $item + 1;
    }

    public function doCalculation() {
        $array = array(1, 2, 3, 4, 5);
        $newArray = array_map(array($this, 'addOne'), $array);
    }
}
```

Another way to apply a function to every item in an array is `array_walk()` and `array_walk_recursive()`. The callback passed into these functions take both the key/index and value of each array item. These functions will not return a new array, instead a boolean for success. For example, to print every element in a simple array:

```
$array = array(1, 2, 3, 4, 5);
array_walk($array, function($value, $key) {
    echo $value . ' ';
});
// prints "1 2 3 4 5"
```

The value parameter of the callback may be passed by reference, allowing you to change the value directly in the original array:

```
$array = array(1, 2, 3, 4, 5);
array_walk($array, function(&$value, $key) {
    $value++;
});
```

`$array` now is `array(2,3,4,5,6);`

For nested arrays, `array_walk_recursive()` will go deeper into each sub-array:

```
$array = array(1, array(2, 3, array(4, 5), 6);
array_walk_recursive($array, function($value, $key) {
    echo $value . ' ';
});
// prints "1 2 3 4 5 6"
```

Note: `array_walk` and `array_walk_recursive` let you change the value of array items, but not the keys. Passing the keys by reference into the callback is valid but has no effect.

Section 14.2: Split array into chunks

array_chunk() splits an array into chunks

Let's say we've following single dimensional array,

```
$input_array = array('a', 'b', 'c', 'd', 'e');
```

Now using **array_chunk()** on above PHP array,

```
$output_array = array_chunk($input_array, 2);
```

Above code will make chunks of 2 array elements and create a multidimensional array as follow.

```
Array
(
    [0] => Array
        (
            [0] => a
            [1] => b
        )

    [1] => Array
        (
            [0] => c
            [1] => d
        )

    [2] => Array
        (
            [0] => e
        )

)
```

If all the elements of the array is not evenly divided by the chunk size, last element of the output array will be remaining elements.

If we pass second argument as less then 1 then **E_WARNING** will be thrown and output array will be **NULL**.

Parameter	Details
$array (array)	Input array, the array to work on
$size (int)	Size of each chunk (Integer value)
$preserve_keys (boolean) (optional)	If you want output array to preserve the keys set it to **TRUE** otherwise **FALSE**.

Section 14.3: Imploding an array into string

`implode()` combines all the array values but looses all the key info:

```
$arr = ['a' => "AA", 'b' => "BB", 'c' => "CC"];

echo implode(" ", $arr); // AA BB CC
```

Imploding keys can be done using `array_keys()` call:

```
$arr = ['a' => "AA", 'b' => "BB", 'c' => "CC"];

echo implode(" ", array_keys($arr)); // a b c
```

Imploding keys with values is more complex but can be done using functional style:

```
$arr = ['a' => "AA", 'b' => "BB", 'c' => "CC"];

echo implode(" ", array_map(function($key, $val) {
    return "$key:$val"; // function that glues key to the value
}, array_keys($arr), $arr));

// Output: a:AA b:BB c:CC
```

Section 14.4: "Destructuring" arrays using list()

Use list() to quick assign a list of variable values into an array. See also compact()

```
// Assigns to $a, $b and $c the values of their respective array elements in          $array with
keys numbered from zero
list($a, $b, $c) = $array;
```

With PHP 7.1 (currently in beta) you will be able to use short list syntax:

```
// Assigns to $a, $b and $c the values of their respective array elements in $array with keys
numbered from zero
[$a, $b, $c] = $array;

// Assigns to $a, $b and $c the values of the array elements in $array with the keys "a", "b" and
"c", respectively
["a" => $a, "b" => $b, "c" => $c] = $array;
```

Section 14.5: array_reduce

`array_reduce` reduces array into a single value. Basically, The `array_reduce` will go through every item with the result from last iteration and produce new value to the next iteration.

Usage: `array_reduce ($array, function($carry, $item){...}, $defaul_value_of_first_carry)`

- $carry is the result from the last round of iteration.
- $item is the value of current position in the array.

Sum of array

```
$result = array_reduce([1, 2, 3, 4, 5], function($carry, $item){
    return $carry + $item;
```

```
});
```

result:15

The largest number in array

```
$result = array_reduce([10, 23, 211, 34, 25], function($carry, $item){
        return $item > $carry ? $item : $carry;
});
```

result:211

Is all item more than 100

```
$result = array_reduce([101, 230, 210, 341, 251], function($carry, $item){
        return $carry && $item > 100;
}, true); //default value must set true
```

result:true

Is any item less than 100

```
$result = array_reduce([101, 230, 21, 341, 251], function($carry, $item){
        return $carry || $item < 100;
}, false);//default value must set false
```

result:true

Like implode($array, $piece)

```
$result = array_reduce(["hello", "world", "PHP", "language"], function($carry, $item){
        return !$carry ? $item : $carry . "-" . $item ;
});
```

result:"hello-world-PHP-language"

if make a implode method, the source code will be:

```
function implode_method($array, $piece){
    return array_reduce($array, function($carry, $item) use ($piece) {
            return !$carry ? $item : ($carry . $piece . $item);
    });
}

$result = implode_method(["hello", "world", "PHP", "language"], "-");
```

result:"hello-world-PHP-language"

Section 14.6: Push a Value on an Array

There are two ways to push an element to an array: `array_push` and `$array[] =`

The array_push is used like this:

```
$array = [1,2,3];
$newArraySize = array_push($array, 5, 6); // The method returns the new size of the array
print_r($array); // Array is passed by reference, therefore the original array is modified to
```

```
contain the new elements
```

This code will print:

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 5
    [4] => 6
)
```

`$array[] =` is used like this:

```
$array = [1,2,3];
$array[] = 5;
$array[] = 6;
print_r($array);
```

This code will print:

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 5
    [4] => 6
)
```

Chapter 15: Manipulating an Array

Section 15.1: Filtering an array

In order to filter out values from an array and obtain a new array containing all the values that satisfy the filter condition, you can use the `array_filter` function.

Filtering non-empty values

The simplest case of filtering is to remove all "empty" values:

```
$my_array = [1,0,2,null,3,'',4,[],5,6,7,8];
$non_empties = array_filter($my_array); // $non_empties will contain [1,2,3,4,5,6,7,8];
```

Filtering by callback

This time we define our own filtering rule. Suppose we want to get only even numbers:

```
$my_array = [1,2,3,4,5,6,7,8];

$even_numbers = array_filter($my_array, function($number) {
    return $number % 2 === 0;
});
```

The `array_filter` function receives the array to be filtered as its first argument, and a callback defining the filter predicate as its second.

Version ≥ 5.6

Filtering by index

A third parameter can be provided to the `array_filter` function, which allows to tweak which values are passed to the callback. This parameter can be set to either `ARRAY_FILTER_USE_KEY` or `ARRAY_FILTER_USE_BOTH`, which will result in the callback receiving the key instead of the value for each element in the array, or both value and key as its arguments. For example, if you want to deal with indexes istead of values:

```
$numbers = [16,3,5,8,1,4,6];

$even_indexed_numbers = array_filter($numbers, function($index) {
    return $index % 2 === 0;
}, ARRAY_FILTER_USE_KEY);
```

Indexes in filtered array

Note that `array_filter` preserves the original array keys. A common mistake would be to try an use `for` loop over the filtered array:

```
<?php

$my_array = [1,0,2,null,3,'',4,[],5,6,7,8];
$filtered = array_filter($my_array);

error_reporting(E_ALL); // show all errors and notices

// innocently looking "for" loop
for ($i = 0; $i < count($filtered); $i++) {
   print $filtered[$i];
}
```

```
/*
Output:
1
Notice: Undefined offset: 1
2
Notice: Undefined offset: 3
3
Notice: Undefined offset: 5
4
Notice: Undefined offset: 7
*/
```

This happens because the values which were on positions 1 (there was `0`), 3 (`null`), 5 (empty string `''`) and 7 (empty array `[]`) were removed along with their corresponding index keys.

If you need to loop through the result of a filter on an indexed array, you should first call `array_values` on the result of `array_filter` in order to create a new array with the correct indexes:

```
$my_array = [1,0,2,null,3,'',4,[],5,6,7,8];
$filtered = array_filter($my_array);
$iterable = array_values($filtered);

error_reporting(E_ALL); // show all errors and notices

for ($i = 0; $i < count($iterable); $i++) {
   print $iterable[$i];
}

// No warnings!
```

Section 15.2: Removing elements from an array

To remove an element inside an array, e.g. the element with the index 1.

```
$fruit = array("bananas", "apples", "peaches");
unset($fruit[1]);
```

This will remove the apples from the list, but notice that `unset` does not change the indexes of the remaining elements. So `$fruit` now contains the indexes `0` and `2`.

For associative array you can remove like this:

```
$fruit = array('banana', 'one'=>'apple', 'peaches');

print_r($fruit);
/*
    Array
    (
        [0] => banana
        [one] => apple
        [1] => peaches
    )
*/

unset($fruit['one']);
```

Now $fruit is

```
print_r($fruit);

/*
Array
(
    [0] => banana
    [1] => peaches
)
*/
```

Note that

```
unset($fruit);
```

unsets the variable and thus removes the whole array, meaning none of its elements are accessible anymore.

Removing terminal elements

array_shift() - Shift an element off the beginning of array.

Example:

```
  $fruit = array("bananas", "apples", "peaches");
  array_shift($fruit);
  print_r($fruit);
```

Output:

```
 Array
(
    [0] => apples
    [1] => peaches
)
```

array_pop() - Pop the element off the end of array.

Example:

```
  $fruit = array("bananas", "apples", "peaches");
  array_pop($fruit);
  print_r($fruit);
```

Output:

```
 Array
(
    [0] => bananas
    [1] => apples
)
```

Section 15.3: Sorting an Array

There are several sort functions for arrays in php:

sort()

Sort an array in ascending order by value.

```php
$fruits = ['Zitrone', 'Orange', 'Banane', 'Apfel'];
sort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [0] => Apfel
    [1] => Banane
    [2] => Orange
    [3] => Zitrone
)
```

rsort()

Sort an array in descending order by value.

```php
$fruits = ['Zitrone', 'Orange', 'Banane', 'Apfel'];
rsort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [0] => Zitrone
    [1] => Orange
    [2] => Banane
    [3] => Apfel
)
```

asort()

Sort an array in ascending order by value and preserve the indices.

```php
$fruits = [1 => 'lemon', 2 => 'orange',  3 => 'banana', 4 => 'apple'];
asort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [4] => apple
    [3] => banana
    [1] => lemon
    [2] => orange
)
```

arsort()

Sort an array in descending order by value and preserve the indices.

```php
$fruits = [1 => 'lemon', 2 => 'orange',  3 => 'banana', 4 => 'apple'];
arsort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [2] => orange
    [1] => lemon
    [3] => banana
    [4] => apple
)
```

ksort()

Sort an array in ascending order by key

```
$fruits = ['d'=>'lemon', 'a'=>'orange', 'b'=>'banana', 'c'=>'apple'];
ksort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [a] => orange
    [b] => banana
    [c] => apple
    [d] => lemon
)
```

krsort()

Sort an array in descending order by key.

```
$fruits = ['d'=>'lemon', 'a'=>'orange', 'b'=>'banana', 'c'=>'apple'];
krsort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [d] => lemon
    [c] => apple
    [b] => banana
    [a] => orange
)
```

natsort()

Sort an array in a way a human being would do (natural order).

```
$files = ['File8.stack', 'file77.stack', 'file7.stack', 'file13.stack', 'File2.stack'];
natsort($files);
print_r($files);
```

results in

```
Array
(
    [4] => File2.stack
    [0] => File8.stack
    [2] => file7.stack
    [3] => file13.stack
```

```
    [1] => file77.stack
)
```

natcasesort()

Sort an array in a way a human being would do (natural order), but case intensive

```
$files = ['File8.stack', 'file77.stack', 'file7.stack', 'file13.stack', 'File2.stack'];
natcasesort($files);
print_r($files);
```

results in

```
Array
(
    [4] => File2.stack
    [2] => file7.stack
    [0] => File8.stack
    [3] => file13.stack
    [1] => file77.stack
)
```

shuffle()

Shuffles an array (sorted randomly).

```
$array = ['aa', 'bb', 'cc'];
shuffle($array);
print_r($array);
```

As written in the description it is random so here only one example in what it can result

```
Array
(
    [0] => cc
    [1] => bb
    [2] => aa
)
```

usort()

Sort an array with a user defined comparison function.

```
function compare($a, $b)
{
    if ($a == $b) {
        return 0;
    }
    return ($a < $b) ? -1 : 1;
}

$array = [3, 2, 5, 6, 1];
usort($array, 'compare');
print_r($array);
```

results in

```
Array
(
    [0] => 1
```

```
    [1] => 2
    [2] => 3
    [3] => 5
    [4] => 6
)
```

uasort()

Sort an array with a user defined comparison function and preserve the keys.

```
function compare($a, $b)
{
    if ($a == $b) {
        return 0;
    }
    return ($a < $b) ? -1 : 1;
}

$array = ['a' => 1, 'b' => -3, 'c' => 5, 'd' => 3, 'e' => -5];
uasort($array, 'compare');
print_r($array);
```

results in

```
Array
(
    [e] => -5
    [b] => -3
    [a] => 1
    [d] => 3
    [c] => 5
)
```

uksort()

Sort an array by keys with a user defined comparison function.

```
function compare($a, $b)
{
    if ($a == $b) {
        return 0;
    }
    return ($a < $b) ? -1 : 1;
}

$array = ['ee' => 1, 'g' => -3, '4' => 5, 'k' => 3, 'oo' => -5];

uksort($array, 'compare');
print_r($array);
```

results in

```
Array
(
    [ee] => 1
    [g] => -3
    [k] => 3
    [oo] => -5
    [4] => 5
)
```

Section 15.4: Whitelist only some array keys

When you want to allow only certain keys in your arrays, especially when the array comes from request parameters, you can use `array_intersect_key` together with `array_flip`.

```
$parameters = ['foo' => 'bar', 'bar' => 'baz', 'boo' => 'bam'];
$allowedKeys = ['foo', 'bar'];
$filteredParameters = array_intersect_key($parameters, array_flip($allowedKeys));

// $filteredParameters contains ['foo' => 'bar', 'bar' => 'baz]
```

If the `parameters` variable doesn't contain any allowed key, then the `filteredParameters` variable will consist of an empty array.

Since PHP 5.6 you can use `array_filter` for this task too, passing the `ARRAY_FILTER_USE_KEY` flag as the third parameter:

```
$parameters  = ['foo' => 1, 'hello' => 'world'];
$allowedKeys = ['foo', 'bar'];
$filteredParameters = array_filter(
    $parameters,
    function ($key) use ($allowedKeys) {
        return in_array($key, $allowedKeys);
    },
    ARRAY_FILTER_USE_KEY
);
```

Using `array_filter` gives the additional flexibility of performing an arbitrary test against the key, e.g. `$allowedKeys` could contain regex patterns instead of plain strings. It also more explicitly states the intention of the code than `array_intersect_key()` combined with `array_flip()`.

Section 15.5: Adding element to start of array

Sometimes you want to add an element to the beginning of an array **without modifying any of the current elements (*order*) within the array**. Whenever this is the case, you can use `array_unshift()`.

> **`array_unshift()`** prepends passed elements to the front of the array. Note that the list of elements is prepended as a whole, so that the prepended elements stay in the same order. All numerical array keys will be modified to start counting from zero while literal keys won't be touched.

Taken from the PHP documentation for `array_unshift()`.

If you'd like to achieve this, all you need to do is the following:

```
$myArray = array(1, 2, 3);

array_unshift($myArray, 4);
```

This will now add `4` as the first element in your array. You can verify this by:

```
print_r($myArray);
```

This returns an array in the following order: `4, 1, 2, 3`.

Since `array_unshift` forces the array to reset the key-value pairs as the new element let the following entries have the keys `n+1` it is smarter to create a new array and append the existing array to the newly created array.

Example:

```
$myArray = array('apples', 'bananas', 'pears');
$myElement = array('oranges');
$joinedArray = $myElement;

foreach ($myArray as $i) {
  $joinedArray[] = $i;
}
```

Output ($joinedArray):

```
Array ( [0] => oranges [1] => apples [2] => bananas [3] => pears )
```

Eaxmple/Demo

Section 15.6: Exchange values with keys

`array_flip` function will exchange all keys with its elements.

```
$colors = array(
    'one' => 'red',
    'two' => 'blue',
    'three' => 'yellow',
);

array_flip($colors); //will output

array(
    'red' => 'one',
    'blue' => 'two',
    'yellow' => 'three'
)
```

Section 15.7: Merge two arrays into one array

```
$a1 = array("red","green");
$a2 = array("blue","yellow");
print_r(array_merge($a1,$a2));

/*
    Array ( [0] => red [1] => green [2] => blue [3] => yellow )
*/
```

Associative array:

```
$a1=array("a"=>"red","b"=>"green");
$a2=array("c"=>"blue","b"=>"yellow");
print_r(array_merge($a1,$a2));
/*
    Array ( [a] => red [b] => yellow [c] => blue )
*/
```

1. Merges the elements of one or more arrays together so that the values of one are appended to the end of

the previous one. It returns the resulting array.
2. If the input arrays have the same string keys, then the later value for that key will overwrite the previous one. If, however, the arrays contain numeric keys, the later value will not overwrite the original value, but will be appended.
3. Values in the input array with numeric keys will be renumbered with incrementing keys starting from zero in the result array.

Chapter 16: Processing Multiple Arrays Together

Section 16.1: Array intersection

The `array_intersect` function will return an array of values that exist in all arrays that were passed to this function.

```
$array_one = ['one', 'two', 'three'];
$array_two = ['two', 'three', 'four'];
$array_three = ['two', 'three'];

$intersect = array_intersect($array_one, $array_two, $array_three);
// $intersect contains ['two', 'three']
```

Array keys are preserved. Indexes from the original arrays are not.

`array_intersect` only check the values of the arrays. `array_intersect_assoc` function will return intersection of arrays with keys.

```
$array_one = [1 => 'one',2 => 'two',3 => 'three'];
$array_two = [1 => 'one', 2 => 'two', 3 => 'two', 4 => 'three'];
$array_three = [1 => 'one', 2 => 'two'];

$intersect = array_intersect_assoc($array_one, $array_two, $array_three);
// $intersect contains [1 =>'one',2 => 'two']
```

`array_intersect_key` function only check the intersection of keys. It will returns keys exist in all arrays.

```
$array_one = [1 => 'one',2 => 'two',3 => 'three'];
$array_two = [1 => 'one', 2 => 'two', 3 => 'four'];
$array_three = [1 => 'one', 3 => 'five'];

$intersect = array_intersect_key($array_one, $array_two, $array_three);
// $intersect contains [1 =>'one',3 => 'three']
```

Section 16.2: Merge or concatenate arrays

```
$fruit1 = ['apples', 'pears'];
$fruit2 = ['bananas', 'oranges'];

$all_of_fruits = array_merge($fruit1, $fruit2);
// now value of $all_of_fruits is [0 => 'apples', 1 => 'pears', 2 => 'bananas', 3 => 'oranges']
```

Note that `array_merge` will change numeric indexes, but overwrite string indexes

```
$fruit1 = ['one' => 'apples',  'two' => 'pears'];
$fruit2 = ['one' => 'bananas', 'two' => 'oranges'];

$all_of_fruits = array_merge($fruit1, $fruit2);
// now value of $all_of_fruits is ['one' => 'bananas', 'two' => 'oranges']
```

`array_merge` overwrites the values of the first array with the values of the second array, if it cannot renumber the index.

You can use the `+` operator to merge two arrays in a way that the values of the first array never get overwritten, but

it does not renumber numeric indexes, so you lose values of arrays that have an index that is also used in the first array.

```
$fruit1 = ['one' => 'apples',  'two' => 'pears'];
$fruit2 = ['one' => 'bananas', 'two' => 'oranges'];

$all_of_fruits = $fruit1 + $fruit2;
// now value of $all_of_fruits is ['one' => 'apples', 'two' => 'pears']

$fruit1 = ['apples', 'pears'];
$fruit2 = ['bananas', 'oranges'];

$all_of_fruits = $fruit1 + $fruit2;
// now value of $all_of_fruits is [0 => 'apples', 1 => 'pears']
```

Section 16.3: Changing a multidimensional array to associative array

If you have a multidimensional array like this:

```
[
    ['foo',  'bar'],
    ['fizz', 'buzz'],
]
```

And you want to change it to an associative array like this:

```
[
    'foo'  => 'bar',
    'fizz' => 'buzz',
]
```

You can use this code:

```
$multidimensionalArray = [
    ['foo',  'bar'],
    ['fizz', 'buzz'],
];
$associativeArrayKeys   = array_column($multidimensionalArray, 0);
$associativeArrayValues = array_column($multidimensionalArray, 1);
$associativeArray       = array_combine($associativeArrayKeys, $associativeArrayValues);
```

Or, you can skip setting `$associativeArrayKeys` and `$associativeArrayValues` and use this simple one liner:

```
$associativeArray = array_combine(array_column($multidimensionalArray, 0),
array_column($multidimensionalArray, 1));
```

Section 16.4: Combining two arrays (keys from one, values from another)

The following example shows how to merge two arrays into one associative array, where the key values will be the items of the first array, and the values will be from the second:

```
$array_one = ['key1', 'key2', 'key3'];
$array_two = ['value1', 'value2', 'value3'];
```

```php
$array_three = array_combine($array_one, $array_two);
var_export($array_three);

/*
    array (
      'key1' => 'value1',
      'key2' => 'value2',
      'key3' => 'value3',
    )
*/
```

Chapter 17: Datetime Class

Section 17.1: Create Immutable version of DateTime from Mutable prior PHP 5.6

To create `\DateTimeImmutable` in PHP 5.6+ use:

```
\DateTimeImmutable::createFromMutable($concrete);
```

Prior PHP 5.6 you can use:

```
\DateTimeImmutable::createFromFormat(\DateTime::ISO8601, $mutable->format(\DateTime::ISO8601),
$mutable->getTimezone());
```

Section 17.2: Add or Subtract Date Intervals

We can use the class DateInterval to add or subtract some interval in a DateTime object.

See the example below, where we are adding an interval of 7 days and printing a message on the screen:

```
$now = new DateTime();// empty argument returns the current date
$interval = new DateInterval('P7D');//this objet represents a 7 days interval
$lastDay = $now->add($interval); //this will return a DateTime object
$formatedLastDay = $lastDay->format('Y-m-d');//this method format the DateTime object and returns a
String
echo "Samara says: Seven Days. You'll be happy on $formatedLastDay.";
```

This will output (running on Aug 1st, 2016):

> Samara says: Seven Days. You'll be happy on 2016-08-08.

We can use the sub method in a similar way to subtract dates

```
$now->sub($interval);
echo "Samara says: Seven Days. You were happy last on $formatedLastDay.";
```

This will output (running on Aug 1st, 2016):

> Samara says: Seven Days. You were happy last on 2016-07-25.

Section 17.3: getTimestamp

`getTimeStemp` is a unix representation of a datetime object.

```
$date = new DateTime();
echo $date->getTimestamp();
```

this will out put an integer indication the seconds that have elapsed since 00:00:00 UTC, Thursday, 1 January 1970.

Section 17.4: setDate

`setDate` sets the date in a DateTime object.

```
$date = new DateTime();
$date->setDate(2016, 7, 25);
```

this example sets the date to be the twenty-fifth of July, 2015, it will produce the following result:

```
2016-07-25 17:52:15.819442
```

Section 17.5: Create DateTime from custom format

PHP is able to parse a number of date formats. If you want to parse a non-standard format, or if you want your code to explicitly state the format to be used, then you can use the static `DateTime::createFromFormat` method:

Object oriented style

```
$format = "Y,m,d";
$time = "2009,2,26";
$date = DateTime::createFromFormat($format, $time);
```

Procedural style

```
$format = "Y,m,d";
$time = "2009,2,26";
$date = date_create_from_format($format, $time);
```

Section 17.6: Printing DateTimes

PHP 4+ supplies a method, format that converts a DateTime object into a string with a desired format. According to PHP Manual, this is the object oriented function:

```
public string DateTime::format ( string $format )
```

The function date() takes one parameters - a format, which is a string

Format

The format is a string, and uses single characters to define the format:

- **Y**: four digit representation of the year (eg: 2016)
- **y**: two digit representation of the year (eg: 16)
- **m**: month, as a number (01 to 12)
- **M**: month, as three letters (Jan, Feb, Mar, etc)
- **j**: day of the month, with no leading zeroes (1 to 31)
- **D**: day of the week, as three letters (Mon, Tue, Wed, etc)
- **h**: hour (12-hour format) (01 to 12)
- **H**: hour (24-hour format) (00 to 23)
- **A**: either AM or PM
- **i**: minute, with leading zeroes (00 to 59)
- **s**: second, with leading zeroes (00 to 59)
- The complete list can be found here

Usage

These characters can be used in various combinations to display times in virtually any format. Here are some examples:

```
$date = new DateTime('2000-05-26T13:30:20'); /* Friday, May 26, 2000 at 1:30:20 PM */

$date->format("H:i");
/* Returns 13:30 */

$date->format("H i s");
/* Returns 13 30 20 */

$date->format("h:i:s A");
/* Returns 01:30:20 PM */

$date->format("j/m/Y");
/* Returns 26/05/2000 */

$date->format("D, M j 'y - h:i A");
/* Returns Fri, May 26 '00 - 01:30 PM */
```

Procedural

The procedural format is similar:

Object-Oriented

```
$date->format($format)
```

Procedural Equivalent

```
date_format($date, $format)
```

Chapter 18: Working with Dates and Time

Section 18.1: Getting the difference between two dates / times

The most feasible way is to use, the `DateTime` class.

An example:

```
<?php
// Create a date time object, which has the value of ~ two years ago
$twoYearsAgo = new DateTime("2014-01-18 20:05:56");
// Create a date time object, which has the value of ~ now
$now = new DateTime("2016-07-21 02:55:07");

// Calculate the diff
$diff = $now->diff($twoYearsAgo);

// $diff->y contains the difference in years between the two dates
$yearsDiff = $diff->y;
// $diff->m contains the difference in minutes between the two dates
$monthsDiff = $diff->m;
// $diff->d contains the difference in days between the two dates
$daysDiff = $diff->d;
// $diff->h contains the difference in hours between the two dates
$hoursDiff = $diff->h;
// $diff->i contains the difference in minutes between the two dates
$minsDiff = $diff->i;
// $diff->s contains the difference in seconds between the two dates
$secondsDiff = $diff->s;

// Total Days Diff, that is the number of days between the two dates
$totalDaysDiff = $diff->days;

// Dump the diff altogether just to get some details ;)
var_dump($diff);
```

Also, comparing two dates is much easier, just use the Comparison operators , like:

```
<?php
// Create a date time object, which has the value of ~ two years ago
$twoYearsAgo = new DateTime("2014-01-18 20:05:56");
// Create a date time object, which has the value of ~ now
$now = new DateTime("2016-07-21 02:55:07");
var_dump($now > $twoYearsAgo); // prints bool(true)
var_dump($twoYearsAgo > $now); // prints bool(false)
var_dump($twoYearsAgo <= $twoYearsAgo); // prints bool(true)
var_dump($now == $now); // prints bool(true)
```

Section 18.2: Convert a date into another format

The Basics

The simplist way to convert one date format into another is to use `strtotime()` with `date()`. `strtotime()` will convert the date into a Unix Timestamp. That Unix Timestamp can then be passed to `date()` to convert it to the new format.

```
$timestamp = strtotime('2008-07-01T22:35:17.02');
```

```
$new_date_format = date('Y-m-d H:i:s', $timestamp);
```

Or as a one-liner:

```
$new_date_format = date('Y-m-d H:i:s', strtotime('2008-07-01T22:35:17.02'));
```

Keep in mind that `strtotime()` requires the date to be in a valid format. Failure to provide a valid format will result in `strtotime()` returning false which will cause your date to be 1969-12-31.

Using `DateTime()`

As of PHP 5.2, PHP offered the `DateTime()` class which offers us more powerful tools for working with dates (and time). We can rewrite the above code using `DateTime()` as so:

```
$date = new DateTime('2008-07-01T22:35:17.02');
$new_date_format = $date->format('Y-m-d H:i:s');
```

Working with Unix timestamps

`date()` takes a Unix timestamp as its second parameter and returns a formatted date for you:

```
$new_date_format = date('Y-m-d H:i:s', '1234567890');
```

DateTime() works with Unix timestamps by adding an `@` before the timestamp:

```
$date = new DateTime('@1234567890');
$new_date_format = $date->format('Y-m-d H:i:s');
```

If the timestamp you have is in milliseconds (it may end in `000` and/or the timestamp is thirteen characters long) you will need to convert it to seconds before you can can convert it to another format. There's two ways to do this:

- Trim the last three digits off using `substr()`

Trimming the last three digits can be achieved several ways, but using `substr()` is the easiest:

```
$timestamp = substr('1234567899000', -3);
```

- Divide the substr by 1000

You can also convert the timestamp into seconds by dividing by 1000. Because the timestamp is too large for 32 bit systems to do math on you will need to use the BCMath library to do the math as strings:

```
$timestamp = bcdiv('1234567899000', '1000');
```

To get a Unix Timestamp you can use `strtotime()` which returns a Unix Timestamp:

```
$timestamp = strtotime('1973-04-18');
```

With DateTime() you can use `DateTime::getTimestamp()`

```
$date = new DateTime('2008-07-01T22:35:17.02');
$timestamp = $date->getTimestamp();
```

If you're running PHP 5.2 you can use the `U` formatting option instead:

```php
$date = new DateTime('2008-07-01T22:35:17.02');
$timestamp = $date->format('U');
```

Working with non-standard and ambiguous date formats

Unfortunately not all dates that a developer has to work with are in a standard format. Fortunately PHP 5.3 provided us with a solution for that. `DateTime::createFromFormat()` allows us to tell PHP what format a date string is in so it can be successfully parsed into a DateTime object for further manipulation.

```php
$date = DateTime::createFromFormat('F-d-Y h:i A', 'April-18-1973 9:48 AM');
$new_date_format = $date->format('Y-m-d H:i:s');
```

In PHP 5.4 we gained the ability to do class member access on instantiation has been added which allows us to turn our `DateTime()` code into a one-liner:

```php
$new_date_format = (new DateTime('2008-07-01T22:35:17.02'))->format('Y-m-d H:i:s');
```

Unfortunately this does not work with `DateTime::createFromFormat()` yet.

Section 18.3: Parse English date descriptions into a Date format

Using the `strtotime()` function combined with `date()` you can parse different English text descriptions to dates:

```php
// Gets the current date
echo date("m/d/Y", strtotime("now")), "\n"; // prints the current date
echo date("m/d/Y", strtotime("10 September 2000")), "\n"; // prints September 10, 2000 in the m/d/Y
format
echo date("m/d/Y", strtotime("-1 day")), "\n"; // prints yesterday's date
echo date("m/d/Y", strtotime("+1 week")), "\n"; // prints the result of the current date + a week
echo date("m/d/Y", strtotime("+1 week 2 days 4 hours 2 seconds")), "\n"; // same as the last
example but with extra days, hours, and seconds added to it
echo date("m/d/Y", strtotime("next Thursday")), "\n"; // prints next Thursday's date
echo date("m/d/Y", strtotime("last Monday")), "\n"; // prints last Monday's date
echo date("m/d/Y", strtotime("First day of next month")), "\n"; // prints date of first day of next
month
echo date("m/d/Y", strtotime("Last day of next month")), "\n"; // prints date of last day of next
month
echo date("m/d/Y", strtotime("First day of last month")), "\n"; // prints date of first day of last
month
echo date("m/d/Y", strtotime("Last day of last month")), "\n"; // prints date of last day of last
month
```

Section 18.4: Using Predefined Constants for Date Format

We can use Predefined Constants for Date format in `date()` instead of the conventional date format strings since PHP 5.1.0.

Predefined Date Format Constants Available

`DATE_ATOM` - Atom (2016-07-22T14:50:01+00:00)

`DATE_COOKIE` - HTTP Cookies (Friday, 22-Jul-16 14:50:01 UTC)

`DATE_RSS` - RSS (Fri, 22 Jul 2016 14:50:01 +0000)

`DATE_W3C` - World Wide Web Consortium (2016-07-22T14:50:01+00:00)

`DATE_ISO8601` - ISO-8601 (2016-07-22T14:50:01+0000)

`DATE_RFC822` - RFC 822 (Fri, 22 Jul 16 14:50:01 +0000)

`DATE_RFC850` - RFC 850 (Friday, 22-Jul-16 14:50:01 UTC)

`DATE_RFC1036` - RFC 1036 (Fri, 22 Jul 16 14:50:01 +0000)

`DATE_RFC1123` - RFC 1123 (Fri, 22 Jul 2016 14:50:01 +0000)

`DATE_RFC2822` - RFC 2822 (Fri, 22 Jul 2016 14:50:01 +0000)

`DATE_RFC3339` - Same as DATE_ATOM (2016-07-22T14:50:01+00:00)

Usage Examples

```
echo date(DATE_RFC822);
```

> This will output: **Fri, 22 Jul 16 14:50:01 +0000**

```
echo date(DATE_ATOM,mktime(0,0,0,8,15,1947));
```

> This will output: **1947-08-15T00:00:00+05:30**

Chapter 19: Control Structures

Section 19.1: if else

The `if` statement in the example above allows to execute a code fragment, when the condition is met. When you want to execute a code fragment, when the condition is not met you extend the `if` with an `else`.

```
if ($a > $b) {
  echo "a is greater than b";
} else {
  echo "a is NOT greater than b";
}
```

PHP Manual - Control Structures - Else

The ternary operator as shorthand syntax for if-else

The ternary operator evaluates something based on a condition being true or not. It is a comparison operator and often used to express a simple if-else condition in a shorter form. It allows to quickly test a condition and often replaces a multi-line if statement, making your code more compact.

This is the example from above using a ternary expression and variable values: `$a=1; $b=2;`

```
echo ($a > $b) ? "a is greater than b" : "a is NOT greater than b";
```

Outputs: `a is NOT greater than b`.

Section 19.2: Alternative syntax for control structures

PHP provides an alternative syntax for some control structures: `if`, `while`, `for`, `foreach`, and `switch`.

When compared to the normal syntax, the difference is, that the opening brace is replaced by a colon (`:`) and the closing brace is replaced by `endif;`, `endwhile;`, `endfor;`, `endforeach;`, or `endswitch;`, respectively. For individual examples, see the topic on alternative syntax for control structures.

```
if ($a == 42):
    echo "The answer to life, the universe and everything is 42.";
endif;
```

Multiple `elseif` statements using short-syntax:

```
if ($a == 5):
    echo "a equals 5";
elseif ($a == 6):
    echo "a equals 6";
else:
    echo "a is neither 5 nor 6";
endif;
```

PHP Manual - Control Structures - Alternative Syntax

Section 19.3: while

`while` loop iterates through a block of code as long as a specified condition is true.

```php
$i = 1;
while ($i < 10) {
    echo $i;
    $i++;
}
```

Output:

123456789

For detailed information, see the Loops topic.

Section 19.4: do-while

`do-while` loop first executes a block of code once, in every case, then iterates through that block of code as long as a specified condition is true.

```php
$i = 0;
do {
    $i++;
    echo $i;
} while ($i < 10);

Output: `12345678910`
```

For detailed information, see the Loops topic.

Section 19.5: goto

The `goto` operator allows to jump to another section in the program. It's available since PHP 5.3.

The goto instruction is a goto followed by the desired target label: `goto MyLabel;`.

The target of the jump is specified by a label followed by a colon: `MyLabel:`.

This example will print `Hello World!`:

```php
<?php
goto MyLabel;
echo 'This text will be skipped, because of the jump.';

MyLabel:
echo 'Hello World!';
?>
```

Section 19.6: declare

`declare` is used to set an execution directive for a block of code.

The following directives are recognized:

- `ticks`
- `encoding`
- `strict_types`

For instance, set ticks to 1:

```
declare(ticks=1);
```

To enable strict type mode, the `declare` statement is used with the `strict_types` declaration:

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

Section 19.7: include & require

require

`require` is similar to `include`, except that it will produce a fatal `E_COMPILE_ERROR` level error on failure. When the `require` fails, it will halt the script. When the `include` fails, it will not halt the script and only emit `E_WARNING`.

```
require 'file.php';
```

PHP Manual - Control Structures - Require

include

The `include` statement includes and evaluates a file.

> *./variables.php*

```
$a = 'Hello World!';
```

> ./main.php`

```
include 'variables.php';
echo $a;
// Output: `Hello World!`
```

Be careful with this approach, since it is considered a code smell, because the included file is altering amount and content of the defined variables in the given scope.

You can also `include` file, which returns a value. This is extremely useful for handling configuration arrays:

> configuration.php

```
<?php
return [
    'dbname' => 'my db',
    'user' => 'admin',
    'pass' => 'password',
];
```

> main.php

```
<?php
```

```php
$config = include 'configuration.php';
```

This approach will prevent the included file from polluting your current scope with changed or added variables.

PHP Manual - Control Structures - Include

include & require can also be used to assign values to a variable when returned something by file.

Example :

include1.php file :

```php
<?php
    $a = "This is to be returned";

    return $a;
?>
```

index.php file :

```php
    $value = include 'include1.php';
   // Here, $value = "This is to be returned"
```

Section 19.8: return

The `return` statement returns the program control to the calling function.

When `return` is called from within a function, the execution of the current function will end.

```php
function returnEndsFunctions()
{
   echo 'This is executed';
   return;
   echo 'This is not executed.';
}
```

When you run `returnEndsFunctions();` you'll get the output `This is executed`;

When `return` is called from within a function with and argument, the execution of the current function will end and the value of the argument will be returned to the calling function.

Section 19.9: for

`for` loops are typically used when you have a piece of code which you want to repeat a given number of times.

```php
for ($i = 1; $i < 10; $i++) {
    echo $i;
}
```

Outputs:

123456789

For detailed information, see the Loops topic.

Section 19.10: foreach

`foreach` is a construct, which enables you to iterate over arrays and objects easily.

```
$array = [1, 2, 3];
foreach ($array as $value) {
    echo $value;
}
```

Outputs:

123

.

To use `foreach` loop with an object, it has to implement `Iterator` interface.

When you iterate over associative arrays:

```
$array = ['color'=>'red'];

foreach($array as $key => $value){
    echo $key . ': ' . $value;
}
```

Outputs:

```
color: red
```

For detailed information, see the Loops topic.

Section 19.11: if elseif else

elseif

`elseif` combines `if` and `else`. The `if` statement is extended to execute a different statement in case the original `if` expression is not met. But, the alternative expression is only executed, when the `elseif` conditional expression is met.

The following code displays either "a is bigger than b", "a is equal to b" or "a is smaller than b":

```
if ($a > $b) {
    echo "a is bigger than b";
} elseif ($a == $b) {
    echo "a is equal to b";
} else {
    echo "a is smaller than b";
}
```

Several elseif statements

You can use multiple elseif statements within the same if statement:

```
if ($a == 1) {
    echo "a is One";
} elseif ($a == 2) {
    echo "a is Two";
```

```
} elseif ($a == 3) {
    echo "a is Three";
} else {
    echo "a is not One, not Two nor Three";
}
```

Section 19.12: if

The if construct allows for conditional execution of code fragments.

```
if ($a > $b) {
  echo "a is bigger than b";
}
```

PHP Manual - Control Structures - If

Section 19.13: switch

The `switch` structure performs the same function as a series of `if` statements, but can do the job in fewer lines of code. The value to be tested, as defined in the `switch` statement, is compared for equality with the values in each of the `case` statements until a match is found and the code in that block is executed. If no matching `case` statement is found, the code in the `default` block is executed, if it exists.

Each block of code in a `case` or `default` statement should end with the `break` statement. This stops the execution of the `switch` structure and continues code execution immediately afterwards. If the `break` statement is omitted, the next `case` statement's code is executed, *even if there is no match*. This can cause unexpected code execution if the `break` statement is forgotten, but can also be useful where multiple `case` statements need to share the same code.

```
switch ($colour) {
case "red":
    echo "the colour is red";
    break;
case "green":
case "blue":
    echo "the colour is green or blue";
    break;
case "yellow":
    echo "the colour is yellow";
    // note missing break, the next block will also be executed
case "black":
    echo "the colour is black";
    break;
default:
    echo "the colour is something else";
    break;
}
```

In addition to testing fixed values, the construct can also be coerced to test dynamic statements by providing a boolean value to the `switch` statement and any expression to the `case` statement. Keep in mind the *first* matching value is used, so the following code will output "more than 100":

```
$i = 1048;
switch (true) {
case ($i > 0):
    echo "more than 0";
    break;
```

```
case ($i > 100):
    echo "more than 100";
    break;
case ($i > 1000):
    echo "more than 1000";
    break;
}
```

For possible issues with loose typing while using the `switch` construct, see Switch Surprises

Chapter 20: Loops

Loops are a fundamental aspect of programming. They allow programmers to create code that repeats for some given number of repetitions, or *iterations*. The number of iterations can be explicit (6 iterations, for example), or continue until some condition is met ('until Hell freezes over').

This topic covers the different types of loops, their associated control statements, and their potential applications in PHP.

Section 20.1: continue

> The `continue` keyword halts the current iteration of a loop but does not terminate the loop.

Just like the `break` statement the `continue` statement is situated inside the loop body. When executed, the `continue` statement causes execution to immediately jump to the loop conditional.

In the following example loop prints out a message based on the values in an array, but skips a specified value.

```
$list = ['apple', 'banana', 'cherry'];

foreach ($list as $value) {
    if ($value == 'banana') {
        continue;
    }
    echo "I love to eat {$value} pie.".PHP_EOL;
}
```

The expected output is:

```
I love to eat apple pie.
I love to eat cherry pie.
```

The `continue` statement may also be used to immediately continue execution to an outer level of a loop by specifying the number of loop levels to jump. For example, consider data such as

Fruit	Color	Cost
Apple	Red	1
Banana	Yellow	7
Cherry	Red	2
Grape	Green	4

In order to only make pies from fruit which cost less than 5

```
$data = [
    [ "Fruit" => "Apple",  "Color" => "Red",    "Cost" => 1 ],
    [ "Fruit" => "Banana", "Color" => "Yellow", "Cost" => 7 ],
    [ "Fruit" => "Cherry", "Color" => "Red",    "Cost" => 2 ],
    [ "Fruit" => "Grape",  "Color" => "Green",  "Cost" => 4 ]
];

foreach($data as $fruit) {
    foreach($fruit as $key => $value) {
        if ($key == "Cost" && $value >= 5) {
```

```
            continue 2;
        }
        /* make a pie */
    }
}
```

When the `continue 2` statement is executed, execution immediately jumps back to `$data as $fruit` continuing the outer loop and skipping all other code (including the conditional in the inner loop.

Section 20.2: break

> The `break` keyword immediately terminates the current loop.

Similar to the `continue` statement, a `break` halts execution of a loop. Unlike a `continue` statement, however, `break` causes the immediate termination of the loop and does *not* execute the conditional statement again.

```
$i = 5;
while(true) {
    echo 120/$i.PHP_EOL;
    $i -= 1;
    if ($i == 0) {
        break;
    }
}
```

This code will produce

```
24
30
40
60
120
```

but will not execute the case where `$i` is 0, which would result in a fatal error due to division by 0.

The break statement may also be used to break out of several levels of loops. Such behavior is very useful when executing nested loops. For example, to copy an array of strings into an output string, removing any # symbols, until the output string is exactly 160 characters

```
$output = "";
$inputs = array(
    "#soblessed #throwbackthursday",
    "happy tuesday",
    "#nofilter",
    /* more inputs */
);
foreach($inputs as $input) {
    for($i = 0; $i < strlen($input); $i += 1) {
        if ($input[$i] == '#') continue;
        $output .= $input[$i];
        if (strlen($output) == 160) break 2;
    }
    $output .= ' ';
}
```

The `break 2` command immediately terminates execution of both the inner and outer loops.

Section 20.3: foreach

> The `foreach` statement is used to loop through arrays.

For each iteration the value of the current array element is assigned to `$value` variable and the array pointer is moved by one and in the next iteration next element will be processed.

The following example displays the items in the array assigned.

```
$list = ['apple', 'banana', 'cherry'];

foreach ($list as $value) {
    echo "I love to eat {$value}. ";
}
```

The expected output is:

```
I love to eat apple. I love to eat banana. I love to eat cherry.
```

You can also access the key / index of a value using foreach:

```
foreach ($list as $key => $value) {
    echo $key . ":" . $value . " ";
}

//Outputs - 0:apple 1:banana 2:cherry
```

By default `$value` is a copy of the value in `$list`, so changes made inside the loop will not be reflected in `$list` afterwards.

```
foreach ($list as $value) {
    $value = $value . " pie";
}
echo $list[0]; // Outputs "apple"
```

To modify the array within the `foreach` loop, use the `&` operator to assign `$value` by reference. It's important to `unset` the variable afterwards so that reusing `$value` elsewhere doesn't overwrite the array.

```
foreach ($list as &$value) { // Or foreach ($list as $key => &$value) {
    $value = $value . " pie";
}
unset($value);
echo $list[0]; // Outputs "apple pie"
```

You can also modify the array items within the `foreach` loop by referencing the array key of the current item.

```
foreach ($list as $key => $value) {
    $list[$key] = $value . " pie";
}
echo $list[0]; // Outputs "apple pie"
```

Section 20.4: do...while

> The `do...while` statement will execute a block of code at least once - it then will repeat the loop as long as a condition is true.

The following example will increment the value of `$i` at least once, and it will continue incrementing the variable `$i` as long as it has a value of less than 25;

```
$i = 0;
do {
    $i++;
} while($i < 25);

echo 'The final value of i is: ', $i;
```

The expected output is:

```
The final value of i is: 25
```

Section 20.5: for

> The `for` statement is used when you know how many times you want to execute a statement or a block of statements.

The initializer is used to set the start value for the counter of the number of loop iterations. A variable may be declared here for this purpose and it is traditional to name it `$i`.

The following example iterates 10 times and displays numbers from 0 to 9.

```
for ($i = 0; $i <= 9; $i++) {
    echo $i, ',';
}

# Example 2
for ($i = 0; ; $i++) {
  if ($i > 9) {
      break;
  }
  echo $i, ',';
}

# Example 3
$i = 0;
for (; ; ) {
    if ($i > 9) {
        break;
    }
    echo $i, ',';
    $i++;
}

# Example 4
for ($i = 0, $j = 0; $i <= 9; $j += $i, print $i. ',', $i++);
```

The expected output is:

```
0,1,2,3,4,5,6,7,8,9,
```

Section 20.6: while

> The `while` statement will execute a block of code if and as long as a test expression is true.

If the test expression is true then the code block will be executed. After the code has executed the test expression will again be evaluated and the loop will continue until the test expression is found to be false.

The following example iterates till the sum reaches 100 before terminating.

```
$i = true;
$sum = 0;

while ($i) {
    if ($sum === 100) {
        $i = false;
    } else {
        $sum += 10;
    }
}
echo 'The sum is: ', $sum;
```

The expected output is:

```
The sum is: 100
```

Chapter 21: Functions

Section 21.1: Variable-length argument lists

Version ≥ 5.6

PHP 5.6 introduced variable-length argument lists (a.k.a. varargs, variadic arguments), using the `...` token before the argument name to indicate that the parameter is variadic, i.e. it is an array including all supplied parameters from that one onward.

```
function variadic_func($nonVariadic, ...$variadic) {
    echo json_encode($variadic);
}

variadic_func(1, 2, 3, 4); // prints [2,3,4]
```

Type names can be added in front of the `...`:

```
function foo(Bar ...$bars) {}
```

The `&` reference operator can be added before the `...`, but after the type name (if any). Consider this example:

```
class Foo{}
function a(Foo &...$foos){
    $i = 0;
    foreach($a as &$foo){ // note the &
        $foo = $i++;
    }
}
$a = new Foo;
$c = new Foo;
$b =& $c;
a($a, $b);
var_dump($a, $b, $c);
```

Output:

```
int(0)
int(1)
int(1)
```

On the other hand, an array (or `Traversable`) of arguments can be unpacked to be passed to a function in the form of an argument list:

```
var_dump(...hash_algos());
```

Output:

```
string(3) "md2"
string(3) "md4"
string(3) "md5"
...
```

Compare with this snippet without using `...`:

```
var_dump(hash_algos());
```

Output:

```
array(46) {
  [0]=>
  string(3) "md2"
  [1]=>
  string(3) "md4"
  ...
}
```

Therefore, redirect functions for variadic functions can now be easily made, for example:

```
public function formatQuery($query, ...$args){
    return sprintf($query, ...array_map([$mysqli, "real_escape_string"], $args));
}
```

Apart from arrays, `Traversables`, such as `Iterator` (especially many of its subclasses from SPL) can also be used. For example:

```
$iterator = new LimitIterator(new ArrayIterator([0, 1, 2, 3, 4, 5, 6]), 2, 3);
echo bin2hex(pack("c*", ...$it)); // Output: 020304
```

If the iterator iterates infinitely, for example:

```
$iterator = new InfiniteIterator(new ArrayIterator([0, 1, 2, 3, 4]));
var_dump(...$iterator);
```

Different versions of PHP behave differently:

- From PHP 7.0.0 up to PHP 7.1.0 (beta 1):
  - A segmentation fault will occur
  - The PHP process will exit with code 139
- In PHP 5.6:
  - A fatal error of memory exhaustion ("Allowed memory size of %d bytes exhausted") will be shown.
  - The PHP process will exit with code 255

> Note: HHVM (v3.10 - v3.12) does not support unpacking `Traversables`. A warning message "Only containers may be unpacked" will be shown in this attempt.

Section 21.2: Optional Parameters

Functions can have optional parameters, for example:

```
function hello($name, $style = 'Formal')
{
    switch ($style) {
        case 'Formal':
            print "Good Day $name";
            break;
        case 'Informal':
            print "Hi $name";
```

```
            break;
        case 'Australian':
            print "G'day $name";
            break;
        default:
            print "Hello $name";
            break;
    }
}

hello('Alice');
    // Good Day Alice

hello('Alice', 'Australian');
    // G'day Alice
```

Section 21.3: Passing Arguments by Reference

Function arguments can be passed "By Reference", allowing the function to modify the variable used outside the function:

```
function pluralize(&$word)
{
    if (substr($word, -1) == 'y') {
        $word = substr($word, 0, -1) . 'ies';
    } else {
      $word .= 's';
    }
}

$word = 'Bannana';
pluralize($word);

print $word;
  // Bannanas
```

Object arguments are always passed by reference:

```
function addOneDay($date)
{
    $date->modify('+1 day');
}

$date = new DateTime('2014-02-28');
addOneDay($date);

print $date->format('Y-m-d');
  // 2014-03-01
```

To avoid implicit passing an object by reference, you should `clone` the object.

Passing by reference can also be used as an alternative way to return parameters. For example, the `socket_getpeername` function:

```
bool socket_getpeername ( resource $socket , string &$address [, int &$port ] )
```

This method actually aims to return the address and port of the peer, but since there are two values to return, it chooses to use reference parameters instead. It can be called like this:

```php
if(!socket_getpeername($socket, $address, $port)) {
    throw new RuntimeException(socket_last_error());
}
echo "Peer: $address:$port\n";
```

The variables `$address` and `$port` do not need to be defined before. They will:

1. be defined as `null` first,
2. then passed to the function with the predefined `null` value
3. then modified in the function
4. end up defined as the address and port in the calling context.

Section 21.4: Basic Function Usage

A basic function is defined and executed like this:

```php
function hello($name)
{
    print "Hello $name";
}

hello("Alice");
```

Section 21.5: Function Scope

Variables inside functions is inside a local scope like this

```php
$number = 5
function foo(){
    $number = 10
    return $number
}

foo(); //Will print 10 because text defined inside function is a local variable
```

Chapter 22: Functional Programming

PHP's functional programming relies on functions. Functions in PHP provide organized, reusable code to perform a set of actions. Functions simplify the coding process, prevent redundant logic, and make code easier to follow. This topic describes the declaration and utilization of functions, arguments, parameters, return statements and scope in PHP.

Section 22.1: Closures

A closure is an anonymous function that can't access outside scope.

When defining an anonymous function as such, you're creating a "namespace" for that function. It currently only has access to that namespace.

```
$externalVariable = "Hello";
$secondExternalVariable = "Foo";
$myFunction = function() {

  var_dump($externalVariable, $secondExternalVariable); // returns two error notice, since the
variables aren't defined

}
```

It doesn't have access to any external variables. To grant this permission for this namespace to access external variables, you need to introduce it via closures (***use***`()`).

```
$myFunction = function() use($externalVariable, $secondExternalVariable) {
   var_dump($externalVariable, $secondExternalVariable); // Hello Foo
}
```

This is heavily attributed to PHP's tight variable scoping - *If a variable isn't defined within the scope, or isn't brought in with* ***`global`*** *then it does not exist.*

Also note:

> Inheriting variables from the parent scope is not the same as using global variables. Global variables exist in the global scope, which is the same no matter what function is executing.

> The parent scope of a closure is the function in which the closure was declared (not necessarily the function it was called from).

Taken from the PHP Documentation for Anonymous Functions

In PHP, closures use an **early-binding** approach. This means that variables passed to the closure's namespace using **`use`** keyword will have the same values when the closure was defined.

To change this behavior you should pass the variable **by-reference**.

```
$rate = .05;

// Exports variable to closure's scope
$calculateTax = function ($value) use ($rate) {
```

```
    return $value * $rate;
};

$rate = .1;

print $calculateTax(100); // 5
```

```
$rate = .05;

// Exports variable to closure's scope
$calculateTax = function ($value) use (&$rate) { // notice the & before $rate
    return $value * $rate;
};

$rate = .1;

print $calculateTax(100); // 10
```

Default arguments are not implicitly required when defining anonymous functions with/without closures.

```
$message = 'Im yelling at you';

$yell = function() use($message) {
    echo strtoupper($message);
};

$yell(); // returns: IM YELLING AT YOU
```

Section 22.2: Assignment to variables

Anonymous functions can be assigned to variables for use as parameters where a callback is expected:

```
$uppercase = function($data) {
    return strtoupper($data);
};

$mixedCase = ["Hello", "World"];
$uppercased = array_map($uppercase, $mixedCase);
print_r($uppercased);
```

These variables can also be used as standalone function calls:

```
echo $uppercase("Hello world!"); // HELLO WORLD!
```

Section 22.3: Objects as a function

```
class SomeClass {
    public function __invoke($param1, $param2) {
        // put your code here
    }
}

$instance = new SomeClass();
$instance('First', 'Second'); // call the __invoke() method
```

An object with an `__invoke` method can be used exactly as any other function.

The `__invoke` method will have access to all properties of the object and will be able to call any methods.

Section 22.4: Using outside variables

The `use` construct is used to import variables into the anonymous function's scope:

```
$divisor = 2332;
$myfunction = function($number) use ($divisor) {
    return $number / $divisor;
};

echo $myfunction(81620); //Outputs 35
```

Variables can also be imported by reference:

```
$collection = [];

$additem = function($item) use (&$collection) {
    $collection[] = $item;
};

$additem(1);
$additem(2);

//$collection is now [1,2]
```

Section 22.5: Anonymous function

An anonymous function is just a **function** that doesn't have a name.

```
// Anonymous function
function() {
    return "Hello World!";
};
```

In PHP, an anonymous function is treated like an **expression** and for this reason, it should be ended with a semicolon `;`.

An anonymous function should be **assigned** to a variable.

```
// Anonymous function assigned to a variable
$sayHello = function($name) {
    return "Hello $name!";
};

print $sayHello('John'); // Hello John
```

Or it should be **passed as parameter** of another function.

```
$users = [
    ['name' => 'Alice', 'age' => 20],
    ['name' => 'Bobby', 'age' => 22],
    ['name' => 'Carol', 'age' => 17]
];

// Map function applying anonymous function
$userName = array_map(function($user) {
    return $user['name'];
}, $users);
```

```
print_r($usersName); // ['Alice', 'Bobby', 'Carol']
```

Or even been **returned** from another function.

Self-executing anonymous functions:

```
// For PHP 7.x
(function () {
    echo "Hello world!";
})();

// For PHP 5.x
call_user_func(function () {
    echo "Hello world!";
});
```

Passing an argument into self-executing anonymous functions:

```
// For PHP 7.x
(function ($name) {
    echo "Hello $name!";
})('John');

// For PHP 5.x
call_user_func(function ($name) {
    echo "Hello $name!";
}, 'John');
```

Section 22.6: Pure functions

A **pure function** is a function that, given the same input, will always return the same output and are **side-effect** free.

```
// This is a pure function
function add($a, $b) {
    return $a + $b;
}
```

Some **side-effects** are *changing the filesystem*, *interacting with databases*, *printing to the screen*.

```
// This is an impure function
function add($a, $b) {
    echo "Adding...";
    return $a + $b;
}
```

Section 22.7: Common functional methods in PHP

Mapping

Applying a function to all elements of an array:

```
array_map('strtoupper', $array);
```

Be aware that this is the only method of the list where the callback comes first.

Reducing (or folding)

Reducing an array to a single value:

```
$sum = array_reduce($numbers, function ($carry, $number) {
   return $carry + $number;
});
```

Filtering

Returns only the array items for which the callback returns `true`:

```
$onlyEven = array_filter($numbers, function ($number) {
    return ($number % 2) === 0;
});
```

Section 22.8: Using built-in functions as callbacks

In functions taking `callable` as an argument, you can also put a string with PHP built-in function. It's common to use `trim` as `array_map` parameter to remove leading and trailing whitespace from all strings in the array.

```
$arr = ['   one  ', 'two   ', '   three'];
var_dump(array_map('trim', $arr));

// array(3) {
//   [0] =>
//   string(3) "one"
//   [1] =>
//   string(3) "two"
//   [2] =>
//   string(5) "three"
// }
```

Section 22.9: Scope

In PHP, an anonymous function has its own **scope** like any other PHP function.

In JavaScript, an anonymous function can access a variable in outside scope. But in PHP, this is not permitted.

```
$name = 'John';

// Anonymous function trying access outside scope
$sayHello = function() {
    return "Hello $name!";
}

print $sayHello('John'); // Hello !
// With notices active, there is also an Undefined variable $name notice
```

Section 22.10: Passing a callback function as a parameter

There are several PHP functions that accept user-defined callback functions as a parameter, such as: `call_user_func()`, `usort()` and `array_map()`.

Depending on where the user-defined callback function was defined there are different ways to pass them:

Procedural style:

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

$initial_array = [1, 2, 3, 4, 5];
$final_array = array_map('square', $initial_array);
var_dump($final_array); // prints the new array with 1, 4, 9, 16, 25
```

Object Oriented style:

```php
class SquareHolder
{
    function square($number)
    {
        return $number * $number;
    }
}

$squaredHolder = new SquareHolder();
$initial_array = [1, 2, 3, 4, 5];
$final_array = array_map([$squaredHolder, 'square'], $initial_array);

var_dump($final_array); // prints the new array with 1, 4, 9, 16, 25
```

Object Oriented style using a static method:

```php
class StaticSquareHolder
{
    public static function square($number)
    {
        return $number * $number;
    }
}

$initial_array = [1, 2, 3, 4, 5];
$final_array = array_map(['StaticSquareHolder', 'square'], $initial_array);
// or:
$final_array = array_map('StaticSquareHolder::square', $initial_array); // for PHP >= 5.2.3

var_dump($final_array); // prints the new array with 1, 4, 9, 16, 25
```

Chapter 23: Alternative Syntax for Control Structures

Section 23.1: Alternative if/else statement

```
<?php

if ($condition):
    do_something();
elseif ($another_condition):
    do_something_else();
else:
    do_something_different();
endif;

?>

<?php if ($condition): ?>
    <p>Do something in HTML</p>
<?php elseif ($another_condition): ?>
    <p>Do something else in HTML</p>
<?php else: ?>
    <p>Do something different in HTML</p>
<?php endif; ?>
```

Section 23.2: Alternative for statement

```
<?php

for ($i = 0; $i < 10; $i++):
    do_something($i);
endfor;

?>

<?php for ($i = 0; $i < 10; $i++): ?>
    <p>Do something in HTML with <?php echo $i; ?></p>
<?php endfor; ?>
```

Section 23.3: Alternative while statement

```
<?php

while ($condition):
    do_something();
endwhile;

?>

<?php while ($condition): ?>
    <p>Do something in HTML</p>
<?php endwhile; ?>
```

Section 23.4: Alternative foreach statement

```
<?php
```

```php
foreach ($collection as $item):
    do_something($item);
endforeach;

?>

<?php foreach ($collection as $item): ?>
    <p>Do something in HTML with <?php echo $item; ?></p>
<?php endforeach; ?>
```

Section 23.5: Alternative switch statement

```php
<?php

switch ($condition):
    case $value:
        do_something();
        break;
    default:
        do_something_else();
        break;
endswitch;

?>

<?php switch ($condition): ?>
<?php case $value: /* having whitespace before your cases will cause an error */ ?>
    <p>Do something in HTML</p>
    <?php break; ?>
<?php default: ?>
    <p>Do something else in HTML</p>
    <?php break; ?>
<?php endswitch; ?>
```

Chapter 24: String formatting

Section 24.1: String interpolation

You can also use interpolation to interpolate (*insert*) a variable within a string. Interpolation works in double quoted strings and the heredoc syntax only.

```
$name = 'Joel';

// $name will be replaced with `Joel`
echo "<p>Hello $name, Nice to see you.</p>";
#                 ↕
#>   "<p>Hello Joel, Nice to see you.</p>"

// Single Quotes: outputs $name as the raw text (without interpreting it)
echo 'Hello $name, Nice to see you.'; # Careful with this notation
#> "Hello $name, Nice to see you."
```

The **complex (curly) syntax** format provides another option which requires that you wrap your variable within curly braces {}. This can be useful when embedding variables within textual content and helping to prevent possible ambiguity between textual content and variables.

```
$name = 'Joel';

// Example using the curly brace syntax for the variable $name
echo "<p>We need more {$name}s to help us!</p>";
#> "<p>We need more Joels to help us!</p>"

// This line will throw an error (as `$names` is not defined)
echo "<p>We need more $names to help us!</p>";
#> "Notice: Undefined variable: names"
```

The {} syntax only interpolates variables starting with a $ into a string. The {} syntax **does not** evaluate arbitrary PHP expressions.

```
// Example tying to interpolate a PHP expression
echo "1 + 2 = {1 + 2}";
#> "1 + 2 = {1 + 2}"

// Example using a constant
define("HELLO_WORLD", "Hello World!!");
echo "My constant is {HELLO_WORLD}";
#> "My constant is {HELLO_WORLD}"

// Example using a function
function say_hello() {
    return "Hello!";
};
echo "I say: {say_hello()}";
#> "I say: {say_hello()}"
```

However, the {} syntax does evaluate any array access, property access and function/method calls on variables, array elements or properties:

```
// Example accessing a value from an array — multidimensional access is allowed
$companions = [0 => ['name' => 'Amy Pond'], 1 => ['name' => 'Dave Random']];
echo "The best companion is: {$companions[0]['name']}";
```

```
#> "The best companion is: Amy Pond"

// Example of calling a method on an instantiated object
class Person {
  function say_hello() {
    return "Hello!";
  }
}

$max = new Person();

echo "Max says: {$max->say_hello()}";
#> "Max says: Hello!"

// Example of invoking a Closure — the parameter list allows for custom expressions
$greet = function($num) {
    return "A $num greetings!";
};
echo "From us all: {$greet(10 ** 3)}";
#> "From us all: A 1000 greetings!"
```

Notice that the dollar $ sign can appear after the opening curly brace { as the above examples, or, like in Perl or Shell Script, can appear before it:

```
$name = 'Joel';

// Example using the curly brace syntax with dollar sign before the opening curly brace
echo "<p>We need more ${name}s to help us!</p>";
#> "<p>We need more Joels to help us!</p>"
```

> The `Complex (curly) syntax` is not called as such because it's complex, but rather because it allows for the use of '**complex expressions**'. Read more about `Complex (curly) syntax`

Section 24.2: Extracting/replacing substrings

Single characters can be extracted using array (square brace) syntax as well as curly brace syntax. These two syntaxes will only return a single character from the string. If more than one character is needed, a function will be required, i.e.- substr

Strings, like everything in PHP, are `0`-indexed.

```
$foo = 'Hello world';

$foo[6]; // returns 'w'
$foo{6}; // also returns 'w'

substr($foo, 6, 1); // also returns 'w'
substr($foo, 6, 2); // returns 'wo'
```

Strings can also be changed one character at a time using the same square brace and curly brace syntax. Replacing more than one character requires a function, i.e.- substr_replace

```
$foo = 'Hello world';

$foo[6] = 'W'; // results in $foo = 'Hello World'
$foo{6} = 'W'; // also results in $foo = 'Hello World'
```

```php
substr_replace($foo, 'W', 6, 1); // also results in $foo = 'Hello World'
substr_replace($foo, 'Whi', 6, 2); // results in 'Hello Whirled'
// note that the replacement string need not be the same length as the substring replaced
```

Chapter 25: String Parsing

Section 25.1: Splitting a string by separators

`explode` and `strstr` are simpler methods to get substrings by separators.

A string containing several parts of text that are separated by a common character can be split into parts with the `explode` function.

```
$fruits = "apple,pear,grapefruit,cherry";
print_r(explode(",",$fruits)); // ['apple', 'pear', 'grapefruit', 'cherry']
```

The method also supports a limit parameter that can be used as follow:

```
$fruits= 'apple,pear,grapefruit,cherry';
```

If the limit parameter is zero, then this is treated as 1.

```
print_r(explode(',',$fruits,0)); // ['apple,pear,grapefruit,cherry']
```

If limit is set and positive, the returned array will contain a maximum of limit elements with the last element containing the rest of string.

```
print_r(explode(',',$fruits,2)); // ['apple', 'pear,grapefruit,cherry']
```

If the limit parameter is negative, all components except the last -limit are returned.

```
print_r(explode(',',$fruits,-1)); // ['apple', 'pear', 'grapefruit']
```

`explode` can be combined with `list` to parse a string into variables in one line:

```
$email = "user@example.com";
list($name, $domain) = explode("@", $email);
```

However, make sure that the result of `explode` contains enough elements, or an undefined index warning would be triggered.

`strstr` strips away or only returns the substring before the first occurrence of the given needle.

```
$string = "1:23:456";
echo json_encode(explode(":", $string)); // ["1","23","456"]
var_dump(strstr($string, ":")); // string(7) ":23:456"

var_dump(strstr($string, ":", true)); // string(1) "1"
```

Section 25.2: Substring

Substring returns the portion of string specified by the start and length parameters.

```
var_dump(substr("Boo", 1)); // string(2) "oo"
```

If there is a possibility of meeting multi-byte character strings, then it would be safer to use mb_substr.

```php
$cake = "cakeæøå";
var_dump(substr($cake, 0, 5)); // string(5) "cake"
var_dump(mb_substr($cake, 0, 5, 'UTF-8')); // string(6) "cakeæ"
```

Another variant is the substr_replace function, which replaces text within a portion of a string.

```php
var_dump(substr_replace("Boo", "0", 1, 1)); // string(3) "B0o"
var_dump(substr_Replace("Boo", "ts", strlen("Boo"))); // string(5) "Boots"
```

Let's say you want to find a specific word in a string - and don't want to use Regex.

```php
$hi = "Hello World!";
$bye = "Goodbye cruel World!";

var_dump(strpos($hi, " ")); // int(5)
var_dump(strpos($bye, " ")); // int(7)

var_dump(substr($hi, 0, strpos($hi, " "))); // string(5) "Hello"
var_dump(substr($bye, -1 * (strlen($bye) - strpos($bye, " ")))); // string(13) " cruel World!"

// If the casing in the text is not important, then using strtolower helps to compare strings
var_dump(substr($hi, 0, strpos($hi, " ")) == 'hello'); // bool(false)
var_dump(strtolower(substr($hi, 0, strpos($hi, " "))) == 'hello'); // bool(true)
```

Another option is a very basic parsing of an email.

```php
$email = "test@example.com";
$wrong = "foobar.co.uk";
$notld = "foo@bar";

$at = strpos($email, "@"); // int(4)
$wat = strpos($wrong, "@"); // bool(false)
$nat = strpos($notld , "@"); // int(3)

$domain = substr($email, $at + 1); // string(11) "example.com"
$womain = substr($wrong, $wat + 1); // string(11) "oobar.co.uk"
$nomain = substr($notld, $nat + 1); // string(3) "bar"

$dot = strpos($domain, "."); // int(7)
$wot = strpos($womain, "."); // int(5)
$not = strpos($nomain, "."); // bool(false)

$tld = substr($domain, $dot + 1); // string(3) "com"
$wld = substr($womain, $wot + 1); // string(5) "co.uk"
$nld = substr($nomain , $not + 1); // string(2) "ar"

// string(25) "test@example.com is valid"
if ($at && $dot) var_dump("$email is valid");
else var_dump("$email is invalid");

// string(21) "foobar.com is invalid"
if ($wat && $wot) var_dump("$wrong is valid");
else var_dump("$wrong is invalid");

// string(18) "foo@bar is invalid"
if ($nat && $not) var_dump("$notld is valid");
else var_dump("$notld is invalid");

// string(27) "foobar.co.uk is an UK email"
if ($tld == "co.uk") var_dump("$email is a UK address");
```

```
if ($wld == "co.uk") var_dump("$wrong is a UK address");
if ($nld == "co.uk") var_dump("$notld is a UK address");
```

Or even putting the "Continue reading" or "..." at the end of a blurb

```
$blurb = "Lorem ipsum dolor sit amet";
$limit = 20;

var_dump(substr($blurb, 0, $limit - 3) . '...'); // string(20) "Lorem ipsum dolor..."
```

Section 25.3: Searching a substring with strpos

`strpos` can be understood as the number of bytes in the haystack before the first occurrence of the needle.

```
var_dump(strpos("haystack", "hay")); // int(0)
var_dump(strpos("haystack", "stack")); // int(3)
var_dump(strpos("haystack", "stackoverflow"); // bool(false)
```

Checking if a substring exists

Be careful with checking against TRUE or FALSE because if a index of 0 is returned an if statement will see this as FALSE.

```
$pos = strpos("abcd", "a"); // $pos = 0;
$pos2 = strpos("abcd", "e"); // $pos2 = FALSE;

// Bad example of checking if a needle is found.
if($pos) { // 0 does not match with TRUE.
    echo "1. I found your string\n";
}
else {
    echo "1. I did not found your string\n";
}

// Working example of checking if needle is found.
if($pos !== FALSE) {
    echo "2. I found your string\n";
}
else {
    echo "2. I did not found your string\n";
}

// Checking if a needle is not found
if($pos2 === FALSE) {
    echo "3. I did not found your string\n";
}
else {
    echo "3. I found your string\n";
}
```

Output of the whole example:

```
1. I did not found your string
2. I found your string
3. I did not found your string
```

Search starting from an offset

```
// With offset we can search ignoring anything before the offset
```

```php
$needle = "Hello";
$haystack = "Hello world! Hello World";

$pos = strpos($haystack, $needle, 1); // $pos = 13, not 0
```

Get all occurrences of a substring

```php
$haystack = "a baby, a cat, a donkey, a fish";
$needle = "a ";
$offsets = [];
// start searching from the beginning of the string
for($offset = 0;
        // If our offset is beyond the range of the
        // string, don't search anymore.
        // If this condition is not set, a warning will
        // be triggered if $haystack ends with $needle
        // and $needle is only one byte long.
        $offset < strlen($haystack); ){
    $pos = strpos($haystack, $needle, $offset);
    // we don't have anymore substrings
    if($pos === false) break;
    $offsets[] = $pos;
    // You may want to add strlen($needle) instead,
    // depending on whether you want to count "aaa"
    // as 1 or 2 "aa"s.
    $offset = $pos + 1;
}
echo json_encode($offsets); // [0,8,15,25]
```

Section 25.4: Parsing string using regular expressions

preg_match can be used to parse string using regular expression. The parts of expression enclosed in parenthesis are called subpatterns and with them you can pick individual parts of the string.

```php
$str = "<a href=\"http://example.org\">My Link</a>";
$pattern = "/<a href=\"(.*)\">(.*)<\/a>/";
$result = preg_match($pattern, $str, $matches);
if($result === 1) {
    // The string matches the expression
    print_r($matches);
} else if($result === 0) {
    // No match
} else {
    // Error occurred
}
```

Output

```
Array
(
    [0] => <a href="http://example.org">My Link</a>
    [1] => http://example.org
    [2] => My Link
)
```

Chapter 26: Classes and Objects

Classes and Objects are used to to make your code more efficient and less repetitive by grouping similar tasks.

A class is used to define the actions and data structure used to build objects. The objects are then built using this predefined structure.

Section 26.1: Class Constants

Class constants provide a mechanism for holding fixed values in a program. That is, they provide a way of giving a name (and associated compile-time checking) to a value like `3.14` or `"Apple"`. Class constants can only be defined with the **`const`** keyword - the define function cannot be used in this context.

As an example, it may be convenient to have a shorthand representation for the value of π throughout a program. A class with **`const`** values provides a simple way to hold such values.

```
class MathValues {
    const PI = M_PI;
    const PHI = 1.61803;
}

$area = MathValues::PI * $radius * $radius;
```

Class constants may be accessed by using the double colon operator (so-called the scope resolution operator) on a class, much like static variables. Unlike static variables, however, class constants have their values fixed at compile time and cannot be reassigned to (e.g. `MathValues::PI = 7` would produce a fatal error).

Class constants are also useful for defining things internal to a class that might need changing later (but do not change frequently enough to warrant storing in, say, a database). We can reference this internally using the **`self`** scope resolutor (which works in both instanced and static implementations)

```
class Labor {
    /** How long, in hours, does it take to build the item? */
    const LABOR_UNITS = 0.26;
    /** How much are we paying employees per hour? */
    const LABOR_COST = 12.75;

    public function getLaborCost($number_units) {
         return (self::LABOR_UNITS * self::LABOR_COST) * $number_units;
    }
}
```

Class constants can only contain scalar values in versions < 5.6

As of PHP 5.6 we can use expressions with constants, meaning math statements and strings with concatenation are acceptable constants

```
class Labor {
    /** How much are we paying employees per hour? Hourly wages * hours taken to make */
    const LABOR_COSTS = 12.75 * 0.26;

    public function getLaborCost($number_units) {
         return self::LABOR_COSTS * $number_units;
    }
}
```

As of PHP 7.0, constants declared with `define` may now contain arrays.

```
define("BAZ", array('baz'));
```

Class constants are useful for more than just storing mathematical concepts. For example, if preparing a pie, it might be convenient to have a single `Pie` class capable of taking different kinds of fruit.

```
class Pie {
    protected $fruit;

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

We can then use the `Pie` class like so

```
$pie = new Pie("strawberry");
```

The problem that arises here is, when instantiating the `Pie` class, no guidance is provided as to the acceptable values. For example, when making a "boysenberry" pie, it might be misspelled "boisenberry". Furthermore, we might not support a plum pie. Instead, it would be useful to have a list of acceptable fruit types already defined somewhere it would make sense to look for them. Say a class named `Fruit`:

```
class Fruit {
    const APPLE = "apple";
    const STRAWBERRY = "strawberry";
    const BOYSENBERRY = "boysenberry";
}

$pie = new Pie(Fruit::STRAWBERRY);
```

Listing the acceptable values as class constants provides a valuable hint as to the acceptable values which a method accepts. It also ensures that misspellings cannot make it past the compiler. While `new Pie('aple')` and `new Pie('apple')` are both acceptable to the compiler, `new Pie(Fruit::APLE)` will produce a compiler error.

Finally, using class constants means that the actual value of the constant may be modified in a single place, and any code using the constant automatically has the effects of the modification.

Whilst the most common method to access a class constant is `MyClass::CONSTANT_NAME`, it may also be accessed by:

```
echo MyClass::CONSTANT;

$classname = "MyClass";
echo $classname::CONSTANT; // As of PHP 5.3.0
```

Class constants in PHP are conventionally named all in uppercase with underscores as word separators, although any valid label name may be used as a class constant name.

As of PHP 7.1, class constants may now be defined with different visibilities from the default public scope. This means that both protected and private constants can now be defined to prevent class constants from unnecessarily leaking into the public scope (see Method and Property Visibility). For example:

```
class Something {
    const PUBLIC_CONST_A = 1;
```

```
    public const PUBLIC_CONST_B = 2;
    protected const PROTECTED_CONST = 3;
    private const PRIVATE_CONST = 4;
}
```

define vs class constants

Although this is a valid construction:

```
function bar() { return 2; };

define('BAR', bar());
```

If you try to do the same with class constants, you'll get an error:

```
function bar() { return 2; };

class Foo {
    const BAR = bar(); // Error: Constant expression contains invalid operations
}
```

But you can do:

```
function bar() { return 2; };

define('BAR', bar());

class Foo {
    const BAR = BAR; // OK
}
```

For more information, see constants in the manual.

Using ::class to retrieve class's name

PHP 5.5 introduced the `::class` syntax to retrieve the full class name, taking namespace scope and `use` statements into account.

```
namespace foo;
use bar\Bar;
echo json_encode(Bar::class); // "bar\\Bar"
echo json_encode(Foo::class); // "foo\\Foo"
echo json_encode(\Foo::class); // "Foo"
```

The above works even if the classes are not even defined (i.e. this code snippet works alone).

This syntax is useful for functions that require a class name. For example, it can be used with `class_exists` to check a class exists. No errors will be generated regardless of return value in this snippet:

```
class_exists(ThisClass\Will\NeverBe\Loaded::class, false);
```

Section 26.2: Abstract Classes

An abstract class is a class that cannot be instantiated. Abstract classes can define abstract methods, which are methods without any body, only a definition:

```
abstract class MyAbstractClass {
```

```
    abstract public function doSomething($a, $b);
}
```

Abstract classes should be extended by a child class which can then provide the implementation of these abstract methods.

The main purpose of a class like this is to provide a kind of template that allows children classes to inherit from, "forcing" a structure to adhere to. Lets elaborate on this with an example:

In this example we will be implementing a `Worker` interface. First we define the interface:

```
interface Worker {
    public function run();
}
```

To ease the development of further Worker implementations, we will create an abstract worker class that already provides the `run()` method from the interface, but specifies some abstract methods that need to be filled in by any child class:

```
abstract class AbstractWorker implements Worker {
    protected $pdo;
    protected $logger;

    public function __construct(PDO $pdo, Logger $logger) {
        $this->pdo = $pdo;
        $this->logger = $logger;
    }

    public function run() {
        try {
            $this->setMemoryLimit($this->getMemoryLimit());
            $this->logger->log("Preparing main");
            $this->prepareMain();
            $this->logger->log("Executing main");
            $this->main();
        } catch (Throwable $e) {
            // Catch and rethrow all errors so they can be logged by the worker
            $this->logger->log("Worker failed with exception: {$e->getMessage()}");
            throw $e;
        }
    }

    private function setMemoryLimit($memoryLimit) {
        ini_set('memory_limit', $memoryLimit);
        $this->logger->log("Set memory limit to $memoryLimit");
    }

    abstract protected function getMemoryLimit();

    abstract protected function prepareMain();

    abstract protected function main();
}
```

First of all, we have provided an abstract method `getMemoryLimit()`. Any class extending from `AbstractWorker` needs to provide this method and return its memory limit. The `AbstractWorker` then sets the memory limit and logs it.

Secondly the `AbstractWorker` calls the `prepareMain()` and `main()` methods, after logging that they have been

called.

Finally, all of these method calls have been grouped in a `try-catch` block. So if any of the abstract methods defined by the child class throws an exception, we will catch that exception, log it and rethrow it. This prevents all child classes from having to implement this themselves.

Now lets define a child class that extends from the `AbstractWorker`:

```
class TranscactionProcessorWorker extends AbstractWorker {
    private $transactions;

    protected function getMemoryLimit() {
        return "512M";
    }

    protected function prepareMain() {
        $stmt = $this->pdo->query("SELECT * FROM transactions WHERE processed = 0 LIMIT 500");
        $stmt->execute();
        $this->transactions = $stmt->fetchAll();
    }

    protected function main() {
        foreach ($this->transactions as $transaction) {
            // Could throw some PDO or MYSQL exception, but that is handled by the AbstractWorker
            $stmt = $this->pdo->query("UPDATE transactions SET processed = 1 WHERE id =
{$transaction['id']} LIMIT 1");
            $stmt->execute();
        }
    }
}
```

As you can see, the `TransactionProcessorWorker` was rather easy to implement, as we only had to specify the memory limit and worry about the actual actions that it needed to perform. No error handling is needed in the `TransactionProcessorWorker` because that is handled in the `AbsractWorker`.

Important Note

> When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child (or the child itself must also be marked abstract); additionally, these methods must be defined with the same (or a less restricted) visibility. For example, if the abstract method is defined as protected, the function implementation must be defined as either protected or public, but not private.

Taken from the PHP Documentation for Class Abstraction.

If you **do not** define the parent abstract classes methods within the child class, you will be thrown a **Fatal PHP Error** like the following.

> **Fatal error:** Class X contains 1 abstract method and must therefore be declared abstract or implement the remaining methods (X::x) in

Section 26.3: Late static binding

In PHP 5.3+ and above you can utilize late static binding to control which class a static property or method is called

from. It was added to overcome the problem inherent with the `self::` scope resolutor. Take the following code

```
class Horse {
    public static function whatToSay() {
        echo 'Neigh!';
    }

    public static function speak() {
        self::whatToSay();
    }
}

class MrEd extends Horse {
    public static function whatToSay() {
        echo 'Hello Wilbur!';
    }
}
```

You would expect that the `MrEd` class will override the parent `whatToSay()` function. But when we run this we get something unexpected

```
Horse::speak(); // Neigh!
MrEd::speak(); // Neigh!
```

The problem is that `self::whatToSay();` can only refer to the `Horse` class, meaning it doesn't obey `MrEd`. If we switch to the `static::` scope resolutor, we don't have this problem. This newer method tells the class to obey the instance calling it. Thus we get the inheritance we're expecting

```
class Horse {
    public static function whatToSay() {
        echo 'Neigh!';
    }

    public static function speak() {
        static::whatToSay(); // Late Static Binding
    }
}

Horse::speak(); // Neigh!
MrEd::speak(); // Hello Wilbur!
```

Section 26.4: Namespacing and Autoloading

Technically, autoloading works by executing a callback when a PHP class is required but not found. Such callbacks usually attempt to load these classes.

Generally, autoloading can be understood as the attempt to load PHP files (especially PHP class files, where a PHP source file is dedicated for a specific class) from appropriate paths according to the class's fully-qualified name (FQN) when a class is needed.

Suppose we have these classes:

Class file for `application\controllers\Base`:

```
<?php
namespace application\controllers { class Base {...} }
```

Class file for `application\controllers\Control`:

```
<?php
namespace application\controllers { class Control {...} }
```

Class file for `application\models\Page`:

```
<?php
namespace application\models { class Page {...} }
```

Under the source folder, these classes should be placed at the paths as their FQNs respectively:

- Source folder
  - `applications`
    - `controllers`
      - `Base.php`
      - `Control.php`
    - `models`
      - `Page.php`

This approach makes it possible to programmatically resolve the class file path according to the FQN, using this function:

```
function getClassPath(string $sourceFolder, string $className, string $extension = ".php") {
    return $sourceFolder . "/" . str_replace("\\", "/", $className) . $extension; // note that "/"
works as a directory separator even on Windows
}
```

The `spl_autoload_register` function allows us to load a class when needed using a user-defined function:

```
const SOURCE_FOLDER = __DIR__ . "/src";
spl_autoload_register(function (string $className) {
    $file = getClassPath(SOURCE_FOLDER, $className);
    if (is_readable($file)) require_once $file;
});
```

This function can be further extended to use fallback methods of loading:

```
const SOURCE_FOLDERS = [__DIR__ . "/src", "/root/src"]);
spl_autoload_register(function (string $className) {
    foreach(SOURCE_FOLDERS as $folder) {
        $extensions = [
            // do we have src/Foo/Bar.php5_int64?
            ".php" . PHP_MAJOR_VERSION . "_int" . (PHP_INT_SIZE * 8),
            // do we have src/Foo/Bar.php7?
            ".php" . PHP_MAJOR_VERSION,
            // do we have src/Foo/Bar.php_int64?
            ".php" . "_int" . (PHP_INT_SIZE * 8),
            // do we have src/Foo/Bar.phps?
            ".phps"
            // do we have src/Foo/Bar.php?
            ".php"
        ];
        foreach($extensions as $ext) {
            $path = getClassPath($folder, $className, $extension);
            if(is_readable($path)) return $path;
        }
```

```
    }
});
```

Note that PHP doesn't attempt to load the classes whenever a file that uses this class is loaded. It may be loaded in the middle of a script, or even in shutdown functions . This is one of the reasons why developers, especially those who use autoloading, should avoid replacing executing source files in the runtime, especially in phar files.

Section 26.5: Method and Property Visibility

There are three visibility types that you can apply to methods (*class/object functions*) and properties (*class/object variables*) within a class, which provide access control for the method or property to which they are applied.

You can read extensively about these in the PHP Documentation for OOP Visibility.

Public

Declaring a method or a property as **`public`** allows the method or property to be accessed by:

- The class that declared it.
- The classes that extend the declared class.
- Any external objects, classes, or code outside the class hierarchy.

An example of this **`public`** access would be:

```
class MyClass {
    // Property
    public $myProperty = 'test';

    // Method
    public function myMethod() {
        return $this->myProperty;
    }
}

$obj = new MyClass();
echo $obj->myMethod();
// Out: test

echo $obj->myProperty;
// Out: test
```

Protected

Declaring a method or a property as **`protected`** allows the method or property to be accessed by:

- The class that declared it.
- The classes that extend the declared class.

This **does not allow** external objects, classes, or code outside the class hierarchy to access these methods or properties. If something using this method/property does not have access to it, it will not be available, and an error will be thrown. **Only** instances of the declared self (or subclasses thereof) have access to it.

An example of this **`protected`** access would be:

```
class MyClass {
    protected $myProperty = 'test';
```

```
    protected function myMethod() {
        return $this->myProperty;
    }
}

class MySubClass extends MyClass {
    public function run() {
        echo $this->myMethod();
    }
}

$obj = new MySubClass();
$obj->run(); // This will call MyClass::myMethod();
// Out: test

$obj->myMethod(); // This will fail.
// Out: Fatal error: Call to protected method MyClass::myMethod() from context ''
```

The example above notes that you can only access the ***`protected`*** *elements within it's own scope. Essentially: "What's in the house can only be access from inside the house."*

Private

Declaring a method or a property as **`private`** allows the method or property to be accessed by:

- The class that declared it **Only** (not subclasses).

A **`private`** method or property is only visible and accessible within the class that created it.

Note that objects of the same type will have access to each others private and protected members even though they are not the same instances.

```
class MyClass {
    private $myProperty = 'test';

    private function myPrivateMethod() {
        return $this->myProperty;
    }

    public function myPublicMethod() {
        return $this->myPrivateMethod();
    }

    public function modifyPrivatePropertyOf(MyClass $anotherInstance) {
        $anotherInstance->myProperty = "new value";
    }
}

class MySubClass extends MyClass {
    public function run() {
        echo $this->myPublicMethod();
    }

    public function runWithPrivate() {
        echo $this->myPrivateMethod();
    }
}

$obj = new MySubClass();
```

```php
$newObj = new MySubClass();

// This will call MyClass::myPublicMethod(), which will then call
// MyClass::myPrivateMethod();
$obj->run();
// Out: test


$obj->modifyPrivatePropertyOf($newObj);

$newObj->run();
// Out: new value

echo $obj->myPrivateMethod(); // This will fail.
// Out: Fatal error: Call to private method MyClass::myPrivateMethod() from context ''

echo $obj->runWithPrivate(); // This will also fail.
// Out: Fatal error: Call to private method MyClass::myPrivateMethod() from context 'MySubClass'
```

*As noted, you can only access the **`private`** method/property from within it's defined class.*

Section 26.6: Interfaces

Introduction

Interfaces are definitions of the public APIs classes must implement to satisfy the interface. They work as "contracts", specifying **what** a set of subclasses does, but **not how** they do it.

Interface definition is much alike class definition, changing the keyword **`class`** to **`interface`**:

```php
interface Foo {

}
```

Interfaces can contain methods and/or constants, but no attributes. Interface constants have the same restrictions as class constants. Interface methods are implicitly abstract:

```php
interface Foo {
    const BAR = 'BAR';

    public function doSomething($param1, $param2);
}
```

Note: interfaces **must not** declare constructors or destructors, since these are implementation details on the class level.

Realization

Any class that needs to implement an interface must do so using the **`implements`** keyword. To do so, the class needs to provide a implementation for every method declared in the interface, respecting the same signature.

A single class **can** implement more than one interface at a time.

```php
interface Foo {
    public function doSomething($param1, $param2);
}

interface Bar {
```

```
    public function doAnotherThing($param1);
}


class Baz implements Foo, Bar {
    public function doSomething($param1, $param2) {
        // ...
    }

    public function doAnotherThing($param1) {
        // ...
    }
}
```

When abstract classes implement interfaces, they do not need to implement all methods. Any method not implemented in the base class must then be implemented by the concrete class that extends it:

```
abstract class AbstractBaz implements Foo, Bar {
    // Partial implementation of the required interface...
    public function doSomething($param1, $param2) {
        // ...
    }
}

class Baz extends AbstractBaz {
    public function doAnotherThing($param1) {
        // ...
    }
}
```

Notice that interface realization is an inherited characteristic. When extending a class that implements an interface, you do not need to redeclare it in the concrete class, because it is implicit.

> **Note:** Prior to PHP 5.3.9, a class could not implement two interfaces that specified a method with the same name, since it would cause ambiguity. More recent versions of PHP allow this as long as the duplicate methods have the same signature[1].

Inheritance

Like classes, it is possible to establish an inheritance relationship between interfaces, using the same keyword **`extends`**. The main difference is that multiple inheritance is allowed for interfaces:

```
interface Foo {

}

interface Bar {

}

interface Baz extends Foo, Bar {

}
```

Examples

In the example bellow we have a simple example interface for a vehicle. Vehicles can go forwards and backwards.

```
interface VehicleInterface {
    public function forward();

    public function reverse();

    ...
}

class Bike implements VehicleInterface {
    public function forward() {
        $this->pedal();
    }

    public function reverse() {
        $this->backwardSteps();
    }

    protected function pedal() {
        ...
    }

    protected function backwardSteps() {
        ...
    }

    ...
}

class Car implements VehicleInterface {
    protected $gear = 'N';

    public function forward() {
        $this->setGear(1);
        $this->pushPedal();
    }

    public function reverse() {
        $this->setGear('R');
        $this->pushPedal();
    }

    protected function setGear($gear) {
        $this->gear = $gear;
    }

    protected function pushPedal() {
        ...
    }

    ...
}
```

Then we create two classes that implement the interface: Bike and Car. Bike and Car internally are very different, but both are vehicles, and must implement the same public methods that VehicleInterface provides.

Typehinting allows methods and functions to request Interfaces. Let's assume that we have a parking garage class, which contains vehicles of all kinds.

```
class ParkingGarage {
    protected $vehicles = [];
```

```
    public function addVehicle(VehicleInterface $vehicle) {
        $this->vehicles[] = $vehicle;
    }
}
```

Because `addVehicle` requires a `$vehicle` of type `VehicleInterface`—not a concrete implementation—we can input both Bikes and Cars, which the ParkingGarage can manipulate and use.

Section 26.7: Final Keyword

Def: **Final** Keyword prevents child classes from overriding a method by prefixing the definition with final. If the class itself is being defined final then it cannot be extended

Final Method

```
class BaseClass {
   public function test() {
       echo "BaseClass::test() called\n";
   }

   final public function moreTesting() {
       echo "BaseClass::moreTesting() called\n";
   }
}

class ChildClass extends BaseClass {
   public function moreTesting() {
       echo "ChildClass::moreTesting() called\n";
   }
}
// Results in Fatal error: Cannot override final method BaseClass::moreTesting()
```

Final Class:

```
final class BaseClass {
   public function test() {
       echo "BaseClass::test() called\n";
   }

   // Here it doesn't matter if you specify the function as final or not
   final public function moreTesting() {
       echo "BaseClass::moreTesting() called\n";
   }
}

class ChildClass extends BaseClass {
}
// Results in Fatal error: Class ChildClass may not inherit from final class (BaseClass)
```

Final constants: Unlike Java, the `final` keyword is not used for class constants in PHP. Use the keyword **`const`** instead.

Why do I have to use `final`?

1. Preventing massive inheritance chain of doom
2. Encouraging composition
3. Force the developer to think about user public API
4. Force the developer to shrink an object's public API

5. A `final` class can always be made extensible
6. **`extends`** breaks encapsulation
7. You don't need that flexibility
8. You are free to change the code

When to avoid `final`: Final classes only work effectively under following assumptions:

1. There is an abstraction (interface) that the final class implements
2. All of the public API of the final class is part of that interface

Section 26.8: Autoloading

Nobody wants to `require` or `include` every time a class or inheritance is used. Because it can be painful and is easy to forget, PHP is offering so called autoloading. If you are already using Composer, read about autoloading using Composer.

What exactly is autoloading?

The name basically says it all. You do not have to get the file where the requested class is stored in, but PHP *auto*matically *load*s it.

How can I do this in basic PHP without third party code?

There is the function `__autoload`, but it is considered better practice to use `spl_autoload_register`. These functions will be considered by PHP every time a class is not defined within the given space. So adding autoload to an existing project is no problem, as defined classes (via `require` i.e.) will work like before. For the sake of preciseness, the following examples will use anonymous functions, if you use PHP < 5.3, you can define the function and pass it's name as argument to `spl_autoload_register`.

Examples

```
spl_autoload_register(function ($className) {
    $path = sprintf('%s.php', $className);
    if (file_exists($path)) {
        include $path;
    } else {
        // file not found
    }
});
```

The code above simply tries to include a filename with the class name and the appended extension ".php" using `sprintf`. If `FooBar` needs to be loaded, it looks if `FooBar.php` exists and if so includes it.

Of course this can be extended to fit the project's individual need. If `_` inside a class name is used to group, e.g. `User_Post` and `User_Image` both refer to `User`, both classes can be kept in a folder called "User" like so:

```
spl_autoload_register(function ($className) {
    //                      replace _ by / or \ (depending on OS)
    $path = sprintf('%s.php', str_replace('_', DIRECTORY_SEPARATOR, $className) );
    if (file_exists($path)) {
        include $path;
    } else {
        // file not found
    }
});
```

The class `User_Post` will now be loaded from "User/Post.php", etc.

`spl_autoload_register` can be tailored to various needs. All your files with classes are named "class.CLASSNAME.php"? No problem. Various nesting (`User_Post_Content` => "User/Post/Content.php")? No problem either.

If you want a more elaborate autoloading mechanism - and still don't want to include Composer - you can work without adding third party libraries.

```
spl_autoload_register(function ($className) {
    $path = sprintf('%1$s%2$s%3$s.php',
        // %1$s: get absolute path
        realpath(dirname(__FILE__)),
        // %2$s: / or \ (depending on OS)
        DIRECTORY_SEPARATOR,
        // %3$s: don't wory about caps or not when creating the files
        strtolower(
            // replace _ by / or \ (depending on OS)
            str_replace('_', DIRECTORY_SEPARATOR, $className)
        )
    );

    if (file_exists($path)) {
        include $path;
    } else {
        throw new Exception(
            sprintf('Class with name %1$s not found. Looked in %2$s.',
                $className,
                $path
            )
        );
    }
});
```

Using autoloaders like this, you can happily write code like this:

```
require_once './autoload.php'; // where spl_autoload_register is defined

$foo = new Foo_Bar(new Hello_World());
```

Using classes:

```
class Foo_Bar extends Foo {}
```

```
class Hello_World implements Demo_Classes {}
```

These examples will be include classes from `foo/bar.php`, `foo.php`, `hello/world.php` and `demo/classes.php`.

Section 26.9: Calling a parent constructor when instantiating a child

A common pitfall of child classes is that, if your parent and child both contain a constructor(`__construct()`) method, **only the child class constructor will run**. There may be occasions where you need to run the parent `__construct()` method from it's child. If you need to do that, then you will need to use the `parent::` scope resolutor:

```
parent::__construct();
```

Now harnessing that within a real-world situation would look something like:

```
class Foo {

    function __construct($args) {
        echo 'parent';
    }

}

class Bar extends Foo {

    function __construct($args) {
        parent::__construct($args);
    }
}
```

The above will run the parent `__construct()` resulting in the `echo` being run.

Section 26.10: Dynamic Binding

Dynamic binding, also referred as **method overriding** is an example of **run time polymorphism** that occurs when multiple classes contain different implementations of the same method, but the object that the method will be called on is *unknown* until **run time**.

This is useful if a certain condition dictates which class will be used to perform an action, where the action is named the same in both classes.

```
interface Animal {
    public function makeNoise();
}

class Cat implements Animal {
    public function makeNoise
    {
        $this->meow();
    }
    ...
}

class Dog implements Animal {
    public function makeNoise {
        $this->bark();
    }
    ...
}

class Person {
    const CAT = 'cat';
    const DOG = 'dog';

    private $petPreference;
    private $pet;

    public function isCatLover(): bool {
        return $this->petPreference == self::CAT;
    }

    public function isDogLover(): bool {
```

```
        return $this->petPreference == self::DOG;
    }

    public function setPet(Animal $pet) {
        $this->pet = $pet;
    }

    public function getPet(): Animal {
        return $this->pet;
    }
}

if($person->isCatLover()) {
    $person->setPet(new Cat());
} else if($person->isDogLover()) {
    $person->setPet(new Dog());
}

$person->getPet()->makeNoise();
```

In the above example, the `Animal` class (`Dog|Cat`) which will `makeNoise` is unknown until run time depending on the property within the `User` class.

Section 26.11: $this, self and static plus the singleton

> Use `$this` to refer to the current object. Use **`self`** to refer to the current class. In other words, use `$this->member` for non-static members, use **`self`**`::$member` for static members.

In the example below, `sayHello()` and `sayGoodbye()` are using **`self`** and `$this` difference can be observed here.

```
class Person {
    private $name;

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

    public function getName() {
        return $this->name;
    }

    public function getTitle() {
        return $this->getName()." the person";
    }

    public function sayHello() {
        echo "Hello, I'm ".$this->getTitle()."<br/>";
    }

    public function sayGoodbye() {
        echo "Goodbye from ".self::getTitle()."<br/>";
    }
}

class Geek extends Person {
    public function __construct($name) {
        parent::__construct($name);
    }
```

```
    public function getTitle() {
        return $this->getName()." the geek";
    }
}

$geekObj = new Geek("Ludwig");
$geekObj->sayHello();
$geekObj->sayGoodbye();
```

`static` refers to whatever class in the hierarchy you called the method on. It allows for better reuse of static class properties when classes are inherited.

Consider the following code:

```
class Car {
    protected static $brand = 'unknown';

    public static function brand() {
         return self::$brand."\n";
    }
}

class Mercedes extends Car {
    protected static $brand = 'Mercedes';
}

class BMW extends Car {
    protected static $brand = 'BMW';
}

echo (new Car)->brand();
echo (new BMW)->brand();
echo (new Mercedes)->brand();
```

This doesn't produce the result you want:

> unknown
> unknown
> unknown

That's because **`self`** refers to the `Car` class whenever method `brand()` is called.

To refer to the correct class, you need to use `static` instead:

```
class Car {
    protected static $brand = 'unknown';

    public static function brand() {
         return static::$brand."\n";
    }
}

class Mercedes extends Car {
    protected static $brand = 'Mercedes';
}

class BMW extends Car {
    protected static $brand = 'BMW';
```

```
}

echo (new Car)->brand();
echo (new BMW)->brand();
echo (new Mercedes)->brand();
```

This does produce the desired output:

> unknown
> BMW
> Mercedes

See also Late static binding

The singleton

If you have an object that's expensive to create or represents a connection to some external resource you want to reuse, i.e. a database connection where there is no connection pooling or a socket to some other system, you can use the `static` and **`self`** keywords in a class to make it a singleton. There are strong opinions about whether the singleton pattern should or should not be used, but it does have its uses.

```
class Singleton {
    private static $instance = null;

    public static function getInstance(){
        if(!isset(self::$instance)){
            self::$instance = new self();
        }

        return self::$instance;
    }

    private function __construct() {
        // Do constructor stuff
    }
}
```

As you can see in the example code we are defining a private static property `$instance` to hold the object reference. Since this is static this reference is shared across ALL objects of this type.

The `getInstance()`method uses a method know as lazy instantiation to delay creating the object to the last possible moment as you do not want to have unused objects lying around in memory never intended to be used. It also saves time and CPU on page load not having to load more objects than necessary. The method is checking if the object is set, creating it if not, and returning it. This ensures that only one object of this kind is ever created.

We are also setting the constructor to be private to ensure that no one creates it with the **`new`** keyword from the outside. If you need to inherit from this class just change the **`private`** keywords to **`protected`**.

To use this object you just write the following:

```
$singleton = Singleton::getInstance();
```

Now I DO implore you to use dependency injection where you can and aim for loosely coupled objects, but sometimes that is just not reasonable and the singleton pattern can be of use.

Section 26.12: Defining a Basic Class

An object in PHP contains variables and functions. Objects typically belong to a class, which defines the variables and functions that all objects of this class will contain.

The syntax to define a class is:

```
class Shape {
    public $sides = 0;

    public function description() {
        return "A shape with $this->sides sides.";
    }
}
```

Once a class is defined, you can create an instance using:

```
$myShape = new Shape();
```

Variables and functions on the object are accessed like this:

```
$myShape = new Shape();
$myShape->sides = 6;

print $myShape->description(); // "A shape with 6 sides"
```

Constructor

Classes can define a special `__construct()` method, which is executed as part of object creation. This is often used to specify the initial values of an object:

```
class Shape {
    public $sides = 0;

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

    public function description() {
        return "A shape with $this->sides sides.";
    }
}

$myShape = new Shape(6);

print $myShape->description(); // A shape with 6 sides
```

Extending Another Class

Class definitions can extend existing class definitions, adding new variables and functions as well as modifying those defined in the parent class.

Here is a class that extends the previous example:

```
class Square extends Shape {
    public $sideLength = 0;

    public function __construct($sideLength) {
       parent::__construct(4);
```

```
        $this->sideLength = $sideLength;
    }

    public function perimeter() {
        return $this->sides * $this->sideLength;
    }

    public function area() {
        return $this->sideLength * $this->sideLength;
    }
}
```

The `Square` class contains variables and behavior for both the `Shape` class and the `Square` class:

```
$mySquare = new Square(10);

print $mySquare->description()/ // A shape with 4 sides

print $mySquare->perimeter() // 40

print $mySquare->area() // 100
```

Section 26.13: Anonymous Classes

Anonymous classes were introduced into PHP 7 to enable for quick one-off objects to be easily created. They can take constructor arguments, extend other classes, implement interfaces, and use traits just like normal classes can.

In its most basic form, an anonymous class looks like the following:

```
new class("constructor argument") {
    public function __construct($param) {
        var_dump($param);
    }
}; // string(20) "constructor argument"
```

Nesting an anonymous class inside of another class does not give it access to private or protected methods or properties of that outer class. Access to protected methods and properties of the outer class can be gained by extending the outer class from the anonymous class. Access to private properties of the outer class can be gained by passing them through to the anonymous class's constructor.

For example:

```
class Outer {
    private $prop = 1;
    protected $prop2 = 2;

    protected function func1() {
        return 3;
    }

    public function func2() {
        // passing through the private $this->prop property
        return new class($this->prop) extends Outer {
            private $prop3;

            public function __construct($prop) {
                $this->prop3 = $prop;
```

```
            }

            public function func3() {
                // accessing the protected property Outer::$prop2
                // accessing the protected method Outer::func1()
                // accessing the local property self::$prop3 that was private from Outer::$prop
                return $this->prop2 + $this->func1() + $this->prop3;
            }
        };
    }
}

echo (new Outer)->func2()->func3(); // 6
```

Chapter 27: Namespaces

Section 27.1: Declaring namespaces

A namespace declaration can look as follows:

- `namespace MyProject;` - Declare the namespace `MyProject`
- `namespace MyProject\Security\Cryptography;` - Declare a nested namespace
- `namespace MyProject { ... }` - Declare a namespace with enclosing brackets.

It is recommended to only declare a single namespace per file, even though you can declare as many as you like in a single file:

```
namespace First {
    class A { ... }; // Define class A in the namespace First.
}

namespace Second {
    class B { ... }; // Define class B in the namespace Second.
}

namespace {
    class C { ... }; // Define class C in the root namespace.
}
```

Every time you declare a namespace, classes you define after that will belong to that namespace:

```
namespace MyProject\Shapes;

class Rectangle { ... }
class Square { ... }
class Circle { ... }
```

A namespace declaration can be used multiple times in different files. The example above defined three classes in the `MyProject\Shapes` namespace in a single file. Preferably this would be split up into three files, each starting with `namespace MyProject\Shapes;`. This is explained in more detail in the PSR-4 standard example.

Section 27.2: Referencing a class or function in a namespace

As shown in Declaring Namespaces, we can define a class in a namespace as follows:

```
namespace MyProject\Shapes;

class Rectangle { ... }
```

To reference this class the full path (including the namespace) needs to be used:

```
$rectangle = new MyProject\Shapes\Rectangle();
```

This can be shortened by importing the class via the **use**-statement:

```
// Rectangle becomes an alias to MyProject\Shapes\Rectangle
use MyProject\Shapes\Rectangle;

$rectangle = new Rectangle();
```

As for PHP 7.0 you can group various `use`-statements in one single statement using brackets:

```
use MyProject\Shapes\{
    Rectangle,          //Same as `use MyProject\Shapes\Rectangle`
    Circle,             //Same as `use MyProject\Shapes\Circle`
    Triangle,           //Same as `use MyProject\Shapes\Triangle`

    Polygon\FiveSides, //You can also import sub-namespaces
    Polygon\SixSides   //In a grouped `use`-statement
};

$rectangle = new Rectangle();
```

Sometimes two classes have the same name. This is not a problem if they are in a different namespace, but it could become a problem when attempting to import them with the `use`-statement:

```
use MyProject\Shapes\Oval;
use MyProject\Languages\Oval; // Apparantly Oval is also a language!
// Error!
```

This can be solved by defining a name for the alias yourself using the `as` keyword:

```
use MyProject\Shapes\Oval as OvalShape;
use MyProject\Languages\Oval as OvalLanguage;
```

To reference a class outside the current namespace, it has to be escaped with a `\`, otherwise a relative namespace path is assumed from the current namespace:

```
namespace MyProject\Shapes;

// References MyProject\Shapes\Rectangle. Correct!
$a = new Rectangle();

// References MyProject\Shapes\Rectangle. Correct, but unneeded!
$a = new \MyProject\Shapes\Rectangle();

// References MyProject\Shapes\MyProject\Shapes\Rectangle. Incorrect!
$a = new MyProject\Shapes\Rectangle();


// Referencing StdClass from within a namespace requires a \ prefix
// since it is not defined in a namespace, meaning it is global.

// References StdClass. Correct!
$a = new \StdClass();

// References MyProject\Shapes\StdClass. Incorrect!
$a = new StdClass();
```

Section 27.3: Declaring sub-namespaces

To declare a single namespace with hierarchy use following example:

```
namespace MyProject\Sub\Level;

const CONNECT_OK = 1;
class Connection { /* ... */ }
```

```
function connect() { /* ... */  }
```

The above example creates:

constant `MyProject\Sub\Level\CONNECT_OK`

class `MyProject\Sub\Level\Connection` and

function `MyProject\Sub\Level\connect`

Section 27.4: What are Namespaces?

The PHP community has a lot of developers creating lots of code. This means that one library's PHP code may use the same class name as another library. When both libraries are used in the same namespace, they collide and cause trouble.

Namespaces solve this problem. As described in the PHP reference manual, namespaces may be compared to operating system directories that namespace files; two files with the same name may co-exist in separate directories. Likewise, two PHP classes with the same name may co-exist in separate PHP namespaces.

It is important for you to namespace your code so that it may be used by other developers without fear of colliding with other libraries.

Chapter 28: Sessions

Section 28.1: session_start() Options

Starting with PHP Sessions we can pass an array with session-based `php.ini` options to the `session_start` function.

Example

```
<?php
   if (version_compare(PHP_VERSION, '7.0.0') >= 0) {
       // php >= 7 version
       session_start([
          'cache_limiter' => 'private',
          'read_and_close' => true,
       ]);
   } else {
       // php < 7 version
       session_start();
   }
?>
```

This feature also introduces a new `php.ini` setting named `session.lazy_write`, which defaults to `true` and means that session data is only rewritten, if it changes.

Referencing: https://wiki.php.net/rfc/session-lock-ini

Section 28.2: Session Locking

As we all are aware that PHP writes session data into a file at server side. When a request is made to php script which starts the session via `session_start()`, PHP locks this session file resulting to block/wait other incoming requests for same `session_id` to complete, because of which the other requests will get stuck on `session_start()` until or unless the **session file locked** is not released

The session file remains locked until the script is completed or session is manually closed. To avoid this situation *i.e. to prevent multiple requests getting blocked*, we can start the session and close the session which will release the lock from session file and allow to continue the remaining requests.

```
// php < 7.0
// start session
session_start();

// write data to session
$_SESSION['id'] = 123; // session file is locked, so other requests are blocked

// close the session, release lock
session_write_close();
```

Now one will think if session is closed how we will read the session values, beautify even after session is closed, session is still available. So, we can still read the session data.

```
echo $_SESSION['id'];    // will output  123
```

In **php >= 7.0**, we can have **read_only** session, **read_write** session and **lazy_write** session, so it may not required to use `session_write_close()`

Section 28.3: Manipulating session data

The `$_SESSION` variable is an array, and you can retrieve or manipulate it like a normal array.

```
<?php
// Starting the session
session_start();

// Storing the value in session
$_SESSION['id'] = 342;

// conditional usage of session values that may have been set in a previous session
if(!isset($_SESSION["login"])) {
    echo "Please login first";
    exit;
}
// now you can use the login safely
$user = $_SESSION["login"];

// Getting a value from the session data, or with default value,
//     using the Null Coalescing operator in PHP 7
$name = $_SESSION['name'] ?? 'Anonymous';
```

Also see Manipulating an Array for more reference how to work on an array.

Note that if you store an object in a session, it can be retrieved gracefully only if you have an class autoloader or you have loaded the class already. Otherwise, the object will come out as the type `__PHP_Incomplete_Class`, which may later lead to crashes. See Namespacing and Autoloading about autoloading.

Warning:

Session data can be hijacked. This is outlined in: *Pro PHP Security: From Application Security Principles to the Implementation of XSS Defense - Chapter 7: Preventing Session Hijacking* So it can be strongly recommended to never store any personal information in `$_SESSION`. This would most critically include **credit card numbers**, **government issued ids**, and **passwords**; but would also extend into less assuming data like **names**, **emails**, **phone numbers**, etc which would allow a hacker to impersonate/compromise a legitimate user. As a general rule, use worthless/non-personal values, such as numerical identifiers, in session data.

Section 28.4: Destroy an entire session

If you've got a session which you wish to destroy, you can do this with `session_destroy()`

```
/*
    Let us assume that our session looks like this:
    Array([firstname] => Jon, [id] => 123)

    We first need to start our session:
*/
session_start();

/*
    We can now remove all the values from the `SESSION` superglobal:
    If you omitted this step all of the global variables stored in the
    superglobal would still exist even though the session had been destroyed.
*/
$_SESSION = array();

// If it's desired to kill the session, also delete the session cookie.
```

```
// Note: This will destroy the session, and not just the session data!
if (ini_get("session.use_cookies")) {
    $params = session_get_cookie_params();
    setcookie(session_name(), '', time() - 42000,
        $params["path"], $params["domain"],
        $params["secure"], $params["httponly"]
    );
}

//Finally we can destroy the session:
session_destroy();
```

Using `session_destroy()` is different to using something like `$_SESSION = array();` which will remove all of the values stored in the `SESSION` superglobal but it will not destroy the actual stored version of the session.

Note: We use `$_SESSION = array();` instead of `session_unset()` because the manual stipulates:

> Only use session_unset() for older deprecated code that does not use $_SESSION.

Section 28.5: Safe Session Start With no Errors

Many developers have this problem when they work on huge projects, especially if they work on some modular CMS on plugins, addons, components etc. Here is solution for safe session start where if first checked PHP version to cover all versions and on next is checked if session is started. If session not exists then I start session safe. If session exists nothing happen.

```
if (version_compare(PHP_VERSION, '7.0.0') >= 0) {
    if(session_status() == PHP_SESSION_NONE) {
        session_start(array(
          'cache_limiter' => 'private',
          'read_and_close' => true,
       ));
    }
}
else if (version_compare(PHP_VERSION, '5.4.0') >= 0)
{
    if (session_status() == PHP_SESSION_NONE) {
        session_start();
    }
}
else
{
    if(session_id() == '') {
        session_start();
    }
}
```

This can help you a lot to avoid `session_start` error.

Section 28.6: Session name

Checking if session cookies have been created

Session name is the name of the cookie used to store sessions. You can use this to detect if cookies for a session have been created for the user:

```
if(isset($_COOKIE[session_name()])) {
    session_start();
}
```

Note that this method is generally not useful unless you really don't want to create cookies unnecessarily.

Changing session name

You can update the session name by calling `session_name()`.

```
//Set the session name
session_name('newname');
//Start the session
session_start();
```

If no argument is provided into `session_name()` then the current session name is returned.

> It should contain only alphanumeric characters; it should be short and descriptive (i.e. for users with enabled cookie warnings). The session name can't consist of digits only, at least one letter must be present. Otherwise a new session id is generated every time.

Chapter 29: Cookies

parameter	detail
name	The name of the cookie. This is also the key you can use to retrieve the value from the `$_COOKIE` super global. *This is the only required parameter*
value	The value to store in the cookie. This data is accessible to the browser so don't store anything sensitive here.
expire	A Unix timestamp representing when the cookie should expire. If set to zero the cookie will expire at the end of the session. If set to a number less than the current Unix timestamp the cookie will expire immediately.
path	The scope of the cookie. If set to `/` the cookie will be available within the entire domain. If set to `/some-path/` then the cookie will only be available in that path and descendants of that path. Defaults to the current path of the file that the cookie is being set in.
domain	The domain or subdomain the cookie is available on. If set to the bare domain `stackoverflow.com` then the cookie will be available to that domain and all subdomains. If set to a subdomain `meta.stackoverflow.com` then the cookie will be available only on that subdomain, and all sub-subdomains.
secure	When set to `TRUE` the cookie will only be set if a secure HTTPS connection exists between the client and the server.
httponly	Specifies that the cookie should only be made available through the HTTP/S protocol and should not be available to client side scripting languages like JavaScript. Only available in PHP 5.2 or later.

An HTTP cookie is a small piece of data sent from a website and stored on the user's computer by the user's web browser while the user is browsing.

Section 29.1: Modifying a Cookie

The value of a cookie can be modified by resetting the cookie

```
setcookie("user", "John", time() + 86400, "/"); // assuming there is a "user" cookie already
```

> Cookies are part of the HTTP header, so `setcookie()` must be called before any output is sent to the browser.

> When modifying a cookie make sure the `path` and `domain` parameters of `setcookie()` matches the existing cookie or a new cookie will be created instead.

> The value portion of the cookie will automatically be urlencoded when you send the cookie, and when it is received, it is automatically decoded and assigned to a variable by the same name as the cookie name

Section 29.2: Setting a Cookie

A cookie is set using the `setcookie()` function. Since cookies are part of the HTTP header, you must set any cookies before sending any output to the browser.

Example:

```
setcookie("user", "Tom", time() + 86400, "/"); // check syntax for function params
```

Description:

- Creates a cookie with name `user`
- (Optional) Value of the cookie is `Tom`
- (Optional) Cookie will expire in 1 day (86400 seconds)
- (Optional) Cookie is available throughout the whole website `/`
- (Optional) Cookie is only sent over HTTPS
- (Optional) Cookie is not accessible to scripting languages such as JavaScript

> A created or modified cookie can only be accessed on subsequent requests (where `path` and `domain` matches) as the superglobal `$_COOKIE`is not populated with the new data immediately.

Section 29.3: Checking if a Cookie is Set

Use the `isset()` function upon the superglobal `$_COOKIE` variable to check if a cookie is set.

Example:

```
// PHP <7.0
if (isset($_COOKIE['user'])) {
    // true, cookie is set
    echo 'User is ' . $_COOKIE['user'];
else {
    // false, cookie is not set
    echo 'User is not logged in';
}

// PHP 7.0+
echo 'User is ' . $_COOKIE['user'] ?? 'User is not logged in';
```

Section 29.4: Removing a Cookie

To remove a cookie, set the expiry timestamp to a time in the past. This triggers the browser's removal mechanism:

```
setcookie('user', '', time() - 3600, '/');
```

> When deleting a cookie make sure the `path` and `domain` parameters of `setcookie()` matches the cookie you're trying to delete or a new cookie, which expires immediately, will be created.

It is also a good idea to unset the `$_COOKIE` value in case the current page uses it:

```
unset($_COOKIE['user']);
```

Section 29.5: Retrieving a Cookie

Retrieve and Output a Cookie Named `user`

The value of a cookie can be retrieved using the global variable `$_COOKIE`. example if we have a cookie named `user` we can retrieve it like this

```
echo $_COOKIE['user'];
```

Chapter 30: Output Buffering

Function	Details
ob_start()	Starts the output buffer, any output placed after this will be captured and not displayed
ob_get_contents()	Returns all content captured by `ob_start()`
ob_end_clean()	Empties the output buffer and turns it off for the current nesting level
ob_get_clean()	Triggers both `ob_get_contents()` and `ob_end_clean()`
ob_get_level()	Returns the current nesting level of the output buffer
ob_flush()	Flush the content buffer and send it to the browser without ending the buffer
ob_implicit_flush()	Enables implicit flushing after every output call.
ob_end_flush()	Flush the content buffer and send it to the browser also ending the buffer

Section 30.1: Basic usage getting content between buffers and clearing

Output buffering allows you to store any textual content (Text, `HTML`) in a variable and send to the browser as one piece at the end of your script. By default, `php` sends your content as it interprets it.

```
<?php

// Turn on output buffering
ob_start();

// Print some output to the buffer (via php)
print 'Hello ';

// You can also `step out` of PHP
?>
<em>World</em>
<?php
// Return the buffer AND clear it
$content = ob_get_clean();

// Return our buffer and then clear it
# $content = ob_get_contents();
# $did_clear_buffer = ob_end_clean();

print($content);

#> "Hello <em>World</em>"
```

Any content outputted between `ob_start()` and `ob_get_clean()` will be captured and placed into the variable `$content`.

Calling `ob_get_clean()` triggers both `ob_get_contents()` and `ob_end_clean()`.

Section 30.2: Processing the buffer via a callback

You can apply any kind of additional processing to the output by passing a callable to `ob_start()`.

```
<?php
function clearAllWhiteSpace($buffer) {
    return str_replace(array("\n", "\t", ' '), '', $buffer);
}
```

```
ob_start('clearAllWhiteSpace');
?>
<h1>Lorem Ipsum</h1>

<p><strong>Pellentesque habitant morbi tristique</strong> senectus et netus et malesuada fames ac
turpis egestas. <a href="#">Donec non enim</a> in turpis pulvinar facilisis.</p>

<h2>Header Level 2</h2>

<ol>
   <li>Lorem ipsum dolor sit amet, consectetuer adipiscing elit.</li>
   <li>Aliquam tincidunt mauris eu risus.</li>
</ol>

<?php
/* Output will be flushed and processed when script ends or call
     ob_end_flush();
*/
```

Output:

```
<h1>LoremIpsum</h1><p><strong>Pellentesquehabitantmorbitristique</strong>senectusetnetusetmalesuada
famesacturpisegestas.<ahref="#">Donecnonenim</a>inturpispulvinarfacilisis.</p><h2>HeaderLevel2</h2>
<ol><li>Loremipsumdolorsitamet,consectetueradipiscingelit.</li><li>Aliquamtinciduntmauriseurisus.</
li></ol>
```

Section 30.3: Nested output buffers

You can nest output buffers and fetch the level for them to provide different content using the `ob_get_level()` function.

```
<?php

$i = 1;
$output = null;

while( $i <= 5 ) {
    // Each loop, creates a new output buffering `level`
    ob_start();
    print "Current nest level: ". ob_get_level() . "\n";
    $i++;
}

// We're at level 5 now
print 'Ended up at level: ' . ob_get_level() . PHP_EOL;

// Get clean will `pop` the contents of the top most level (5)
$output .= ob_get_clean();
print $output;

print 'Popped level 5, so we now start from 4' . PHP_EOL;

// We're now at level 4 (we pop'ed off 5 above)

// For each level we went up, come back down and get the buffer
while( $i > 2 ) {
    print "Current nest level: " . ob_get_level() . "\n";
    echo ob_get_clean();
    $i--;
```

```
}
```

Outputs:

```
Current nest level: 1
Current nest level: 2
Current nest level: 3
Current nest level: 4
Current nest level: 5
Ended up at level: 5
Popped level 5, so we now start from 4
Current nest level: 4
Current nest level: 3
Current nest level: 2
Current nest level: 1
```

Section 30.4: Running output buffer before any content

```
ob_start();

$user_count = 0;
foreach( $users as $user ) {
    if( $user['access'] != 7 ) { continue; }
    ?>
    <li class="users user-<?php echo $user['id']; ?>">
        <a href="<?php echo $user['link']; ?>">
            <?php echo $user['name'] ?>
        </a>
    </li>
<?php
    $user_count++;
}
$users_html = ob_get_clean();

if( !$user_count ) {
    header('Location: /404.php');
    exit();
}
?>
<html>
<head>
    <title>Level 7 user results (<?php echo $user_count; ?>)</title>
</head>

<body>
<h2>We have a total of <?php echo $user_count; ?> users with access level 7</h2>
<ul class="user-list">
    <?php echo $users_html; ?>
</ul>
</body>
</html>
```

In this example we assume `$users` to be a multidimensional array, and we loop through it to find all users with an access level of 7.

If there are no results, we redirect to an error page.

We are using the output buffer here because we are triggering a `header()` redirect based on the result of the loop

Section 30.5: Stream output to client

```
/**
 * Enables output buffer streaming. Calling this function
 * immediately flushes the buffer to the client, and any
 * subsequent output will be sent directly to the client.
 */
function _stream() {
    ob_implicit_flush(true);
    ob_end_flush();
}
```

Section 30.6: Using Output buffer to store contents in a file, useful for reports, invoices etc

```
<?php
ob_start();
?>
    <html>
    <head>
        <title>Example invoice</title>
    </head>
    <body>
    <h1>Invoice #0000</h1>
   <h2>Cost: &pound;15,000</h2>
    ...
    </body>
    </html>
<?php
$html = ob_get_clean();

$handle = fopen('invoices/example-invoice.html', 'w');
fwrite($handle, $html);
fclose($handle);
```

This example takes the complete document, and writes it to file, it does not output the document into the browser, but do by using `echo $html;`

Section 30.7: Typical usage and reasons for using ob_start

`ob_start` is especially handy when you have redirections on your page. For example, the following code won't work:

```
Hello!
<?php
  header("Location: somepage.php");
?>
```

The error that will be given is something like: `headers already sent by <xxx> on line <xxx>`.

In order to fix this problem, you would write something like this at the start of your page:

```
<?php
  ob_start();
?>
```

And something like this at the end of your page:

```php
<?php
  ob_end_flush();
?>
```

This stores all generated content into an output buffer, and displays it in one go. Hence, if you have any redirection calls on your page, those will trigger before any data is sent, removing the possibility of a `headers already sent` error occurring.

Section 30.8: Capturing the output buffer to re-use later

In this example, we have an array containing some data.

We capture the output buffer in `$items_li_html` and use it twice in the page.

```php
<?php

// Start capturing the output
ob_start();

$items = ['Home', 'Blog', 'FAQ', 'Contact'];

foreach($items as $item):

// Note we're about to step "out of PHP land"
?>
  <li><?php echo $item ?></li>
<?php
// Back in PHP land
endforeach;

// $items_lists contains all the HTML captured by the output buffer
$items_li_html = ob_get_clean();
?>

<!-- Menu 1: We can now re-use that (multiple times if required) in our HTML. -->
<ul class="header-nav">
    <?php echo $items_li_html ?>
</ul>

<!-- Menu 2 -->
<ul class="footer-nav">
    <?php echo $items_li_html ?>
</ul>
```

Save the above code in a file `output_buffer.php` and run it via `php output_buffer.php`.

You should see the 2 list items we created above with the same list items we generated in PHP using the output buffer:

```
<!-- Menu 1: We can now re-use that (multiple times if required) in our HTML. -->
<ul class="header-nav">
  <li>Home</li>
  <li>Blog</li>
  <li>FAQ</li>
  <li>Contact</li>
</ul>

<!-- Menu 2 -->
<ul class="footer-nav">
```

```
  <li>Home</li>
  <li>Blog</li>
  <li>FAQ</li>
  <li>Contact</li>
</ul>
```

Chapter 31: JSON

Parameter	Details
json_encode	-
value	The value being encoded. Can be any type except a resource. All string data must be UTF-8 encoded.
options	Bitmask consisting of JSON_HEX_QUOT, JSON_HEX_TAG, JSON_HEX_AMP, JSON_HEX_APOS, JSON_NUMERIC_CHECK, JSON_PRETTY_PRINT, JSON_UNESCAPED_SLASHES, JSON_FORCE_OBJECT, JSON_PRESERVE_ZERO_FRACTION, JSON_UNESCAPED_UNICODE, JSON_PARTIAL_OUTPUT_ON_ERROR. The behaviour of these constants is described on the JSON constants page.
depth	Set the maximum depth. Must be greater than zero.
json_decode	-
json	The json string being decoded. This function only works with UTF-8 encoded strings.
assoc	Should function return associative array instead of objects.
options	Bitmask of JSON decode options. Currently only JSON_BIGINT_AS_STRING is supported (default is to cast large integers as floats)

JSON (JavaScript Object Notation) is a platform and language independent way of serializing objects into plaintext. Because it is often used on web and so is PHP, there is a basic extension for working with JSON in PHP.

Section 31.1: Decoding a JSON string

The `json_decode()` function takes a JSON-encoded string as its first parameter and parses it into a PHP variable.

Normally, `json_decode()` will return an **object of \stdClass** if the top level item in the JSON object is a dictionary or an **indexed array** if the JSON object is an array. It will also return scalar values or `NULL` for certain scalar values, such as simple strings, `"true"`, `"false"`, and `"null"`. It also returns `NULL` on any error.

```
// Returns an object (The top level item in the JSON string is a JSON dictionary)
$json_string = '{"name": "Jeff", "age": 20, "active": true, "colors": ["red", "blue"]}';
$object = json_decode($json_string);
printf('Hello %s, You are %s years old.', $object->name, $object->age);
#> Hello Jeff, You are 20 years old.

// Returns an array (The top level item in the JSON string is a JSON array)
$json_string = '["Jeff", 20, true, ["red", "blue"]]';
$array = json_decode($json_string);
printf('Hello %s, You are %s years old.', $array[0], $array[1]);
```

Use `var_dump()` to view the types and values of each property on the object we decoded above.

```
// Dump our above $object to view how it was decoded
var_dump($object);
```

Output (note the variable types):

```
class stdClass#2 (4) {
["name"] => string(4) "Jeff"
 ["age"] => int(20)
 ["active"] => bool(true)
 ["colors"] =>
   array(2) {
     [0] => string(3) "red"
     [1] => string(4) "blue"
   }
```

```
}
```

Note: The variable **types** in JSON were converted to their PHP equivalent.

To return an associative array for JSON objects instead of returning an object, pass `true` as the second parameter to `json_decode()`.

```
$json_string = '{"name": "Jeff", "age": 20, "active": true, "colors": ["red", "blue"]}';
$array = json_decode($json_string, true); // Note the second parameter
var_dump($array);
```

Output (note the array associative structure):

```
array(4) {
  ["name"] => string(4) "Jeff"
  ["age"] => int(20)
  ["active"] => bool(true)
  ["colors"] =>
  array(2) {
    [0] => string(3) "red"
    [1] => string(4) "blue"
  }
}
```

The second parameter (`$assoc`) has no effect if the variable to be returned is not an object.

Note: If you use the `$assoc` parameter, you will lose the distinction between an empty array and an empty object. This means that running `json_encode()` on your decoded output again, will result in a different JSON structure.

If the JSON string has a "depth" more than 512 elements (*20 elements in versions older than 5.2.3, or 128 in version 5.2.3*) in recursion, the function `json_decode()` returns `NULL`. In versions 5.3 or later, this limit can be controlled using the third parameter (`$depth`), as discussed below.

According to the manual:

> PHP implements a superset of JSON as specified in the original » RFC 4627 - it will also encode and decode scalar types and NULL. RFC 4627 only supports these values when they are nested inside an array or an object. Although this superset is consistent with the expanded definition of "JSON text" in the newer » RFC 7159 (which aims to supersede RFC 4627) and » ECMA-404, this may cause interoperability issues with older JSON parsers that adhere strictly to RFC 4627 when encoding a single scalar value.

This means, that, for example, a simple string will be considered to be a valid JSON object in PHP:

```
$json = json_decode('"some string"', true);
var_dump($json, json_last_error_msg());
```

Output:

```
string(11) "some string"
string(8) "No error"
```

But simple strings, not in an array or object, are not part of the RFC 4627 standard. As a result, such online checkers as JSLint, JSON Formatter & Validator (in RFC 4627 mode) will give you an error.

There is a third `$depth` parameter for the depth of recursion (the default value is `512`), which means the amount of nested objects inside the original object to be decoded.

There is a fourth `$options` parameter. It currently accepts only one value, `JSON_BIGINT_AS_STRING`. The default behavior (which leaves off this option) is to cast large integers to floats instead of strings.

> Invalid non-lowercased variants of the true, false and null literals are no longer accepted as valid input.

So this example:

```
var_dump(json_decode('tRue'), json_last_error_msg());
var_dump(json_decode('tRUe'), json_last_error_msg());
var_dump(json_decode('tRUE'), json_last_error_msg());
var_dump(json_decode('TRUe'), json_last_error_msg());
var_dump(json_decode('TRUE'), json_last_error_msg());
var_dump(json_decode('true'), json_last_error_msg());
```

Before PHP 5.6:

```
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
```

And after:

```
NULL
string(12) "Syntax error"
NULL
string(12) "Syntax error"
NULL
string(12) "Syntax error"
NULL
string(12) "Syntax error"
NULL
string(12) "Syntax error"
bool(true)
string(8) "No error"
```

Similar behavior occurs for **`false`** and **`null`**.

Note that `json_decode()` will return **`NULL`** if the string cannot be converted.

```
$json = "{'name': 'Jeff', 'age': 20 }" ;  // invalid json

$person = json_decode($json);
echo $person->name;    //  Notice: Trying to get property of non-object: returns null
echo json_last_error();
#  4 (JSON_ERROR_SYNTAX)
```

```
echo json_last_error_msg();
#  unexpected character
```

It is not safe to rely only on the return value being `NULL` to detect errors. For example, if the JSON string contains nothing but `"null"`, `json_decode()` will return `null`, even though no error occurred.

Section 31.2: Encoding a JSON string

The `json_encode` function will convert a PHP array (or, since PHP 5.4, an object which implements the `JsonSerializable` interface) to a JSON-encoded string. It returns a JSON-encoded string on success or FALSE on failure.

```
$array = [
    'name' => 'Jeff',
    'age' => 20,
    'active' => true,
    'colors' => ['red', 'blue'],
    'values' => [0=>'foo', 3=>'bar'],
];
```

During encoding, the PHP data types string, integer, and boolean are converted to their JSON equivalent. Associative arrays are encoded as JSON objects, and – when called with default arguments – indexed arrays are encoded as JSON arrays. (Unless the array keys are not a continuous numeric sequence starting from 0, in which case the array will be encoded as a JSON object.)

```
echo json_encode($array);
```

Output:

```
{"name":"Jeff","age":20,"active":true,"colors":["red","blue"],"values":{"0":"foo","3":"bar"}}
```

Arguments

Since PHP 5.3, the second argument to `json_encode` is a bitmask which can be one or more of the following.

As with any bitmask, they can be combined with the binary OR operator `|`.

PHP 5.x `Version ≥ 5.3`
`JSON_FORCE_OBJECT`

Forces the creation of an object instead of an array

```
$array = ['Joel', 23, true, ['red', 'blue']];
echo json_encode($array);
echo json_encode($array, JSON_FORCE_OBJECT);
```

Output:

```
["Joel",23,true,["red","blue"]]
{"0":"Joel","1":23,"2":true,"3":{"0":"red","1":"blue"}}
```

`JSON_HEX_TAG`, `JSON_HEX_AMP`, `JSON_HEX_APOS`, `JSON_HEX_QUOT`

Ensures the following conversions during encoding:

Constant	Input	Output
JSON_HEX_TAG	<	\u003C
JSON_HEX_TAG	>	\u003E
JSON_HEX_AMP	&	\u0026
JSON_HEX_APOS	'	\u0027
JSON_HEX_QUOT	"	\u0022

```
$array = ["tag"=>"<>", "amp"=>"&", "apos"=>"'", "quot"=>"\""];
echo json_encode($array);
echo json_encode($array, JSON_HEX_TAG | JSON_HEX_AMP | JSON_HEX_APOS | JSON_HEX_QUOT);
```

Output:

```
{"tag":"<>","amp":"&","apos":"'","quot":"\""}
{"tag":"\u003C\u003E","amp":"\u0026","apos":"\u0027","quot":"\u0022"}
```

PHP 5.x `Version ≥ 5.3`

JSON_NUMERIC_CHECK

Ensures numeric strings are converted to integers.

```
$array = ['23452', 23452];
echo json_encode($array);
echo json_encode($array, JSON_NUMERIC_CHECK);
```

Output:

```
["23452",23452]
[23452,23452]
```

PHP 5.x `Version ≥ 5.4`

JSON_PRETTY_PRINT

Makes the JSON easily readable

```
$array = ['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4];
echo json_encode($array);
echo json_encode($array, JSON_PRETTY_PRINT);
```

Output:

```
{"a":1,"b":2,"c":3,"d":4}
{
    "a": 1,
    "b": 2,
    "c": 3,
    "d": 4
}
```

JSON_UNESCAPED_SLASHES

Includes unescaped `/` forward slashes in the output

```
$array = ['filename' => 'example.txt', 'path' => '/full/path/to/file/'];
echo json_encode($array);
echo json_encode($array, JSON_UNESCAPED_SLASHES);
```

Output:

```
{"filename":"example.txt","path":"\/full\/path\/to\/file"}
{"filename":"example.txt","path":"/full/path/to/file"}
```

JSON_UNESCAPED_UNICODE

Includes UTF8-encoded characters in the output instead of `\u`-encoded strings

```
$blues = ["english"=>"blue", "norwegian"=>"blå", "german"=>"blau"];
echo json_encode($blues);
echo json_encode($blues, JSON_UNESCAPED_UNICODE);
```

Output:

```
{"english":"blue","norwegian":"bl\u00e5","german":"blau"}
{"english":"blue","norwegian":"blå","german":"blau"}
```

PHP 5.x `Version ≥ 5.5`

JSON_PARTIAL_OUTPUT_ON_ERROR

Allows encoding to continue if some unencodable values are encountered.

```
$fp = fopen("foo.txt", "r");
$array = ["file"=>$fp, "name"=>"foo.txt"];
echo json_encode($array); // no output
echo json_encode($array, JSON_PARTIAL_OUTPUT_ON_ERROR);
```

Output:

```
{"file":null,"name":"foo.txt"}
```

PHP 5.x `Version ≥ 5.6`

JSON_PRESERVE_ZERO_FRACTION

Ensures that floats are always encoded as floats.

```
$array = [5.0, 5.5];
echo json_encode($array);
echo json_encode($array, JSON_PRESERVE_ZERO_FRACTION);
```

Output:

```
[5,5.5]
[5.0,5.5]
```

PHP 7.x `Version ≥ 7.1`

JSON_UNESCAPED_LINE_TERMINATORS

When used with `JSON_UNESCAPED_UNICODE`, reverts to the behaviour of older PHP versions, and *does not* escape the characters U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR. Although valid in JSON, these characters are not valid in JavaScript, so the default behaviour of `JSON_UNESCAPED_UNICODE` was changed in version 7.1.

```
$array = ["line"=>"\xe2\x80\xa8", "paragraph"=>"\xe2\x80\xa9"];
echo json_encode($array, JSON_UNESCAPED_UNICODE);
echo json_encode($array, JSON_UNESCAPED_UNICODE | JSON_UNESCAPED_LINE_TERMINATORS);
```

Output:

```
{"line":"\u2028","paragraph":"\u2029"}
```

```
{"line":"","paragraph":""}
```

Section 31.3: Debugging JSON errors

When `json_encode` or `json_decode` fails to parse the string provided, it will return **`false`**. PHP itself will not raise any errors or warnings when this happens, the onus is on the user to use the json_last_error() and json_last_error_msg() functions to check if an error occurred and act accordingly in your application (debug it, show an error message, etc.).

The following example shows a common error when working with JSON, a failure to decode/encode a JSON string *(due to the passing of a bad UTF-8 encoded string, for example)*.

```
// An incorrectly formed JSON string
$jsonString = json_encode("{'Bad JSON':\xB1\x31}");

if (json_last_error() != JSON_ERROR_NONE) {
    printf("JSON Error: %s", json_last_error_msg());
}

#> JSON Error: Malformed UTF-8 characters, possibly incorrectly encoded
```

json_last_error_msg

json_last_error_msg() returns a human readable message of the last error that occurred when trying to encode/decode a string.

- This function will **always return a string**, even if no error occurred.
  The default *non-error* string is `No Error`
- It will return **`false`** if some other (unknown) error occurred
- Careful when using this in loops, as json_last_error_msg will be overridden on each iteration.

You should only use this function to get the message for display, **not** to test against in control statements.

```
// Don't do this:
if (json_last_error_msg()){} // always true (it's a string)
if (json_last_error_msg() != "No Error"){} // Bad practice

// Do this: (test the integer against one of the pre-defined constants)
if (json_last_error() != JSON_ERROR_NONE) {
    // Use json_last_error_msg to display the message only, (not test against it)
    printf("JSON Error: %s", json_last_error_msg());
}
```

This function doesn't exist before PHP 5.5. Here is a polyfill implementation:

```
if (!function_exists('json_last_error_msg')) {
    function json_last_error_msg() {
        static $ERRORS = array(
            JSON_ERROR_NONE => 'No error',
            JSON_ERROR_DEPTH => 'Maximum stack depth exceeded',
            JSON_ERROR_STATE_MISMATCH => 'State mismatch (invalid or malformed JSON)',
            JSON_ERROR_CTRL_CHAR => 'Control character error, possibly incorrectly encoded',
            JSON_ERROR_SYNTAX => 'Syntax error',
            JSON_ERROR_UTF8 => 'Malformed UTF-8 characters, possibly incorrectly encoded'
        );

        $error = json_last_error();
```

```
        return isset($ERRORS[$error]) ? $ERRORS[$error] : 'Unknown error';
    }
}
```

json_last_error

`json_last_error()` returns an **integer** mapped to one of the pre-defined constants provided by PHP.

Constant	Meaning
`JSON_ERROR_NONE`	No error has occurred
`JSON_ERROR_DEPTH`	The maximum stack depth has been exceeded
`JSON_ERROR_STATE_MISMATCH`	Invalid or malformed JSON
`JSON_ERROR_CTRL_CHAR`	Control character error, possibly incorrectly encoded
`JSON_ERROR_SYNTAX`	Syntax error *(since PHP 5.3.3)*
`JSON_ERROR_UTF8`	Malformed UTF-8 characters, possibly incorrectly encoded *(since PHP 5.5.0)*
`JSON_ERROR_RECURSION`	One or more recursive references in the value to be encoded
`JSON_ERROR_INF_OR_NAN`	One or more NAN or INF values in the value to be encoded
`JSON_ERROR_UNSUPPORTED_TYPE`	A value of a type that cannot be encoded was given

Section 31.4: Using JsonSerializable in an Object

PHP 5.x `Version ≥ 5.4`

When you build REST API's, you may need to reduce the information of an object to be passed to the client application. For this purpose, this example illustrates how to use the `JsonSerialiazble` interface.

In this example, the class `User` actually extends a DB model object of a hypotetical ORM.

```
class User extends Model implements JsonSerializable {
    public $id;
    public $name;
    public $surname;
    public $username;
    public $password;
    public $email;
    public $date_created;
    public $date_edit;
    public $role;
    public $status;

    public function jsonSerialize() {
        return [
            'name' => $this->name,
            'surname' => $this->surname,
            'username' => $this->username
        ];
    }
}
```

Add `JsonSerializable` implementation to the class, by providing the `jsonSerialize()` method.

```
public function jsonSerialize()
```

Now in your application controller or script, when passing the object User to `json_encode()` you will get the return json encoded array of the `jsonSerialize()` method instead of the entire object.

```
json_encode($User);
```

Will return:

```
{"name":"John", "surname":"Doe", "username" : "TestJson"}
```

properties values example.

This will both reduce the amount of data returned from a RESTful endpoint, and allow to exclude object properties from a json representation.

Using Private and Protected Properties with `json_encode()`

To avoid using JsonSerializable, it is also possible to use private or protected properties to hide class information from `json_encode()` output. The Class then does not need to implement \JsonSerializable.

> The json_encode() function will only encode public properties of a class into JSON.

```
<?php

class User {
    // private properties only within this class
    private $id;
    private $date_created;
    private $date_edit;

    // properties used in extended classes
    protected $password;
    protected $email;
    protected $role;
    protected $status;

    // share these properties with the end user
    public $name;
    public $surname;
    public $username;

    // jsonSerialize() not needed here
}

$theUser = new User();

var_dump(json_encode($theUser));
```

Output:

```
string(44) "{"name":null,"surname":null,"username":null}"
```

Section 31.5: Header json and the returned response

By adding a header with content type as JSON:

```
<?php
 $result = array('menu1' => 'home', 'menu2' => 'code php', 'menu3' => 'about');
```

```php
//return the json response:
header('Content-Type: application/json');  // <-- header declaration
echo json_encode($result, true);    // <--- encode
exit();
```

The header is there so your app can detect what data was returned and how it should handle it.
Note that: the content header is just information about type of returned data.

If you are using UTF-8, you can use:

```php
header("Content-Type: application/json;charset=utf-8");
```

Example jQuery:

```
$.ajax({
        url:'url_your_page_php_that_return_json'
    }).done(function(data){
        console.table('json ',data);
        console.log('Menu1: ', data.menu1);
    });
```

Chapter 32: SOAP Client

Parameter	Details
$wsdl	URI of WSDL or `NULL` if using non-WSDL mode
$options	Array of options for SoapClient. Non-WSDL mode requires `location` and `uri` to set, all other options are optional. See table below for possible values.

Section 32.1: WSDL Mode

First, create a new `SoapClient` object, passing the URL to the WSDL file and optionally, an array of options.

```
// Create a new client object using a WSDL URL
$soap = new SoapClient('https://example.com/soap.wsdl', [
    # This array and its values are optional
   'soap_version' => SOAP_1_2,
    'compression' => SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_GZIP,
    'cache_wsdl' => WSDL_CACHE_BOTH,
    # Helps with debugging
   'trace' => TRUE,
    'exceptions' => TRUE
]);
```

Then use the `$soap` object to call your SOAP methods.

```
$result = $soap->requestData(['a', 'b', 'c']);
```

Section 32.2: Non-WSDL Mode

This is similar to WSDL mode, except we pass `NULL` as the WSDL file and make sure to set the `location` and `uri` options.

```
$soap = new SoapClient(NULL, [
    'location' => 'https://example.com/soap/endpoint',
    'uri' => 'namespace'
]);
```

Section 32.3: Classmaps

When creating a SOAP Client in PHP, you can also set a `classmap` key in the configuration array. This `classmap` defines which types defined in the WSDL should be mapped to actual classes, instead of the default `StdClass`. The reason you would want to do this is because you can get auto-completion of fields and method calls on these classes, instead of having to guess which fields are set on the regular `StdClass`.

```
class MyAddress {
    public $country;
    public $city;
    public $full_name;
    public $postal_code; // or zip_code
    public $house_number;
}

class MyBook {
    public $name;
    public $author;
```

```
    // The classmap also allows us to add useful functions to the objects
    // that are returned from the SOAP operations.
    public function getShortDescription() {
        return "{$this->name}, written by {$this->author}";
    }
}

$soap_client = new SoapClient($link_to_wsdl, [
    // Other parameters
    "classmap" => [
        "Address" => MyAddress::class, // ::class simple returns class as string
        "Book" => MyBook::class,
    ]
]);
```

After configuring the classmap, whenever you perform a certain operation that returns a type `Address` or `Book`, the SoapClient will instantiate that class, fill the fields with the data and return it from the operation call.

```
// Lets assume 'getAddress(1234)' returns an Address by ID in the database
$address = $soap_client->getAddress(1234);

// $address is now of type MyAddress due to the classmap
echo $address->country;

// Lets assume the same for 'getBook(1234)'
$book = $soap_client->getBook(124);

// We can not use other functions defined on the MyBook class
echo $book->getShortDescription();

// Any type defined in the WSDL that is not defined in the classmap
// will become a regular StdClass object
$author = $soap_client->getAuthor(1234);

// No classmap for Author type, $author is regular StdClass.
// We can still access fields, but no auto-completion and no custom functions
// to define for the objects.
echo $author->name;
```

Section 32.4: Tracing SOAP request and response

Sometimes we want to look at what is sent and received in the SOAP request. The following methods will return the XML in the request and response:

```
SoapClient::__getLastRequest()
SoapClient::__getLastRequestHeaders()
SoapClient::__getLastResponse()
SoapClient::__getLastResponseHeaders()
```

For example, suppose we have an `ENVIRONMENT` constant and when this constant's value is set to `DEVELOPMENT` we want to echo all information when the call to `getAddress` throws an error. One solution could be:

```
try {
    $address = $soap_client->getAddress(1234);
} catch (SoapFault $e) {
    if (ENVIRONMENT === 'DEVELOPMENT') {
        var_dump(
            $soap_client->__getLastRequestHeaders()
            $soap_client->__getLastRequest(),
```

```
            $soap_client->__getLastResponseHeaders(),
            $soap_client->__getLastResponse()
        );
    }
    ...
}
```

Chapter 33: Using cURL in PHP

Parameter	Details
curl_init	-- Initialize a cURL session
url	The url to be used in the cURL request
curl_setopt	-- Set an option for a cURL transfer
ch	The cURL handle (return value from **curl_init()**)
option	CURLOPT_XXX to be set - see PHP documentation for the list of options and acceptable values
value	The value to be set on the cURL handle for the given option
curl_exec	-- Perform a cURL session
ch	The cURL handle (return value from **curl_init()**)
curl_close	-- Close a cURL session
ch	The cURL handle (return value from **curl_init()**)

Section 33.1: Basic Usage (GET Requests)

cURL is a tool for transferring data with URL syntax. It support HTTP, FTP, SCP and many others(curl >= 7.19.4).
Remember, you need to install and enable the cURL extension to use it.

```
// a little script check is the cURL extension loaded or not
if(!extension_loaded("curl")) {
    die("cURL extension not loaded! Quit Now.");
}

// Actual script start

// create a new cURL resource
// $curl is the handle of the resource
$curl = curl_init();

// set the URL and other options
curl_setopt($curl, CURLOPT_URL, "http://www.example.com");

// execute and pass the result to browser
curl_exec($curl);

// close the cURL resource
curl_close($curl);
```

Section 33.2: POST Requests

If you want to mimic HTML form POST action, you can use cURL.

```
// POST data in array
$post = [
    'a' => 'apple',
    'b' => 'banana'
];

// Create a new cURL resource with URL to POST
$ch = curl_init('http://www.example.com');

// We set parameter CURLOPT_RETURNTRANSFER to read output
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
```

```php
// Let's pass POST data
curl_setopt($ch, CURLOPT_POSTFIELDS, $post);

// We execute our request, and get output in a $response variable
$response = curl_exec($ch);

// Close the connection
curl_close($ch);
```

Section 33.3: Using Cookies

cURL can keep cookies received in responses for use with subsequent requests. For simple session cookie handling in memory, this is achieved with a single line of code:

```php
curl_setopt($ch, CURLOPT_COOKIEFILE, "");
```

In cases where you are required to keep cookies after the cURL handle is destroyed, you can specify the file to store them in:

```php
curl_setopt($ch, CURLOPT_COOKIEJAR, "/tmp/cookies.txt");
```

Then, when you want to use them again, pass them as the cookie file:

```php
curl_setopt($ch, CURLOPT_COOKIEFILE, "/tmp/cookies.txt");
```

Remember, though, that these two steps are not necessary unless you need to carry cookies between different cURL handles. For most use cases, setting `CURLOPT_COOKIEFILE` to the empty string is all you need.

Cookie handling can be used, for example, to retrieve resources from a web site that requires a login. This is typically a two-step procedure. First, POST to the login page.

```php
<?php

# create a cURL handle
$ch  = curl_init();

# set the URL (this could also be passed to curl_init() if desired)
curl_setopt($ch, CURLOPT_URL, "https://www.example.com/login.php");

# set the HTTP method to POST
curl_setopt($ch, CURLOPT_POST, true);

# setting this option to an empty string enables cookie handling
# but does not load cookies from a file
curl_setopt($ch, CURLOPT_COOKIEFILE, "");

# set the values to be sent
curl_setopt($ch, CURLOPT_POSTFIELDS, array(
    "username"=>"joe_bloggs",
    "password"=>"$up3r_$3cr3t",
));

# return the response body
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

# send the request
$result = curl_exec($ch);
```

The second step (after standard error checking is done) is usually a simple GET request. The important thing is to **reuse the existing cURL handle** for the second request. This ensures the cookies from the first response will be automatically included in the second request.

```
# we are not calling curl_init()

# simply change the URL
curl_setopt($ch, CURLOPT_URL, "https://www.example.com/show_me_the_foo.php");

# change the method back to GET
curl_setopt($ch, CURLOPT_HTTPGET, true);

# send the request
$result = curl_exec($ch);

# finished with cURL
curl_close($ch);

# do stuff with $result...
```

This is only intended as an example of cookie handling. In real life, things are usually more complicated. Often you must perform an initial GET of the login page to pull a login token that needs to be included in your POST. Other sites might block the cURL client based on its User-Agent string, requiring you to change it.

Section 33.4: Using multi_curl to make multiple POST requests

Sometimes we need to make a lot of POST requests to one or many different endpoints. To deal with this scenario, we can use `multi_curl`.

First of all, we create how many requests as needed exactly in the same way of the simple example and put them in an array.

We use the curl_multi_init and add each handle to it.

In this example, we are using 2 different endpoints:

```
//array of data to POST
$request_contents = array();
//array of URLs
$urls = array();
//array of cURL handles
$chs = array();

//first POST content
$request_contents[] = [
    'a' => 'apple',
    'b' => 'banana'
];
//second POST content
$request_contents[] = [
    'a' => 'fish',
    'b' => 'shrimp'
];
//set the urls
$urls[] = 'http://www.example.com';
$urls[] = 'http://www.example2.com';
```

```php
//create the array of cURL handles and add to a multi_curl
$mh = curl_multi_init();
foreach ($urls as $key => $url) {
    $chs[$key] = curl_init($url);
    curl_setopt($chs[$key], CURLOPT_RETURNTRANSFER, true);
    curl_setopt($chs[$key], CURLOPT_POST, true);
    curl_setopt($chs[$key], CURLOPT_POSTFIELDS, $request_contents[$key]);

    curl_multi_add_handle($mh, $chs[$key]);
}
```

Then, we use curl_multi_exec to send the requests

```php
//running the requests
$running = null;
do {
  curl_multi_exec($mh, $running);
} while ($running);

//getting the responses
foreach(array_keys($chs) as $key){
    $error = curl_error($chs[$key]);
    $last_effective_URL = curl_getinfo($chs[$key], CURLINFO_EFFECTIVE_URL);
    $time = curl_getinfo($chs[$key], CURLINFO_TOTAL_TIME);
    $response = curl_multi_getcontent($chs[$key]);  // get results
    if (!empty($error)) {
      echo "The request $key return a error: $error" . "\n";
    }
    else {
      echo "The request to '$last_effective_URL' returned '$response' in $time seconds." . "\n";
    }

    curl_multi_remove_handle($mh, $chs[$key]);
}

// close current handler
curl_multi_close($mh);
```

A possible return for this example could be:

> The request to 'http://www.example.com' returned 'fruits' in 2 seconds.
>
> The request to 'http://www.example2.com' returned 'seafood' in 5 seconds.

Section 33.5: Sending multi-dimensional data and multiple files with CurlFile in one request

Let's say we have a form like the one below. We want to send the data to our webserver via AJAX and from there to a script running on an external server.

So we have normal inputs, a multi-select field and a file dropzone where we can upload multiple files.

Assuming the AJAX POST request was successful we get the following data on PHP site:

```
// print_r($_POST)

Array
(
    [first_name] => John
    [last_name] => Doe
    [activities] => Array
        (
            [0] => soccer
            [1] => hiking
        )
)
```

and the files should look like this

```
// print_r($_FILES)

Array
(
    [upload] => Array
        (
            [name] => Array
                (
                    [0] => my_photo.jpg
                    [1] => my_life.pdf
                )
```

```
            [type] => Array
                (
                    [0] => image/jpg
                    [1] => application/pdf
                )

            [tmp_name] => Array
                (
                    [0] => /tmp/phpW5spji
                    [1] => /tmp/phpWgnUeY
                )

            [error] => Array
                (
                    [0] => 0
                    [1] => 0
                )

            [size] => Array
                (
                    [0] => 647548
                    [1] => 643223
                )

        )

)
```

So far, so good. Now we want to send this data and files to the external server using cURL with the CurlFile Class

Since cURL only accepts a simple but not a multi-dimensional array, we have to flatten the $_POST array first.

To do this, you could use this function for example which gives you the following:

```
// print_r($new_post_array)

Array
(
    [first_name] => John
    [last_name] => Doe
    [activities[0]] => soccer
    [activities[1]] => hiking
)
```

The next step is to create CurlFile Objects for the uploaded files. This is done by the following loop:

```
$files = array();

foreach ($_FILES["upload"]["error"] as $key => $error) {
    if ($error == UPLOAD_ERR_OK) {

        $files["upload[$key]"] = curl_file_create(
            $_FILES['upload']['tmp_name'][$key],
            $_FILES['upload']['type'][$key],
            $_FILES['upload']['name'][$key]
        );
    }
}
```

curl_file_create is a helper function of the CurlFile Class and creates the CurlFile objects. We save each object in the

$files array with keys named "upload[0]" and "upload[1]" for our two files.

We now have to combine the flattened post array and the files array and save it as $data like this:

```
$data = $new_post_array + $files;
```

The last step is to send the cURL request:

```
$ch = curl_init();

curl_setopt_array($ch, array(
    CURLOPT_POST => 1,
    CURLOPT_URL => "https://api.externalserver.com/upload.php",
    CURLOPT_RETURNTRANSFER => 1,
    CURLINFO_HEADER_OUT => 1,
    CURLOPT_POSTFIELDS => $data
));

$result = curl_exec($ch);

curl_close ($ch);
```

Since $data is now a simple (flat) array, cURL automatically sends this POST request with Content Type: multipart/form-data

In upload.php on the external server you can now get the post data and files with $_POST and $_FILES as you would normally do.

Section 33.6: Creating and sending a request with a custom method

By default, PHP Curl supports `GET` and `POST` requests. It is possible to also send custom requests, such as `DELETE`, `PUT` or `PATCH` (or even non-standard methods) using the `CURLOPT_CUSTOMREQUEST` parameter.

```
$method = 'DELETE'; // Create a DELETE request

$ch = curl_init($url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, $method);
$content = curl_exec($ch);
curl_close($ch);
```

Section 33.7: Get and Set custom http headers in php

Sending The Request Header

```
$uri = 'http://localhost/http.php';
$ch = curl_init($uri);
curl_setopt_array($ch, array(
    CURLOPT_HTTPHEADER  => array('X-User: admin', 'X-Authorization: 123456'),
    CURLOPT_RETURNTRANSFER  =>true,
    CURLOPT_VERBOSE     => 1
));
$out = curl_exec($ch);
curl_close($ch);
// echo response output
echo $out;
```

Reading the custom header

```
print_r(apache_request_headers());
```

Output:

```
Array
(
    [Host] => localhost
    [Accept] => */*
    [X-User] => admin
    [X-Authorization] => 123456
    [Content-Length] => 9
    [Content-Type] => application/x-www-form-urlencoded
)
```

We can also send the header using below syntax:

```
curl --header "X-MyHeader: 123" www.google.com
```

Chapter 34: Reflection

Section 34.1: Feature detection of classes or objects

Feature detection of classes can partly be done with the `property_exists` and `method_exists` functions.

```
class MyClass {
    public $public_field;
    protected $protected_field;
    private $private_field;
    static $static_field;
    const CONSTANT = 0;
    public function public_function() {}
    protected function protected_function() {}
    private function private_function() {}
    static function static_function() {}
}

// check properties
$check = property_exists('MyClass', 'public_field');    // true
$check = property_exists('MyClass', 'protected_field'); // true
$check = property_exists('MyClass', 'private_field');   // true, as of PHP 5.3.0
$check = property_exists('MyClass', 'static_field');    // true
$check = property_exists('MyClass', 'other_field');     // false

// check methods
$check = method_exists('MyClass', 'public_function');    // true
$check = method_exists('MyClass', 'protected_function');    // true
$check = method_exists('MyClass', 'private_function');    // true
$check = method_exists('MyClass', 'static_function');    // true

// however...
$check = property_exists('MyClass', 'CONSTANT');  // false
$check = property_exists($object, 'CONSTANT');    // false
```

With a `ReflectionClass`, also constants can be detected:

```
$r = new ReflectionClass('MyClass');
$check = $r->hasProperty('public_field');  // true
$check = $r->hasMethod('public_function'); // true
$check = $r->hasConstant('CONSTANT');      // true
// also works for protected, private and/or static members.
```

Note: for `property_exists` and `method_exists`, also an object of the class of interest can be provided instead of the class name. Using reflection, the `ReflectionObject` class should be used instead of `ReflectionClass`.

Section 34.2: Testing private/protected methods

Sometimes it's useful to test private & protected methods as well as public ones.

```
class Car
{
    /**
     * @param mixed $argument
     *
     * @return mixed
     */
    protected function drive($argument)
```

```
    {
        return $argument;
    }

    /**
     * @return bool
     */
    private static function stop()
    {
        return true;
    }
}
```

Easiest way to test drive method is using reflection

```
class DriveTest
{
    /**
     * @test
     */
    public function testDrive()
    {
        // prepare
        $argument = 1;
        $expected = $argument;
        $car = new \Car();

        $reflection = new ReflectionClass(\Car::class);
        $method = $reflection->getMethod('drive');
        $method->setAccessible(true);

        // invoke logic
        $result = $method->invokeArgs($car, [$argument]);

        // test
        $this->assertEquals($expected, $result);
    }
}
```

If the method is static you pass null in the place of the class instance

```
class StopTest
{
    /**
     * @test
     */
    public function testStop()
    {
        // prepare
        $expected = true;

        $reflection = new ReflectionClass(\Car::class);
        $method = $reflection->getMethod('stop');
        $method->setAccessible(true);

        // invoke logic
        $result = $method->invoke(null);

        // test
        $this->assertEquals($expected, $result);
    }
```

```
}
```

Section 34.3: Accessing private and protected member variables

Reflection is often used as part of software testing, such as for the runtime creation/instantiation of mock objects. It's also great for inspecting the state of an object at any given point in time. Here's an example of using Reflection in a unit test to verify a protected class member contains the expected value.

Below is a very basic class for a Car. It has a protected member variable that will contain the value representing the color of the car. Because the member variable is protected we cannot access it directly and must use a getter and setter method to retrieve and set its value respectively.

```
class Car
{
    protected $color

    public function setColor($color)
    {
        $this->color = $color;
    }

    public function getColor($color)
    {
        return $this->color;
    }
}
```

To test this many developers will create a Car object, set the car's color using `Car::setColor()`, retrieve the color using `Car::getColor()`, and compare that value to the color they set:

```
/**
 * @test
 * @covers     \Car::setColor
 */
public function testSetColor()
{
    $color = 'Red';

    $car = new \Car();
    $car->setColor($color);
    $getColor = $car->getColor();

    $this->assertEquals($color, $reflectionColor);
}
```

On the surface this seems okay. After all, all `Car::getColor()` does is return the value of the protected member variable `Car::$color`. But this test is flawed in two ways:

1. It exercises `Car::getColor()` which is out of the scope of this test
2. It depends on `Car::getColor()` which may have a bug itself which can make the test have a false positive or negative

Let's look at why we shouldn't use `Car::getColor()` in our unit test and should use Reflection instead. Let's say a developer is assigned a task to add "Metallic" to every car color. So they attempt to modify the `Car::getColor()` to prepend "Metallic" to the car's color:

```
class Car
{
    protected $color

    public function setColor($color)
    {
        $this->color = $color;
    }

    public function getColor($color)
    {
        return "Metallic "; $this->color;
    }
}
```

Do you see the error? The developer used a semi-colon instead of the concatenation operator in an attempt to prepend "Metallic" to the car's color. As a result, whenever `Car::getColor()` is called, "Metallic " will be returned regardless of what the car's actual color is. As a result our `Car::setColor()` unit test will fail *even though `Car::setColor()` works perfectly fine and was not affected by this change*.

So how do we verify `Car::$color` contains the value we are setting via `Car::setColor()`? We can use Refelection to inspect the protected member variable directly. So how do we do *that*? We can use Refelection to make the protected member variable accessible to our code so it can retrieve the value.

Let's see the code first and then break it down:

```
/**
 * @test
 * @covers     \Car::setColor
 */
public function testSetColor()
{
    $color = 'Red';

    $car = new \Car();
    $car->setColor($color);

    $reflectionOfCar = new \ReflectionObject($car);
    $protectedColor = $reflectionOfForm->getProperty('color');
    $protectedColor->setAccessible(true);
    $reflectionColor = $protectedColor->getValue($car);

    $this->assertEquals($color, $reflectionColor);
}
```

Here is how we are using Reflection to get the value of `Car::$color` in the code above:

1. We create a new ReflectionObject representing our Car object
2. We get a ReflectionProperty for `Car::$color` (this "represents" the `Car::$color` variable)
3. We make `Car::$color` accessible
4. We get the value of `Car::$color`

As you can see by using Reflection we could get the value of `Car::$color` without having to call `Car::getColor()` or any other accessor function which could cause invalid test results. Now our unit test for `Car::setColor()` is safe and accurate.

Chapter 35: Dependency Injection

Dependency Injection (DI) is a fancy term for *"passing things in"*. All it really means is passing the dependencies of an object via the constructor and / or setters instead of creating them upon object creation inside the object. Dependency Injection might also refer to Dependency Injection Containers which automate the construction and injection.

Section 35.1: Constructor Injection

Objects will often depend on other objects. Instead of creating the dependency in the constructor, the dependency should be passed into the constructor as a parameter. This ensures there is not tight coupling between the objects, and enables changing the dependency upon class instantiation. This has a number of benefits, including making code easier to read by making the dependencies explicit, as well as making testing simpler since the dependencies can be switched out and mocked more easily.

In the following example, `Component` will depend on an instance of `Logger`, but it doesn't create one. It requires one to be passed as argument to the constructor instead.

```
interface Logger {
    public function log(string $message);
}

class Component {
    private $logger;

    public function __construct(Logger $logger) {
        $this->logger = $logger;
    }
}
```

Without dependency injection, the code would probably look similar to:

```
class Component {
    private $logger;

    public function __construct() {
        $this->logger = new FooLogger();
    }
}
```

Using **`new`** to create new objects in the constructor indicates that dependency injection was not used (or was used incompletely), and that the code is tightly coupled. It is also a sign that the code is incompletely tested or may have brittle tests that make incorrect assumptions about program state.

In the above example, where we are using dependency injection instead, we could easily change to a different Logger if doing so became necessary. For example, we might use a Logger implementation that logs to a different location, or that uses a different logging format, or that logs to the database instead of to a file.

Section 35.2: Setter Injection

Dependencies can also be injected by setters.

```
interface Logger {
    public function log($message);
}
```

```
class Component {
    private $logger;
    private $databaseConnection;

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

    public function setLogger(Logger $logger) {
        $this->logger = $logger;
    }

    public function core() {
        $this->logSave();
        return $this->databaseConnection->save($this);
    }

    public function logSave() {
         if ($this->logger) {
            $this->logger->log('saving');
        }
    }
}
```

This is especially interesting when the core functionality of the class does not rely on the dependency to work.

Here, the **only** needed dependency is the `DatabaseConnection` so it's in the constructor. The `Logger` dependency is optional and thus does not need to be part of the constructor, making the class easier to use.

Note that when using setter injection, it's better to extend the functionality rather than replacing it. When setting a dependency, there's nothing confirming that the dependency won't change at some point, which could lead in unexpected results. For example, a `FileLogger` could be set at first, and then a `MailLogger` could be set. This breaks encapsulation and makes logs hard to find, because we're **replacing** the dependency.

To prevent this, we should **add** a dependency with setter injection, like so:

```
interface Logger {
    public function log($message);
}

class Component {
    private $loggers = array();
    private $databaseConnection;

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

    public function addLogger(Logger $logger) {
        $this->loggers[] = $logger;
    }

    public function core() {
        $this->logSave();
        return $this->databaseConnection->save($this);
    }

    public function logSave() {
        foreach ($this->loggers as $logger) {
            $logger->log('saving');
```

```
        }
    }
}
```

Like this, whenever we'll use the core functionality, it won't break even if there is no logger dependency added, and any logger added will be used even though another logger could've been added. We're **extending** functionality instead of **replacing** it.

Section 35.3: Container Injection

Dependency Injection (DI) in the context of using a Dependency Injection Container (DIC) can be seen as a superset of constructor injection. A DIC will typically analyze a class constructor's typehints and resolve its needs, effectively injecting the dependencies needed for the instance execution.

The exact implementation goes well beyond the scope of this document but at its very heart, a DIC relies on using the signature of a class...

```
namespace Documentation;

class Example
{
    private $meaning;

    public function __construct(Meaning $meaning)
    {
        $this->meaning = $meaning;
    }
}
```

... to automatically instantiate it, relying most of the time on an autoloading system.

```
// older PHP versions
$container->make('Documentation\Example');

// since PHP 5.5
$container->make(\Documentation\Example::class);
```

If you are using PHP in version at least 5.5 and want to get a name of a class in a way that's being shown above, the correct way is the second approach. That way you can quickly find usages of the class using modern IDEs, which will greatly help you with potential refactoring. You do not want to rely on regular strings.

In this case, the `Documentation\Example` knows it needs a `Meaning`, and a DIC would in turn instantiate a `Meaning` type. The concrete implementation need not depend on the consuming instance.

Instead, we set rules in the container, prior to object creation, that instructs how specific types should be instantiated if need be.

This has a number of advantages, as a DIC can

- Share common instances
- Provide a factory to resolve a type signature
- Resolve an interface signature

If we define rules about how specific type needs to be managed we can achieve fine control over which types are shared, instantiated, or created from a factory.

Chapter 36: XML

Section 36.1: Create a XML using DomDocument

To create a XML using DOMDocument,basically, we need to create all the tags and attributes using the `createElement()` and `createAttribute()` methods and them create the XML structure with the `appendChild()`.

The example below includes tags, attributes, a CDATA section and a different namespace for the second tag:

```
$dom = new DOMDocument('1.0', 'utf-8');
$dom->preserveWhiteSpace = false;
$dom->formatOutput = true;

//create the main tags, without values
$books = $dom->createElement('books');
$book_1 = $dom->createElement('book');

// create some tags with values
$name_1 = $dom->createElement('name', 'PHP - An Introduction');
$price_1 = $dom->createElement('price', '$5.95');
$id_1 = $dom->createElement('id', '1');

//create  and append an attribute
$attr_1 = $dom->createAttribute('version');
$attr_1->value = '1.0';
//append the attribute
$id_1->appendChild($attr_1);

//create the second tag book with different namespace
$namespace = 'www.example.com/libraryns/1.0';

//include the namespace prefix in the books tag
$books->setAttributeNS('http://www.w3.org/2000/xmlns/', 'xmlns:ns', $namespace);
$book_2 = $dom->createElementNS($namespace,'ns:book');
$name_2 = $dom->createElementNS($namespace, 'ns:name');

//create a CDATA section (that is another DOMNode instance) and put it inside the name tag
$name_cdata = $dom->createCDATASection('PHP - Advanced');
$name_2->appendChild($name_cdata);
$price_2 = $dom->createElementNS($namespace, 'ns:price', '$25.00');
$id_2 = $dom->createElementNS($namespace, 'ns:id', '2');

//create the XML structure
$books->appendChild($book_1);
$book_1->appendChild($name_1);
$book_1->appendChild($price_1);
$book_1->appendChild($id_1);
$books->appendChild($book_2);
$book_2->appendChild($name_2);
$book_2->appendChild($price_2);
$book_2->appendChild($id_2);

$dom->appendChild($books);

//saveXML() method returns the XML in a String
print_r ($dom->saveXML());
```

This will output the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<books xmlns:ns="www.example.com/libraryns/1.0">
  <book>
    <name>PHP - An Introduction</name>
    <price>$5.95</price>
    <id version="1.0">1</id>
  </book>
  <ns:book>
    <ns:name><![CDATA[PHP - Advanced]]></ns:name>
    <ns:price>$25.00</ns:price>
    <ns:id>2</ns:id>
  </ns:book>
</books>
```

Section 36.2: Read a XML document with DOMDocument

Similarly to the SimpleXML, you can use DOMDocument to parse XML from a string or from a XML file

1. From a string

```
$doc = new DOMDocument();
$doc->loadXML($string);
```

2. From a file

```
$doc = new DOMDocument();
$doc->load('books.xml');// use the actual file path. Absolute or relative
```

Example of parsing

Considering the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
   <book>
      <name>PHP - An Introduction</name>
      <price>$5.95</price>
      <id>1</id>
   </book>
   <book>
      <name>PHP - Advanced</name>
      <price>$25.00</price>
      <id>2</id>
   </book>
</books>
```

This is a example code to parse it

```
$books = $doc->getElementsByTagName('book');
foreach ($books as $book) {
    $title = $book->getElementsByTagName('name')->item(0)->nodeValue;
    $price = $book->getElementsByTagName('price')->item(0)->nodeValue;
    $id = $book->getElementsByTagName('id')->item(0)->nodeValue;
    print_r ("The title of the book $id is $title and it costs $price." . "\n");
}
```

This will output:

> The title of the book 1 is PHP - An Introduction and it costs $5.95.
>
> The title of the book 2 is PHP - Advanced and it costs $25.00.

Section 36.3: Leveraging XML with PHP's SimpleXML Library

SimpleXML is a powerful library which converts XML strings to an easy to use PHP object.

The following assumes an XML structure as below.

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
    <book>
        <bookName>StackOverflow SimpleXML Example</bookName>
        <bookAuthor>PHP Programmer</bookAuthor>
    </book>
    <book>
        <bookName>Another SimpleXML Example</bookName>
        <bookAuthor>Stack Overflow Community</bookAuthor>
        <bookAuthor>PHP Programmer</bookAuthor>
        <bookAuthor>FooBar</bookAuthor>
    </book>
</document>
```

Read our data in to SimpleXML

To get started, we need to read our data into SimpleXML. We can do this in 3 different ways. Firstly, we can load our data from a DOM node.

```
$xmlElement = simplexml_import_dom($domNode);
```

Our next option is to load our data from an XML file.

```
$xmlElement = simplexml_load_file($filename);
```

Lastly, we can load our data from a variable.

```
$xmlString = '<?xml version="1.0" encoding="UTF-8"?>
<document>
    <book>
        <bookName>StackOverflow SimpleXML Example</bookName>
        <bookAuthor>PHP Programmer</bookAuthor>
    </book>
    <book>
        <bookName>Another SimpleXML Example</bookName>
        <bookAuthor>Stack Overflow Community</bookAuthor>
        <bookAuthor>PHP Programmer</bookAuthor>
        <bookAuthor>FooBar</bookAuthor>
    </book>
</document>';
$xmlElement = simplexml_load_string($xmlString);
```

Whether you've picked to load from a DOM Element, from a file or from a string, you are now left with a SimpleXMLElement variable called `$xmlElement`. Now, we can start to make use of our XML in PHP.

Accessing our SimpleXML Data

The simplest way to access data in our SimpleXMLElement object is to call the properties directly. If we want to access our first bookName, `StackOverflow SimpleXML Example`, then we can access it as per below.

```
echo $xmlElement->book->bookName;
```

At this point, SimpleXML will assume that because we have not told it explicitly which book we want, that we want the first one. However, if we decide that we do not want the first one, rather that we want `Another SimpleXML Example`, then we can access it as per below.

```
echo $xmlElement->book[1]->bookName;
```

It is worth noting that using `[0]` works the same as not using it, so

```
$xmlElement->book
```

works the same as

```
$xmlElement->book[0]
```

Looping through our XML

There are many reasons you may wish to loop through XML, such as that you have a number of items, books in our case, that we would like to display on a webpage. For this, we can use a foreach loop or a standard for loop, taking advantage of SimpleXMLElement's count function..

```
foreach ( $xmlElement->book as $thisBook ) {
    echo $thisBook->bookName
}
```

or

```
$count = $xmlElement->count();
for ( $i=0; $i<$count; $i++ ) {
    echo $xmlElement->book[$i]->bookName;
}
```

Handling Errors

Now we have come so far, it is important to realise that we are only humans, and will likely encounter an error eventually - especially if we are playing with different XML files all the time. And so, we will want to handle those errors.

Consider we created an XML file. You will notice that while this XML is much alike what we had earlier, the problem with this XML file is that the final closing tag is /doc instead of /document.

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
    <book>
        <bookName>StackOverflow SimpleXML Example</bookName>
        <bookAuthor>PHP Programmer</bookAuthor>
    </book>
    <book>
        <bookName>Another SimpleXML Example</bookName>
        <bookAuthor>Stack Overflow Community</bookAuthor>
        <bookAuthor>PHP Programmer</bookAuthor>
        <bookAuthor>FooBar</bookAuthor>
```

```
    </book>
</doc>
```

Now, say, we load this into our PHP as $file.

```
libxml_use_internal_errors(true);
$xmlElement = simplexml_load_file($file);
if ( $xmlElement === false ) {
    $errors = libxml_get_errors();
    foreach ( $errors as $thisError ) {
        switch ( $thisError->level ) {
            case LIBXML_ERR_FATAL:
                echo "FATAL ERROR: ";
                break;
            case LIBXML_ERR_ERROR:
                echo "Non Fatal Error: ";
                break;
            case LIBXML_ERR_WARNING:
                echo "Warning: ";
                break;
        }
        echo $thisError->code . PHP_EOL .
            'Message: ' . $thisError->message . PHP_EOL .
            'Line: ' . $thisError->line . PHP_EOL .
            'Column: ' . $thisError->column . PHP_EOL .
            'File: ' . $thisError->file;
    }
    libxml_clear_errors();
} else {
    echo 'Happy Days';
}
```

We will be greeted with the following

```
FATAL ERROR: 76
Message: Opening and ending tag mismatch: document line 2 and doc

Line: 13
Column: 10
File: filepath/filename.xml
```

However as soon as we fix this problem, we are presented with "Happy Days".

Section 36.4: Create an XML file using XMLWriter

Instantiate a XMLWriter object:

```
$xml = new XMLWriter();
```

Next open the file to which you want to write. For example, to write to `/var/www/example.com/xml/output.xml`, use:

```
$xml->openUri('file:///var/www/example.com/xml/output.xml');
```

To start the document (create the XML open tag):

```
$xml->startDocument('1.0', 'utf-8');
```

This will output:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Now you can start writing elements:

```
$xml->writeElement('foo', 'bar');
```

This will generate the XML:

```
<foo>bar</foo>
```

If you need something a little more complex than simply nodes with plain values, you can also "start" an element and add attributes to it before closing it:

```
$xml->startElement('foo');
$xml->writeAttribute('bar', 'baz');
$xml->writeCdata('Lorem ipsum');
$xml->endElement();
```

This will output:

```
<foo bar="baz"><![CDATA[Lorem ipsum]]></foo>
```

Section 36.5: Read a XML document with SimpleXML

You can parse XML from a string or from a XML file

1. From a string

```
$xml_obj = simplexml_load_string($string);
```

2. From a file

```
$xml_obj = simplexml_load_file('books.xml');
```

Example of parsing

Considering the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
   <book>
      <name>PHP - An Introduction</name>
      <price>$5.95</price>
      <id>1</id>
   </book>
   <book>
      <name>PHP - Advanced</name>
      <price>$25.00</price>
      <id>2</id>
   </book>
</books>
```

This is a example code to parse it

```php
$xml = simplexml_load_string($xml_string);
$books = $xml->book;
foreach ($books as $book) {
    $id = $book->id;
    $title = $book->name;
    $price = $book->price;
    print_r ("The title of the book $id is $title and it costs $price." . "\n");
}
```

This will output:

> The title of the book 1 is PHP - An Introduction and it costs $5.95.
> The title of the book 2 is PHP - Advanced and it costs $25.00.

Chapter 37: SimpleXML

Section 37.1: Loading XML data into simplexml

Loading from string

Use `simplexml_load_string` to create a `SimpleXMLElement` from a string:

```
$xmlString = "<?xml version='1.0' encoding='UTF-8'?>";
$xml = simplexml_load_string($xmlString) or die("Error: Cannot create object");
```

Note that `or` not `||` must be used here because the precedence of `or` is higher than `=`. The code after `or` will only be executed if `$xml` finally resolves to false.

Loading from file

Use `simplexml_load_file` to load XML data from a file or a URL:

```
$xml = simplexml_load_string("filePath.xml");

$xml = simplexml_load_string("https://example.com/doc.xml");
```

The URL can be of any schemes that PHP supports, or custom stream wrappers.

Chapter 38: Parsing HTML

Section 38.1: Parsing HTML from a string

PHP implements a DOM Level 2 compliant parser, allowing you to work with HTML using familiar methods like `getElementById()` or `appendChild()`.

```
$html = '<html><body><span id="text">Hello, World!</span></body></html>';

$doc = new DOMDocument();
libxml_use_internal_errors(true);
$doc->loadHTML($html);

echo $doc->getElementById("text")->textContent;
```

Outputs:

```
Hello, World!
```

Note that PHP will emit warnings about any problems with the HTML, especially if you are importing a document fragment. To avoid these warnings, tell the DOM library (libxml) to handle its own errors by calling `libxml_use_internal_errors()` before importing your HTML. You can then use `libxml_get_errors()` to handle errors if needed.

Section 38.2: Using XPath

```
$html = '<html><body><span class="text">Hello, World!</span></body></html>';

$doc = new DOMDocument();
$doc->loadHTML($html);

$xpath = new DOMXPath($doc);
$span = $xpath->query("//span[@class='text']")->item(0);

echo $span->textContent;
```

Outputs:

```
Hello, World!
```

Section 38.3: SimpleXML

Presentation

- SimpleXML is a PHP library which provides an easy way to work with XML documents (especially reading and iterating through XML data).
- The only restraint is that the XML document must be well-formed.

Parsing XML using procedural approach

```
// Load an XML string
$xmlstr = file_get_contents('library.xml');
```

```php
$library = simplexml_load_string($xmlstr);

// Load an XML file
$library = simplexml_load_file('library.xml');

// You can load a local file path or a valid URL (if allow_url_fopen is set to "On" in php.ini
```

Parsing XML using OOP approach

```php
// $isPathToFile: it informs the constructor that the 1st argument represents the path to a file,
// rather than a string that contains 1the XML data itself.

// Load an XML string
$xmlstr = file_get_contents('library.xml');
$library = new SimpleXMLElement($xmlstr);

// Load an XML file
$library = new SimpleXMLElement('library.xml', NULL, true);

// $isPathToFile: it informs the constructor that the first argument represents the path to a file,
rather than a string that contains 1the XML data itself.
```

Accessing Children and Attributes

- When SimpleXML parses an XML document, it converts all its XML elements, or nodes, to properties of the resulting SimpleXMLElement object
- In addition, it converts XML attributes to an associative array that may be accessed from the property to which they belong.

When you know their names:

```php
$library = new SimpleXMLElement('library.xml', NULL, true);
foreach ($library->book as $book){
    echo $book['isbn'];
    echo $book->title;
    echo $book->author;
    echo $book->publisher;
}
```

- The major drawback of this approach is that it is necessary to know the names of every element and attribute in the XML document.

When you don't know their names (or you don't want to know them):

```php
foreach ($library->children() as $child){
    echo $child->getName();
    // Get attributes of this element
    foreach ($child->attributes() as $attr){
        echo ' ' . $attr->getName() . ': ' . $attr;
    }
    // Get children
    foreach ($child->children() as $subchild){
        echo ' ' . $subchild->getName() . ': ' . $subchild;
    }
}
```

Chapter 39: Regular Expressions (regexp/PCRE)

Parameter	Details
`$pattern`	a string with a regular expression (PCRE pattern)

Section 39.1: Global RegExp match

A *global* RegExp match can be performed using `preg_match_all`. `preg_match_all` returns all matching results in the subject string (in contrast to `preg_match`, which only returns the first one).

The `preg_match_all` function returns the number of matches. Third parameter `$matches` will contain matches in format controlled by flags that can be given in fourth parameter.

If given an array, `$matches` will contain array in similar format you'd get with `preg_match`, except that `preg_match` stops at first match, where `preg_match_all` iterates over the string until the string is wholly consumed and returns result of each iteration in a multidimensional array, which format can be controlled by the flag in fourth argument.

The fourth argument, `$flags`, controls structure of `$matches` array. Default mode is `PREG_PATTERN_ORDER` and possible flags are `PREG_SET_ORDER` and `PREG_PATTERN_ORDER`.

Following code demonstrates usage of `preg_match_all`:

```
$subject = "a1b c2d3e f4g";
$pattern = '/[a-z]([0-9])[a-z]/';

var_dump(preg_match_all($pattern, $subject, $matches, PREG_SET_ORDER)); // int(3)
var_dump($matches);
preg_match_all($pattern, $subject, $matches); // the flag is PREG_PATTERN_ORDER by default
var_dump($matches);
// And for reference, same regexp run through preg_match()
preg_match($pattern, $subject, $matches);
var_dump($matches);
```

The first var_dump from `PREG_SET_ORDER` gives this output:

```
array(3) {
  [0]=>
  array(2) {
    [0]=>
    string(3) "a1b"
    [1]=>
    string(1) "1"
  }
  [1]=>
  array(2) {
    [0]=>
    string(3) "c2d"
    [1]=>
    string(1) "2"
  }
  [2]=>
  array(2) {
    [0]=>
    string(3) "f4g"
    [1]=>
```

```
    string(1) "4"
  }
}
```

`$matches` has three nested arrays. Each array represents one match, which has the same format as the return result of `preg_match`.

The second var_dump (`PREG_PATTERN_ORDER`) gives this output:

```
array(2) {
  [0]=>
  array(3) {
    [0]=>
    string(3) "a1b"
    [1]=>
    string(3) "c2d"
    [2]=>
    string(3) "f4g"
  }
  [1]=>
  array(3) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "4"
  }
}
```

When the same regexp is run through `preg_match`, following array is returned:

```
array(2) {
  [0] =>
  string(3) "a1b"
  [1] =>
  string(1) "1"
}
```

Section 39.2: String matching with regular expressions

`preg_match` checks whether a string matches the regular expression.

```
$string = 'This is a string which contains numbers: 12345';

$isMatched = preg_match('%^[a-zA-Z]+: [0-9]+$%', $string);
var_dump($isMatched); // bool(true)
```

If you pass in a third parameter, it will be populated with the matching data of the regular expression:

```
preg_match('%^([a-zA-Z]+): ([0-9]+)$%', 'This is a string which contains numbers: 12345',
$matches);
// $matches now contains results of the regular expression matches in an array.
echo json_encode($matches); // ["numbers: 12345", "numbers", "12345"]
```

`$matches` contains an array of the whole match then substrings in the regular expression bounded by parentheses, in the order of open parenthesis's offset. That means, if you have `/z(a(b))/` as the regular expression, index 0 contains the whole substring `zab`, index 1 contains the substring bounded by the outer parentheses `ab` and index 2

contains the inner parentheses `b`.

Section 39.3: Split string into array by a regular expression

```
$string = "0| PHP 1| CSS 2| HTML 3| AJAX 4| JSON";

//[0-9]: Any single character in the range 0 to 9
// +   : One or more of 0 to 9
$array = preg_split("/[0-9]+\|/", $string, -1, PREG_SPLIT_NO_EMPTY);
//Or
// []  : Character class
// \d  : Any digit
//  +  : One or more of Any digit
$array = preg_split("/[\d]+\|/", $string, -1, PREG_SPLIT_NO_EMPTY);
```

Output:

```
Array
(
    [0] =>  PHP
    [1] =>  CSS
    [2] =>  HTML
    [3] =>  AJAX
    [4] =>  JSON
)
```

To split a string into a array simply pass the string and a regexp for `preg_split();` to match and search, adding a third parameter (`limit`) allows you to set the number of "matches" to perform, the remaining string will be added to the end of the array.

The fourth parameter is (`flags`) here we use the `PREG_SPLIT_NO_EMPTY` which prevents our array from containing any empty keys / values.

Section 39.4: String replacing with regular expression

```
$string = "a;b;c\nd;e;f";
// $1, $2 and $3 represent the first, second and third capturing groups
echo preg_replace("(^([^;]+);([^;]+);([^;]+)$)m", "$3;$2;$1", $string);
```

Outputs

```
c;b;a
f;e;d
```

Searches for everything between semicolons and reverses the order.

Section 39.5: String replace with callback

`preg_replace_callback` works by sending every matched capturing group to the defined callback and replaces it with the return value of the callback. This allows us to replace strings based on any kind of logic.

```
$subject = "He said 123abc, I said 456efg, then she said 789hij";
$regex = "/\b(\d+)\w+/";

// This function replaces the matched entries conditionally
// depending upon the first character of the capturing group
```

```php
function regex_replace($matches){
    switch($matches[1][0]){
        case '7':
            $replacement = "<b>{$matches[0]}</b>";
            break;
        default:
            $replacement = "<i>{$matches[0]}</i>";
    }
    return $replacement;
}

$replaced_str = preg_replace_callback($regex, "regex_replace", $subject);

print_r($replaced_str);
# He said <i>123abc</i>, I said <i>456efg</i>, then she said <b>789hij</b>
```

Chapter 40: Traits

Section 40.1: What is a Trait?

PHP only allows single inheritance. In other words, a class can only `extend` one other class. But what if you need to include something that doesn't belong in the parent class? Prior to PHP 5.4 you would have to get creative, but in 5.4 Traits were introduced. Traits allow you to basically "copy and paste" a portion of a class into your main class

```
trait Talk {
    /** @var string */
    public $phrase = 'Well Wilbur...';
    public function speak() {
        echo $this->phrase;
    }
}

class MrEd extends Horse {
    use Talk;
    public function __construct() {
        $this->speak();
    }

    public function setPhrase($phrase) {
        $this->phrase = $phrase;
    }
}
```

So here we have `MrEd`, which is already extending `Horse`. But not all horses `Talk`, so we have a Trait for that. Let's note what this is doing

First, we define our Trait. We can use it with autoloading and Namespaces (see also Referencing a class or function in a namespace). Then we include it into our `MrEd` class with the keyword **`use`**.

You'll note that `MrEd` takes to using the `Talk` functions and variables without defining them. Remember what we said about **copy and paste**? These functions and variables are all defined within the class now, as if this class had defined them.

Traits are most closely related to Abstract classes in that you can define variables and functions. You also cannot instantiate a Trait directly (i.e. **`new Trait()`**). Traits cannot force a class to implicitly define a function like an Abstract class or an Interface can. Traits are **only** for explicit definitions (since you can `implement` as many Interfaces as you want, see Interfaces).

When should I use a Trait?

The first thing you should do, when considering a Trait, is to ask yourself this important question

> Can I avoid using a Trait by restructuring my code?

More often than not, the answer is going to be *Yes*. Traits are edge cases caused by single inheritance. The temptation to misuse or overuse Traits can be high. But consider that a Trait introduces another source for your code, which means there's another layer of complexity. In the example here, we're only dealing with 3 classes. But Traits mean you can now be dealing with far more than that. For each Trait, your class becomes that much harder to deal with, since you must now go reference each Trait to find out what it defines (and potentially where a collision happened, see Conflict Resolution). Ideally, you should keep as few Traits in your code as possible.

Section 40.2: Traits to facilitate horizontal code reuse

Let's say we have an interface for logging:

```
interface Logger {
    function log($message);
}
```

Now say we have two concrete implementations of the `Logger` interface: the `FileLogger` and the `ConsoleLogger`.

```
class FileLogger implements Logger {
    public function log($message) {
        // Append log message to some file
    }
}

class ConsoleLogger implements Logger {
    public function log($message) {
        // Log message to the console
    }
}
```

Now if you define some other class `Foo` which you also want to be able to perform logging tasks, you could do something like this:

```
class Foo implements Logger {
    private $logger;

    public function setLogger(Logger $logger) {
        $this->logger = $logger;
    }

    public function log($message) {
        if ($this->logger) {
            $this->logger->log($message);
        }
    }
}
```

`Foo` is now also a `Logger`, but its functionality depends on the `Logger` implementation passed to it via `setLogger()`. If we now want class `Bar` to also have this logging mechanism, we would have to duplicate this piece of logic in the `Bar` class.

Instead of duplicating the code, a trait can be defined:

```
trait LoggableTrait {
    protected $logger;

    public function setLogger(Logger $logger) {
        $this->logger = $logger;
    }

    public function log($message) {
        if ($this->logger) {
            $this->logger->log($message);
        }
    }
}
```

Now that we have defined the logic in a trait, we can use the trait to add the logic to the `Foo` and `Bar` classes:

```
class Foo {
    use LoggableTrait;
}

class Bar {
    use LoggableTrait;
}
```

And, for example, we can use the `Foo` class like this:

```
$foo = new Foo();
$foo->setLogger( new FileLogger() );

//note how we use the trait as a 'proxy' to call the Logger's log method on the Foo instance
$foo->log('my beautiful message');
```

Section 40.3: Conflict Resolution

Trying to use several traits into one class could result in issues involving conflicting methods. You need to resolve such conflicts manually.

For example, let's create this hierarchy:

```
trait MeowTrait {
    public function say() {
        print "Meow \n";
    }
}

trait WoofTrait {
    public function say() {
        print "Woof \n";
    }
}

abstract class UnMuteAnimals {
    abstract function say();
}

class Dog extends UnMuteAnimals {
    use WoofTrait;
}

class Cat extends UnMuteAnimals {
    use MeowTrait;
}
```

Now, let's try to create the following class:

```
class TalkingParrot extends UnMuteAnimals {
    use MeowTrait, WoofTrait;
}
```

The php interpreter will return a fatal error:

> **Fatal error:** Trait method say has not been applied, because there are collisions with other trait methods on TalkingParrot

To resolve this conflict, we could do this:

- use keyword `insteadof` to use the method from one trait instead of method from another trait
- create an alias for the method with a construct like `WoofTrait::say as sayAsDog;`

```
class TalkingParrotV2 extends UnMuteAnimals {
    use MeowTrait, WoofTrait {
        MeowTrait::say insteadof WoofTrait;
        WoofTrait::say as sayAsDog;
    }
}

$talkingParrot = new TalkingParrotV2();
$talkingParrot->say();
$talkingParrot->sayAsDog();
```

This code will produce the following output:

> Meow
> Woof

Section 40.4: Implementing a Singleton using Traits

Disclaimer: *In no way does this example advocate the use of singletons. Singletons are to be used with a lot of care.*

In PHP there is quite a standard way of implementing a singleton:

```
public class Singleton {
    private $instance;

    private function __construct() { };

    public function getInstance() {
        if (!self::$instance) {
            // new self() is 'basically' equivalent to new Singleton()
            self::$instance = new self();
        }

        return self::$instance;
    }

    // Prevent cloning of the instance
    protected function __clone() { }

    // Prevent serialization of the instance
    protected function __sleep() { }

    // Prevent deserialization of the instance
    protected function __wakeup() { }
}
```

To prevent code duplication, it is a good idea to extract this behaviour into a trait.

```php
trait SingletonTrait {
    private $instance;

    protected function __construct() { };

    public function getInstance() {
        if (!self::$instance) {
            // new self() will refer to the class that uses the trait
            self::$instance = new self();
        }

        return self::$instance;
    }

    protected function __clone() { }
    protected function __sleep() { }
    protected function __wakeup() { }
}
```

Now any class that wants to function as a singleton can simply use the trait:

```php
class MyClass {
    use SingletonTrait;
}

// Error! Constructor is not publicly accessible
$myClass = new MyClass();

$myClass = MyClass::getInstance();

// All calls below will fail due to method visibility
$myClassCopy = clone $myClass; // Error!
$serializedMyClass = serialize($myClass); // Error!
$myClass = deserialize($serializedMyclass); // Error!
```

Even though it is now impossible to serialize a singleton, it is still useful to also disallow the deserialize method.

Section 40.5: Traits to keep classes clean

Over time, our classes may implement more and more interfaces. When these interfaces have many methods, the total number of methods in our class will become very large.

For example, let's suppose that we have two interfaces and a class implementing them:

```php
interface Printable {
    public function print();
    //other interface methods...
}

interface Cacheable {
    //interface methods
}

class Article implements Cachable, Printable {
    //here we must implement all the interface methods
    public function print(){ {
        /* code to print the article */
    }
}
```

Instead of implementing all the interface methods inside the `Article` class, we could use separate Traits to implement these interfaces, **keeping the class smaller and separating the code of the interface implementation from the class.**

From example, to implement the `Printable` interface we could create this trait:

```
trait PrintableArticle {
    //implements here the interface methods
    public function print() {
        /* code to print the article */
    }
}
```

and make the class use the trait:

```
class Article implements Cachable, Printable {
    use PrintableArticle;
    use CacheableArticle;
}
```

The primary benefits would be that our interface-implementation methods will be separated from the rest of the class, and stored in a trait who has the **sole responsibility to implement the interface for that particular type of object.**

Section 40.6: Multiple Traits Usage

```
trait Hello {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait World {
    public function sayWorld() {
        echo 'World';
    }
}

class MyHelloWorld {
    use Hello, World;
    public function sayExclamationMark() {
        echo '!';
    }
}

$o = new MyHelloWorld();
$o->sayHello();
$o->sayWorld();
$o->sayExclamationMark();
```

The above example will output:

```
Hello World!
```

Section 40.7: Changing Method Visibility

```
trait HelloWorld {
```

```
    public function sayHello() {
        echo 'Hello World!';
    }
}

// Change visibility of sayHello
class MyClass1 {
    use HelloWorld { sayHello as protected; }
}

// Alias method with changed visibility
// sayHello visibility not changed
class MyClass2 {
    use HelloWorld { sayHello as private myPrivateHello; }
}
```

Running this example:

```
(new MyClass1())->sayHello();
// Fatal error: Uncaught Error: Call to protected method MyClass1::sayHello()

(new MyClass2())->myPrivateHello();
// Fatal error: Uncaught Error: Call to private method MyClass2::myPrivateHello()

(new MyClass2())->sayHello();
// Hello World!
```

So be aware that in the last example in `MyClass2` the original un-aliased method from `trait HelloWorld` stays accessible as-is.

Chapter 41: Composer Dependency Manager

Parameter	Details
license	Defines the type of license you want to use in the Project.
authors	Defines the authors of the project, as well as the author details.
support	Defines the support emails, irc channel, and various links.
require	Defines the actual dependencies as well as the package versions.
require-dev	Defines the packages necessary for developing the project.
suggest	Defines the package suggestions, i.e. packages which can help if installed.
autoload	Defines the autoloading policies of the project.
autoload-dev	Defines the autoloading policies for developing the project.

Composer is PHP's most commonly used dependency manager. It's analogous to `npm` in Node, `pip` for Python, or `NuGet` for .NET.

Section 41.1: What is Composer?

Composer is a dependency/package manager for PHP. It can be used to install, keep track of, and update your project dependencies. Composer also takes care of autoloading the dependencies that your application relies on, letting you easily use the dependency inside your project without worrying about including them at the top of any given file.

Dependencies for your project are listed within a `composer.json` file which is typically located in your project root. This file holds information about the required versions of packages for production and also development.

A full outline of the `composer.json` schema can be found on the Composer Website.

This file can be edited manually using any text-editor or automatically through the command line via commands such as `composer require <package>` or `composer require-dev <package>`.

To start using composer in your project, you will need to create the `composer.json` file. You can either create it manually or simply run `composer init`. After you run `composer init` in your terminal, it will ask you for some basic information about your project: **Package name** (*vendor/package* - e.g. `laravel/laravel`), **Description** - *optional*, **Author** and some other information like Minimum Stability, License and Required Packages.

The `require` key in your `composer.json` file specifies Composer which packages your project depends on. `require` takes an object that maps package names (e.g. *monolog/monolog*) to version constraints (e.g. *1.0.**).

```
{
    "require": {
        "composer/composer": "1.2.*"
    }
}
```

To install the defined dependencies, you will need to run the `composer install` command and it will then find the defined packages that matches the supplied `version` constraint and download it into the `vendor` directory. It's a convention to put third party code into a directory named `vendor`.

You will notice the `install` command also created a `composer.lock` file.

A `composer.lock` file is automatically generated by Composer. This file is used to track the currently installed

versions and state of your dependencies. Running `composer install` will install packages to exactly the state stored in the lock file.

Section 41.2: Autoloading with Composer

While composer provides a system to manage dependencies for PHP projects (e.g. from Packagist), it can also notably serve as an autoloader, specifying where to look for specific namespaces or include generic function files.

It starts with the `composer.json` file:

```
{
    // ...
    "autoload": {
        "psr-4": {
            "MyVendorName\\MyProject": "src/"
        },
        "files": [
            "src/functions.php"
        ]
    },
    "autoload-dev": {
        "psr-4": {
            "MyVendorName\\MyProject\\Tests": "tests/"
        }
    }
}
```

This configuration code ensures that all classes in the namespace `MyVendorName\MyProject` are mapped to the `src` directory and all classes in `MyVendorName\MyProject\Tests` to the `tests` directory (relative to your root directory). It will also automatically include the file `functions.php`.

After putting this in your `composer.json` file, run `composer update` in a terminal to have composer update the dependencies, the lock file and generate the `autoload.php` file. When deploying to a production environment you would use `composer install --no-dev`. The `autoload.php` file can be found in the `vendor` directory which should be generated in the directory where `composer.json` resides.

You should `require` this file early at a setup point in the lifecycle of your application using a line similar to that below.

```
require_once __DIR__ . '/vendor/autoload.php';
```

Once included, the `autoload.php` file takes care of loading all the dependencies that you provided in your `composer.json` file.

Some examples of the class path to directory mapping:

- `MyVendorName\MyProject\Shapes\Square` ➔ `src/Shapes/Square.php`.
- `MyVendorName\MyProject\Tests\Shapes\Square` ➔ `tests/Shapes/Square.php`.

Section 41.3: Difference between 'composer install' and 'composer update'

`composer update`

`composer update` will update our dependencies as they are specified in `composer.json`.

For example, if our project uses this configuration:

```
"require": {
    "laravelcollective/html": "2.0.*"
}
```

Supposing we have actually installed the `2.0.1` version of the package, running `composer update` will cause an upgrade of this package (for example to `2.0.2`, if it has already been released).

In detail `composer update` will:

- Read `composer.json`
- Remove installed packages that are no more required in `composer.json`
- Check the availability of the latest versions of our required packages
- Install the latest versions of our packages
- Update `composer.lock` to store the installed packages version

`composer install`

`composer install` will install all of the dependencies as specified in the `composer.lock` file at the version specified (locked), without updating anything.

In detail:

- Read `composer.lock` file
- Install the packages specified in the `composer.lock` file

When to install and when to update

- `composer update` is mostly used in the 'development' phase, to upgrade our project packages.
- `composer install` is primarily used in the 'deploying phase' to install our application on a production server or on a testing environment, using the same dependencies stored in the `composer.lock` file created by `composer update`.

Section 41.4: Composer Available Commands

Command	Usage
about	Short information about Composer
archive	Create an archive of this composer package
browse	Opens the package's repository URL or homepage in your browser.
clear-cache	Clears composer's internal package cache.
clearcache	Clears composer's internal package cache.
config	Set config options
create-project	Create new project from a package into given directory.
depends	Shows which packages cause the given package to be installed
diagnose	Diagnoses the system to identify common errors.
dump-autoload	Dumps the autoloader
dumpautoload	Dumps the autoloader
exec	Execute a vendored binary/script
global	Allows running commands in the global composer dir ($COMPOSER_HOME).
help	Displays help for a command
home	Opens the package's repository URL or homepage in your browser.

info	Show information about packages
init	Creates a basic composer.json file in current directory.
install	Installs the project dependencies from the composer.lock file if present, or falls back on the composer.json.
licenses	Show information about licenses of dependencies
list	Lists commands
outdated	Shows a list of installed packages that have updates available, including their latest version.
prohibits	Shows which packages prevent the given package from being installed
remove	Removes a package from the require or require-dev
require	Adds required packages to your composer.json and installs them
run-script	Run the scripts defined in composer.json.
search	Search for packages
self-update	Updates composer.phar to the latest version.
selfupdate	Updates composer.phar to the latest version.
show	Show information about packages
status	Show a list of locally modified packages
suggests	Show package suggestions
update	Updates your dependencies to the latest version according to composer.json, and updates the composer.lock file.
validate	Validates a composer.json and composer.lock
why	Shows which packages cause the given package to be installed
why-not	Shows which packages prevent the given package from being installed

Section 41.5: Benefits of Using Composer

Composer tracks which versions of packages you have installed in a file called `composer.lock`, which is intended to be committed to version control, so that when the project is cloned in the future, simply running `composer install` will download and install all the project's dependencies.

Composer deals with PHP dependencies on a per-project basis. This makes it easy to have several projects on one machine that depend on separate versions of one PHP package.

Composer tracks which dependencies are only intended for dev environments only

```
composer require --dev phpunit/phpunit
```

Composer provides an autoloader, making it extremely easy to get started with any package. For instance, after installing Goutte with `composer require fabpot/goutte`, you can immediately start to use Goutte in a new project:

```
<?php

require __DIR__ . '/vendor/autoload.php';

$client = new Goutte\Client();

// Start using Goutte
```

Composer allows you to easily update a project to the latest version that is allowed by your composer.json. EG. `composer update fabpot/goutte`, or to update each of your project's dependencies: `composer update`.

Section 41.6: Installation

You may install Composer locally, as part of your project, or globally as a system wide executable.

Locally

To install, run these commands in your terminal.

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
# to check the validity of the downloaded installer, check here against the SHA-384:
# https://composer.github.io/pubkeys.html
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

This will download `composer.phar` (a PHP Archive file) to the current directory. Now you can run `php composer.phar` to use Composer, e.g.

```
php composer.phar install
```

Globally

To use Composer globally, place the composer.phar file to a directory that is part of your `PATH`

```
mv composer.phar /usr/local/bin/composer
```

Now you can use `composer` anywhere instead of `php composer.phar`, e.g.

```
composer install
```

Chapter 42: Magic Methods

Section 42.1: __call() and __callStatic()

`__call()` and `__callStatic()` are called when somebody is calling nonexistent object method in object or static context.

```
class Foo
{
    /**
     * This method will be called when somebody will try to invoke a method in object
     * context, which does not exist, like:
     *
     * $foo->method($arg, $arg1);
     *
     * First argument will contain the method name(in example above it will be "method"),
     * and the second will contain the values of $arg and $arg1 as an array.
     */
    public function __call($method, $arguments)
    {
        // do something with that information here, like overloading
        // or something generic.
        // For sake of example let's say we're making a generic class,
        // that holds some data and allows user to get/set/has via
        // getter/setter methods. Also let's assume that there is some
        // CaseHelper which helps to convert camelCase into snake_case.
        // Also this method is simplified, so it does not check if there
        // is a valid name or
        $snakeName = CaseHelper::camelToSnake($method);
        // Get get/set/has prefix
        $subMethod = substr($snakeName, 0, 3);

        // Drop method name.
        $propertyName = substr($snakeName, 4);

        switch ($subMethod) {
            case "get":
                return $this->data[$propertyName];
            case "set":
                $this->data[$propertyName] = $arguments[0];
                break;
            case "has":
                return isset($this->data[$propertyName]);
            default:
                throw new BadMethodCallException("Undefined method $method");
        }
    }

    /**
     * __callStatic will be called from static content, that is, when calling a nonexistent
     * static method:
     *
     * Foo::buildSomethingCool($arg);
     *
     * First argument will contain the method name(in example above it will be "buildSomethingCool"),
     * and the second will contain the value $arg in an array.
     *
     * Note that signature of this method is different(requires static keyword). This method was not
     * available prior PHP 5.3
     */
```

```
    public static function __callStatic($method, $arguments)
    {
        // This method can be used when you need something like generic factory
        // or something else(to be honest use case for this is not so clear to me).
        print_r(func_get_args());
    }
}
```

Example:

```
$instance = new Foo();

$instance->setSomeState("foo");
var_dump($instance->hasSomeState());      // bool(true)
var_dump($instance->getSomeState());      // string "foo"

Foo::exampleStaticCall("test");
// outputs:
Array
(
    [0] => exampleCallStatic
    [1] => test
)
```

Section 42.2: __get(), __set(), __isset() and __unset()

Whenever you attempt to retrieve a certain field from a class like so:

```
$animal = new Animal();
$height = $animal->height;
```

PHP invokes the magic method `__get($name)`, with `$name` equal to `"height"` in this case. Writing to a class field like so:

```
$animal->height = 10;
```

Will invoke the magic method `__set($name, $value)`, with `$name` equal to `"height"` and `$value` equal to `10`.

PHP also has two built-in functions `isset()`, which check if a variable exists, and `unset()`, which destroys a variable. Checking whether a objects field is set like so:

```
isset($animal->height);
```

Will invoke the `__isset($name)` function on that object. Destroying a variable like so:

```
unset($animal->height);
```

Will invoke the `__unset($name)` function on that object.

Normally, when you don't define these methods on your class, PHP just retrieves the field as it is stored in your class. However, you can override these methods to create classes that can hold data like an array, but are usable like an object:

```
class Example {
    private $data = [];

    public function __set($name, $value) {
```

```
        $this->data[$name] = $value;
    }

    public function __get($name) {
        if (!array_key_exists($name, $this->data)) {
            return null;
        }

        return $this->data[$name];
    }

    public function __isset($name) {
        return isset($this->data[$name]);
    }

    public function __unset($name) {
        unset($this->data[$name]);
    }
}

$example = new Example();

// Stores 'a' in the $data array with value 15
$example->a = 15;

// Retrieves array key 'a' from the $data array
echo $example->a; // prints 15

// Attempt to retrieve non-existent key from the array returns null
echo $example->b; // prints nothing

// If __isset('a') returns true, then call __unset('a')
if (isset($example->a)) {
    unset($example->a));
}
```

empty() function and magic methods

Note that calling `empty()` on a class attribute will invoke `__isset()` because as the PHP manual states:

> empty() is essentially the concise equivalent to **!isset($var) || $var == false**

Section 42.3: __construct() and __destruct()

`__construct()` is the most common magic method in PHP, because it is used to set up a class when it is initialized. The opposite of the `__construct()` method is the `__destruct()` method. This method is called when there are no more references to an object that you created or when you force its deletion. PHP's garbage collection will clean up the object by first calling its destructor and then removing it from memory.

```
class Shape {
    public function __construct() {
        echo "Shape created!\n";
    }
}

class Rectangle extends Shape {
    public $width;
    public $height;
```

```
    public function __construct($width, $height) {
        parent::__construct();

        $this->width = $width;
        $this->height = $height;
        echo "Created {$this->width}x{$this->height} Rectangle\n";
    }

    public function __destruct() {
        echo "Destroying {$this->width}x{$this->height} Rectangle\n";
    }
}

function createRectangle() {
    // Instantiating an object will call the constructor with the specified arguments
    $rectangle = new Rectangle(20, 50);

    // 'Shape Created' will be printed
    // 'Created 20x50 Rectangle' will be printed
}

createRectangle();
// 'Destroying 20x50 Rectangle' will be printed, because
// the `$rectangle` object was local to the createRectangle function, so
// When the function scope is exited, the object is destroyed and its
// destructor is called.

// The destructor of an object is also called when unset is used:
unset(new Rectangle(20, 50));
```

Section 42.4: __toString()

Whenever an object is treated as a string, the `__toString()` method is called. This method should return a string representation of the class.

```
class User {
    public $first_name;
    public $last_name;
    public $age;

    public function __toString() {
        return "{$this->first_name} {$this->last_name} ({$this->age})";
    }
}

$user = new User();
$user->first_name = "Chuck";
$user->last_name = "Norris";
$user->age = 76;

// Anytime the $user object is used in a string context, __toString() is called

echo $user; // prints 'Chuck Norris (76)'

// String value becomes: 'Selected user: Chuck Norris (76)'
$selected_user_string = sprintf("Selected user: %s", $user);

// Casting to string also calls __toString()
$user_as_string = (string) $user;
```

Section 42.5: __clone()

`__clone` is invoked by use of the `clone` keyword. It is used to manipulate object state upon cloning, after the object has been actually cloned.

```
class CloneableUser
{
    public $name;
    public $lastName;

    /**
     * This method will be invoked by a clone operator and will prepend "Copy " to the
     * name and lastName properties.
     */
    public function __clone()
    {
        $this->name = "Copy " . $this->name;
        $this->lastName = "Copy " . $this->lastName;
    }
}
```

Example:

```
$user1 = new CloneableUser();
$user1->name = "John";
$user1->lastName = "Doe";

$user2 = clone $user1; // triggers the __clone magic method

echo $user2->name;     // Copy John
echo $user2->lastName; // Copy Doe
```

Section 42.6: __invoke()

This magic method is called when user tries to invoke object as a function. Possible use cases may include some approaches like functional programming or some callbacks.

```
class Invokable
{
    /**
     * This method will be called if object will be executed like a function:
     *
     * $invokable();
     *
     * Args will be passed as in regular method call.
     */
    public function __invoke($arg, $arg, ...)
    {
        print_r(func_get_args());
    }
}

// Example:
$invokable = new Invokable();
$invokable([1, 2, 3]);

// optputs:
Array
(
```

```
    [0] => 1
    [1] => 2
    [2] => 3
)
```

Section 42.7: __sleep() and __wakeup()

`__sleep` and `__wakeup` are methods that are related to the serialization process. `serialize` function checks if a class has a `__sleep` method. If so, it will be executed before any serialization. `__sleep` is supposed to return an array of the names of all variables of an object that should be serialized.

`__wakeup` in turn will be executed by `unserialize` if it is present in class. It's intention is to re-establish resources and other things that are needed to be initialized upon unserialization.

```
class Sleepy {
    public $tableName;
    public $tableFields;
    public $dbConnection;

    /**
     * This magic method will be invoked by serialize function.
     * Note that $dbConnection is excluded.
     */
    public function __sleep()
    {
        // Only $this->tableName and $this->tableFields will be serialized.
        return ['tableName', 'tableFields'];
    }

    /**
     * This magic method will be called by unserialize function.
     *
     * For sake of example, lets assume that $this->c, which was not serialized,
     * is some kind of a database connection. So on wake up it will get reconnected.
     */
    public function __wakeup()
    {
        // Connect to some default database and store handler/wrapper returned into
        // $this->dbConnection
        $this->dbConnection = DB::connect();
    }
}
```

Section 42.8: __debugInfo()

> This method is called by `var_dump()` when dumping an object to get the properties that should be shown. If the method isn't defined on an object, then all public, protected and private properties will be shown.
> — PHP Manual

```
class DeepThought {
    public function __debugInfo() {
        return [42];
    }
}
```

Version ≤ 5.6

```
var_dump(new DeepThought());
```

The above example will output:

```
class DeepThought#1 (0) {
}
```

Version ≥ 5.6

```
var_dump(new DeepThought());
```

The above example will output:

```
class DeepThought#1 (1) {
 public ${0} =>
  int(42)
}
```

Chapter 43: File handling

Parameter	Description
filename	The filename being read.
use_include_path	You can use the optional second parameter and set it to TRUE, if you want to search for the file in the include_path, too.
context	A context stream resource.

Section 43.1: Convenience functions

Raw direct IO

`file_get_contents` and `file_put_contents` provide the ability to read/write from/to a file to/from a PHP string in a single call.

`file_put_contents` can also be used with the `FILE_APPEND` bitmask flag to append to, instead of truncate and overwrite, the file. It can be used along with `LOCK_EX` bitmask to acquire an exclusive lock to the file while proceeding to writing. Bitmask flags can be joined with the `|` bitwise-OR operator.

```
$path = "file.txt";
// reads contents in file.txt to $contents
$contents = file_get_contents($path);
// let's change something... for example, convert the CRLF to LF!
$contents = str_replace("\r\n", "\n", $contents);
// now write it back to file.txt, replacing the original contents
file_put_contents($path, $contents);
```

`FILE_APPEND` is handy for appending to log files while `LOCK_EX` helps prevent race condition of file writing from multiple processes. For example, to write to a log file about the current session:

```
file_put_contents("logins.log", "{$_SESSION["username"]} logged in", FILE_APPEND | LOCK_EX);
```

CSV IO

```
fgetcsv($file, $length, $separator)
```

The `fgetcsv` parses line from open file checking for csv fields. It returns CSV fields in an array on success or **`FALSE`** on failure.

By default, it will read only one line of the CSV file.

```
$file = fopen("contacts.csv","r");
print_r(fgetcsv($file));
print_r(fgetcsv($file,5," "));
fclose($file);
```

`contacts.csv`

```
Kai Jim, Refsnes, Stavanger, Norway
Hege, Refsnes, Stavanger, Norway
```

Output:

```
Array
(
    [0] => Kai Jim
```

```
    [1] => Refsnes
    [2] => Stavanger
    [3] => Norway
)
Array
(
    [0] => Hege,
)
```

Reading a file to stdout directly

readfile copies a file to the output buffer. readfile() will not present any memory issues, even when sending large files, on its own.

```
$file = 'monkey.gif';

if (file_exists($file)) {
    header('Content-Description: File Transfer');
    header('Content-Type: application/octet-stream');
    header('Content-Disposition: attachment; filename="'.basename($file).'"');
    header('Expires: 0');
    header('Cache-Control: must-revalidate');
    header('Pragma: public');
    header('Content-Length: ' . filesize($file));
    readfile($file);
    exit;
}
```

Or from a file pointer

Alternatively, to seek a point in the file to start copying to stdout, use fpassthru instead. In the following example, the last 1024 bytes are copied to stdout:

```
$fh = fopen("file.txt", "rb");
fseek($fh, -1024, SEEK_END);
fpassthru($fh);
```

Reading a file into an array

file returns the lines in the passed file in an array. Each element of the array corresponds to a line in the file, with the newline still attached.

```
print_r(file("test.txt"));
```

test.txt

```
Welcome to File handling
This is to test file handling
```

Output:

```
Array
(
    [0] => Welcome to File handling
    [1] => This is to test file handling
)
```

Section 43.2: Deleting files and directories

Deleting files

The `unlink` function deletes a single file and returns whether the operation was successful.

```
$filename = '/path/to/file.txt';

if (file_exists($filename)) {
    $success = unlink($filename);

    if (!$success) {
         throw new Exception("Cannot delete $filename");
    }
}
```

Deleting directories, with recursive deletion

On the other hand, directories should be deleted with `rmdir`. However, this function only deletes empty directories. To delete a directory with files, delete the files in the directories first. If the directory contains subdirectories, *recursion* may be needed.

The following example scans files in a directory, deletes member files/directories recursively, and returns the number of files (not directories) deleted.

```
function recurse_delete_dir(string $dir) : int {
    $count = 0;

    // ensure that $dir ends with a slash so that we can concatenate it with the filenames directly
    $dir = rtrim($dir, "/\\") . "/";

    // use dir() to list files
    $list = dir($dir);

    // store the next file name to $file. if $file is false, that's all -- end the loop.
    while(($file = $list->read()) !== false) {
        if($file === "." || $file === "..") continue;
        if(is_file($dir . $file)) {
            unlink($dir . $file);
            $count++;
        } elseif(is_dir($dir . $file)) {
            $count += recurse_delete_dir($dir . $file);
        }
    }

    // finally, safe to delete directory!
    rmdir($dir);

    return $count;
}
```

Section 43.3: Getting file information

Check if a path is a directory or a file

The `is_dir` function returns whether the argument is a directory, while `is_file` returns whether the argument is a file. Use `file_exists` to check if it is either.

```
$dir  = "/this/is/a/directory";
```

```php
$file = "/this/is/a/file.txt";

echo is_dir($dir) ? "$dir is a directory" : "$dir is not a directory", PHP_EOL,
    is_file($dir) ? "$dir is a file" : "$dir is not a file", PHP_EOL,
    file_exists($dir) ? "$dir exists" : "$dir doesn't exist", PHP_EOL,
    is_dir($file) ? "$file is a directory" : "$file is not a directory", PHP_EOL,
    is_file($file) ? "$file is a file" : "$file is not a file", PHP_EOL,
    file_exists($file) ? "$file exists" : "$file doesn't exist", PHP_EOL;
```

This gives:

```
/this/is/a/directory is a directory
/this/is/a/directory is not a file
/this/is/a/directory exists
/this/is/a/file.txt is not a directory
/this/is/a/file.txt is a file
/this/is/a/file.txt exists
```

Checking file type

Use `filetype` to check the type of a file, which may be:

- `fifo`
- `char`
- `dir`
- `block`
- `link`
- `file`
- `socket`
- `unknown`

Passing the filename to the `filetype` directly:

```php
echo filetype("~"); // dir
```

Note that `filetype` returns false and triggers an `E_WARNING` if the file doesn't exist.

Checking readability and writability

Passing the filename to the `is_writable` and `is_readable` functions check whether the file is writable or readable respectively.

The functions return `false` gracefully if the file does not exist.

Checking file access/modify time

Using `filemtime` and `fileatime` returns the timestamp of the last modification or access of the file. The return value is a Unix timestamp -- see Working with Dates and Time for details.

```php
echo "File was last modified on " . date("Y-m-d", filemtime("file.txt"));
echo "File was last accessed on " . date("Y-m-d", fileatime("file.txt"));
```

Get path parts with fileinfo

```php
$fileToAnalyze = ('/var/www/image.png');

$filePathParts = pathinfo($fileToAnalyze);
```

```
echo '<pre>';
   print_r($filePathParts);
echo '</pre>';
```

This example will output:

```
Array
(
    [dirname] => /var/www
    [basename] => image.png
    [extension] => png
    [filename] => image
)
```

Which can be used as:

```
$filePathParts['dirname']
$filePathParts['basename']
$filePathParts['extension']
$filePathParts['filename']
```

Parameter	Details
$path	The full path of the file to be parsed
$option	One of four available options [PATHINFO_DIRNAME, PATHINFO_BASENAME, PATHINFO_EXTENSION or PATHINFO_FILENAME]

- If an option (the second parameter) is not passed, an associative array is returned otherwise a string is returned.
- Does not validate that the file exists.
- Simply parses the string into parts. No validation is done on the file (no mime-type checking, etc.)
- The extension is simply the last extension of `$path` The path for the file `image.jpg.png` would be `.png` even if it technically a `.jpg` file. A file without an extension will not return an extension element in the array.

Section 43.4: Stream-based file IO

Opening a stream

fopen opens a file stream handle, which can be used with various functions for reading, writing, seeking and other functions on top of it. This value is of `resource` type, and cannot be passed to other threads persisting its functionality.

```
$f = fopen("errors.log", "a"); // Will try to open errors.log for writing
```

The second parameter is the mode of the file stream:

Mode	Description
`r`	Open in read only mode, starting at the beginning of the file
`r+`	Open for reading and writing, starting at the beginning of the file
`w`	open for writing only, starting at the beginning of the file. If the file exists it will empty the file. If it doesn't exist it will attempt to create it.
`w+`	open for reading and writing, starting at the beginning of the file. If the file exists it will empty the file. If it doesn't exist it will attempt to create it.
`a`	open a file for writing only, starting at the end of the file. If the file does not exist, it will try to create it
`a+`	open a file for reading and writing, starting at the end of the file. If the file does not exist, it will try to create it

`x` create and open a file for writing only. If the file exists the `fopen` call will fail

`x+` create and open a file for reading and writing. If the file exists the `fopen` call will fail

`c` open the file for writing only. If the file does not exist it will try to create it. It will start writing at the beginning of the file, but will not empty the file ahead of writing

`c+` open the file for reading and writing. If the file does not exist it will try to create it. It will start writing at the beginning of the file, but will not empty the file ahead of writing

Adding a `t` behind the mode (e.g. `a+b`, `wt`, etc.) in Windows will translate `"\n"` line endings to `"\r\n"` when working with the file. Add `b` behind the mode if this is not intended, especially if it is a binary file.

The PHP application should close streams using `fclose` when they are no longer used to prevent the `Too many open files` error. This is particularly important in CLI programs, since the streams are only closed when the runtime shuts down -- this means that in web servers, it *may not be necessary* (but still *should*, as a practice to prevent resource leak) to close the streams if you do not expect the process to run for a long time, and will not open many streams.

Reading

Using `fread` will read the given number of bytes from the file pointer, or until an EOF is met.

Reading lines

Using `fgets` will read the file until an EOL is reached, or the given length is read.

Both `fread` and `fgets` will move the file pointer while reading.

Reading everything remaining

Using `stream_get_contents` will all remaining bytes in the stream into a string and return it.

Adjusting file pointer position

Initially after opening the stream, the file pointer is at the beginning of the file (or the end, if the mode `a` is used). Using the `fseek` function will move the file pointer to a new position, relative to one of three values:

- `SEEK_SET`: This is the default value; the file position offset will be relative to the beginning of the file.
- `SEEK_CUR`: The file position offset will be relative to the current position.
- `SEEK_END`: The file position offset will be relative to the end of the file. Passing a negative offset is the most common use for this value; it will move the file position to the specified number of bytes before the end of file.

`rewind` is a convenience shortcut of `fseek($fh, 0, SEEK_SET)`.

Using `ftell` will show the absolute position of the file pointer.

For example, the following script reads skips the first 10 bytes, reads the next 10 bytes, skips 10 bytes, reads the next 10 bytes, and then the last 10 bytes in file.txt:

```
$fh = fopen("file.txt", "rb");
fseek($fh, 10); // start at offset 10
echo fread($fh, 10); // reads 10 bytes
fseek($fh, 10, SEEK_CUR); // skip 10 bytes
echo fread($fh, 10); // read 10 bytes
fseek($fh, -10, SEEK_END); // skip to 10 bytes before EOF
echo fread($fh, 10); // read 10 bytes
```

```
fclose($fh);
```

Writing

Using fwrite writes the provided string to the file starting at the current file pointer.

```
fwrite($fh, "Some text here\n");
```

Section 43.5: Moving and Copying files and directories

Copying files

copy copies the source file in the first argument to the destination in the second argument. The resolved destination needs to be in a directory that is already created.

```
if (copy('test.txt', 'dest.txt')) {
    echo 'File has been copied successfully';
} else {
    echo 'Failed to copy file to destination given.'
}
```

Copying directories, with recursion

Copying directories is pretty much similar to deleting directories, except that for files copy instead of unlink is used, while for directories, mkdir instead of rmdir is used, at the beginning instead of being at the end of the function.

```
function recurse_delete_dir(string $src, string $dest) : int {
    $count = 0;

    // ensure that $src and $dest end with a slash so that we can concatenate it with the filenames
directly
    $src = rtrim($dest, "/\\") . "/";
    $dest = rtrim($dest, "/\\") . "/";

    // use dir() to list files
    $list = dir($src);

    // create $dest if it does not already exist
    @mkdir($dest);

    // store the next file name to $file. if $file is false, that's all -- end the loop.
    while(($file = $list->read()) !== false) {
        if($file === "." || $file === "..") continue;
        if(is_file($src . $file)) {
            copy($src . $file, $dest . $file);
            $count++;
        } elseif(is_dir($src . $file)) {
            $count += recurse_copy_dir($src . $file, $dest . $file);
        }
    }

    return $count;
}
```

Renaming/Moving

Renaming/Moving files and directories is much simpler. Whole directories can be moved or renamed in a single call, using the rename function.

- `rename("~/file.txt", "~/file.html");`

- `rename("~/dir", "~/old_dir");`
- `rename("~/dir/file.txt", "~/dir2/file.txt");`

Section 43.6: Minimize memory usage when dealing with large files

If we need to parse a large file, e.g. a CSV more than 10 Mbytes containing millions of rows, some use `file` or `file_get_contents` functions and end up with hitting `memory_limit` setting with

> Allowed memory size of XXXXX bytes exhausted

error. Consider the following source (top-1m.csv has exactly 1 million rows and is about 22 Mbytes of size)

```
var_dump(memory_get_usage(true));
$arr = file('top-1m.csv');
var_dump(memory_get_usage(true));
```

This outputs:

```
int(262144)
int(210501632)
```

because the interpreter needed to hold all the rows in `$arr` array, so it consumed ~200 Mbytes of RAM. Note that we haven't even done anything with the contents of the array.

Now consider the following code:

```
var_dump(memory_get_usage(true));
$index = 1;
if (($handle = fopen("top-1m.csv", "r")) !== FALSE) {
    while (($row = fgetcsv($handle, 1000, ",")) !== FALSE) {
        file_put_contents('top-1m-reversed.csv',$index . ',' . strrev($row[1]) . PHP_EOL,
FILE_APPEND);
        $index++;
    }
    fclose($handle);
}
var_dump(memory_get_usage(true));
```

which outputs

```
int(262144)
int(262144)
```

so we don't use a single extra byte of memory, but parse the whole CSV and save it to another file reversing the value of the 2nd column. That's because `fgetcsv` reads only one row and `$row` is overwritten in every loop.

Chapter 44: Streams

Parameter Name	Description
Stream Resource	The data provider consisting of the `<scheme>://<target>` syntax

Section 44.1: Registering a stream wrapper

A stream wrapper provides a handler for one or more specific schemes.

The example below shows a simple stream wrapper that sends `PATCH` HTTP requests when the stream is closed.

```
// register the FooWrapper class as a wrapper for foo:// URLs.
stream_wrapper_register("foo", FooWrapper::class, STREAM_IS_URL) or die("Duplicate stream wrapper
registered");

class FooWrapper {
    // this will be modified by PHP to show the context passed in the current call.
    public $context;

    // this is used in this example internally to store the URL
    private $url;

    // when fopen() with a protocol for this wrapper is called, this method can be implemented to
store data like the host.
    public function stream_open(string $path, string $mode, int $options, string &$openedPath) :
bool {
        $url = parse_url($path);
        if($url === false) return false;
        $this->url = $url["host"] . "/" . $url["path"];
        return true;
    }

    // handles calls to fwrite() on this stream
    public function stream_write(string $data) : int {
        $this->buffer .= $data;
        return strlen($data);
    }

    // handles calls to fclose() on this stream
    public function stream_close() {
        $curl = curl_init("http://" . $this->url);
        curl_setopt($curl, CURLOPT_POSTFIELDS, $this->buffer);
        curl_setopt($curl, CURLOPT_CUSTOMREQUEST, "PATCH");
        curl_exec($curl);
        curl_close($curl);
        $this->buffer = "";
    }

    // fallback exception handler if an unsupported operation is attempted.
    // this is not necessary.
    public function __call($name, $args) {
        throw new \RuntimeException("This wrapper does not support $name");
    }

    // this is called when unlink("foo://something-else") is called.
    public function unlink(string $path) {
        $url = parse_url($path);
        $curl = curl_init("http://" . $url["host"] . "/" . $url["path"]);
        curl_setopt($curl, CURLOPT_CUSTOMREQUEST, "DELETE");
```

```
        curl_exec($curl);
        curl_close($curl);
    }
}
```

This example only shows some examples of what a generic stream wrapper would contain. These are not all methods available. A full list of methods that can be implemented can be found at http://php.net/streamWrapper.

Chapter 45: Type hinting

Section 45.1: Type hinting classes and interfaces

Type hinting for classes and interfaces was added in PHP 5.

Class type hint

```php
<?php

class Student
{
    public $name = 'Chris';
}

class School
{
    public $name = 'University of Edinburgh';
}

function enroll(Student $student, School $school)
{
    echo $student->name . ' is being enrolled at ' . $school->name;
}

$student = new Student();
$school = new School();

enroll($student, $school);
```

The above script outputs:

> Chris is being enrolled at University of Edinburgh

Interface type hint

```php
<?php

interface Enrollable {};
interface Attendable {};

class Chris implements Enrollable
{
    public $name = 'Chris';
}

class UniversityOfEdinburgh implements Attendable
{
    public $name = 'University of Edinburgh';
}

function enroll(Enrollable $enrollee, Attendable $premises)
{
    echo $enrollee->name . ' is being enrolled at ' . $premises->name;
}

$chris = new Chris();
$edinburgh = new UniversityOfEdinburgh();
```

```
enroll($chris, $edinburgh);
```

The above example outputs the same as before:

> Chris is being enrolled at University of Edinburgh

Self type hints

The `self` keyword can be used as a type hint to indicate that the value must be an instance of the class that declares the method.

Section 45.2: Type hinting scalar types, arrays and callables

Support for type hinting array parameters (and return values after PHP 7.1) was added in PHP 5.1 with the keyword `array`. Any arrays of any dimensions and types, as well as empty arrays, are valid values.

Support for type hinting callables was added in PHP 5.4. Any value that `is_callable()` is valid for parameters and return values hinted `callable`, i.e. `Closure` objects, function name strings and `array(class_name|object, method_name)`.

If a typo occurs in the function name such that it is not `is_callable()`, a less obvious error message would be displayed:

> Fatal error: Uncaught TypeError: Argument 1 passed to foo() must be of the type callable, string/array given

```
function foo(callable $c) {}
foo("count"); // valid
foo("Phar::running"); // valid
foo(["Phar", "running"); // valid
foo([new ReflectionClass("stdClass"), "getName"]); // valid
foo(function() {}); // valid

foo("no_such_function"); // callable expected, string given
```

Nonstatic methods can also be passed as callables in static format, resulting in a deprecation warning and level E_STRICT error in PHP 7 and 5 respectively.

Method visibility is taken into account. If the *context of the method with the `callable` parameter* does not have access to the callable provided, it will end up as if the method does not exist.

```
class Foo{
  private static function f(){
    echo "Good" . PHP_EOL;
  }

  public static function r(callable $c){
    $c();
  }
}

function r(callable $c){}

Foo::r(["Foo", "f"]);
```

```
r(["Foo", "f"]);
```

Output:

> Fatal error: Uncaught TypeError: Argument 1 passed to r() must be callable, array given

Support for type hinting scalar types was added in PHP 7. This means that we gain type hinting support for `booleans`, `integers`, `floats` and `strings`.

```
<?php

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

var_dump(add(1, 2)); // Outputs "int(3)"
```

By default, PHP will attempt to cast any provided argument to match its type hint. Changing the call to `add(1.5, 2)` gives exactly the same output, since the float `1.5` was cast to `int` by PHP.

To stop this behavior, one must add `declare(strict_types=1);` to the top of every PHP source file that requires it.

```
<?php

declare(strict_types=1);

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

var_dump(add(1.5, 2));
```

The above script now produces a fatal error:

> Fatal error: Uncaught TypeError: Argument 1 passed to add() must be of the type integer, float given

An Exception: Special Types

Some PHP functions may return a value of type `resource`. Since this is not a scalar type, but a special type, it is not possible to type hint it.

As an example, `curl_init()` will return a `resource`, as well as `fopen()`. Of course, those two resources aren't compatible to each other. Because of that, PHP 7 will *always* throw the following TypeError when type hinting `resource` explicitly:

> TypeError: Argument 1 passed to sample() must be an instance of resource, resource given

Section 45.3: Nullable type hints

Parameters

Nullable type hint was added in PHP 7.1 using the `?` operator before the type hint.

```php
function f(?string $a) {}
function g(string $a) {}

f(null); // valid
g(null); // TypeError: Argument 1 passed to g() must be of the type string, null given
```

Before PHP 7.1, if a parameter has a type hint, it must declare a default value `null` to accept null values.

```php
function f(string $a = null) {}
function g(string $a) {}

f(null); // valid
g(null); // TypeError: Argument 1 passed to g() must be of the type string, null given
```

Return values

In PHP 7.0, functions with a return type must not return null.

In PHP 7.1, functions can declare a nullable return type hint. However, the function must still return null, not void (no/empty return statements).

```php
function f() : ?string {
    return null;
}

function g() : ?string {}
function h() : ?string {}

f(); // OK
g(); // TypeError: Return value of g() must be of the type string or null, none returned
h(); // TypeError: Return value of h() must be of the type string or null, none returned
```

Section 45.4: Type hinting generic objects

Since PHP objects don't inherit from any base class (including `stdClass`), there is no support for type hinting a generic object type.

For example, the below will not work.

```php
<?php

function doSomething(object $obj) {
    return $obj;
}

class ClassOne {}
class ClassTwo {}

$classOne= new ClassOne();
$classTwo= new ClassTwo();

doSomething($classOne);
doSomething($classTwo);
```

And will throw a fatal error:

> Fatal error: Uncaught TypeError: Argument 1 passed to doSomething() must be an instance of object, instance of OperationOne given

A workaround to this is to declare a degenerate interface that defines no methods, and have all of your objects implement this interface.

```
<?php

interface Object {}

function doSomething(Object $obj) {
    return $obj;
}

class ClassOne implements Object {}
class ClassTwo implements Object {}

$classOne = new ClassOne();
$classTwo = new ClassTwo();

doSomething($classOne);
doSomething($classTwo);
```

Section 45.5: Type Hinting No Return(Void)

In PHP 7.1, the `void` return type was added. While PHP has no actual `void` value, it is generally understood across programming languages that a function that returns nothing is returning `void`. This should not be confused with returning `null`, as `null` is a value that can be returned.

```
function lacks_return(): void {
    // valid
}
```

Note that if you declare a `void` return, you cannot return any values or you will get a fatal error:

```
function should_return_nothing(): void {
    return null; // Fatal error: A void function must not return a value
}
```

However, using return to exit the function is valid:

```
function returns_nothing(): void {
    return; // valid
}
```

Chapter 46: Filters & Filter Functions

Parameter	Details
variable	Value to filter. Note that scalar values are converted to string internally before they are filtered.
------	------
filter	The ID of the filter to apply. The Types of filters manual page lists the available filters.If omitted, FILTER_DEFAULT will be used, which is equivalent to FILTER_UNSAFE_RAW. This will result in no filtering taking place by default.
------	------
options	Associative array of options or bitwise disjunction of flags. If filter accepts options, flags can be provided in "flags" field of array. For the "callback" filter, callable type should be passed. The callback must accept one argument, the value to be filtered, and return the value after filtering/sanitizing it.

This extension filters data by either validating or sanitizing it. This is especially useful when the data source contains unknown (or foreign) data, like user supplied input. For example, this data may come from an HTML form.

Section 46.1: Validating Boolean Values

```
var_dump(filter_var(true, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // true
var_dump(filter_var(false, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var(1, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // true
var_dump(filter_var(0, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var('1', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // true
var_dump(filter_var('0', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var('', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var(' ', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var('true', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // true
var_dump(filter_var('false', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var([], FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // NULL
var_dump(filter_var(null, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
```

Section 46.2: Validating A Number Is A Float

Validates value as float, and converts to float on success.

```
var_dump(filter_var(1, FILTER_VALIDATE_FLOAT));
var_dump(filter_var(1.0, FILTER_VALIDATE_FLOAT));
var_dump(filter_var(1.0000, FILTER_VALIDATE_FLOAT));
var_dump(filter_var(1.00001, FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1.0', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1.0000', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1.00001', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1,000', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1,000.0', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1,000.0000', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1,000.00001', FILTER_VALIDATE_FLOAT));


var_dump(filter_var(1, FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.0, FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.0000, FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.00001, FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.0', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.0000', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.00001', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
```

```
var_dump(filter_var('1,000', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.0', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.0000', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.00001', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
```

Results

```
float(1)
float(1)
float(1)
float(1.00001)
float(1)
float(1)
float(1)
float(1.00001)
bool(false)
bool(false)
bool(false)
bool(false)

float(1)
float(1)
float(1)
float(1.00001)
float(1)
float(1)
float(1)
float(1.00001)
float(1000)
float(1000)
float(1000)
float(1000.00001)
```

Section 46.3: Validate A MAC Address

Validates a value is a valid MAC address

```
var_dump(filter_var('FA-F9-DD-B2-5E-0D', FILTER_VALIDATE_MAC));
var_dump(filter_var('DC-BB-17-9A-CE-81', FILTER_VALIDATE_MAC));
var_dump(filter_var('96-D5-9E-67-40-AB', FILTER_VALIDATE_MAC));
var_dump(filter_var('96-D5-9E-67-40', FILTER_VALIDATE_MAC));
var_dump(filter_var('', FILTER_VALIDATE_MAC));
```

Results:

```
string(17) "FA-F9-DD-B2-5E-0D"
string(17) "DC-BB-17-9A-CE-81"
string(17) "96-D5-9E-67-40-AB"
bool(false)
bool(false)
```

Section 46.4: Sanitze Email Addresses

Remove all characters except letters, digits and !#$%&'*+-=?^_`{|}~@.[].

```
var_dump(filter_var('john@example.com', FILTER_SANITIZE_EMAIL));
var_dump(filter_var("!#$%&'*+-=?^_`{|}~.[]@example.com", FILTER_SANITIZE_EMAIL));
var_dump(filter_var('john/@example.com', FILTER_SANITIZE_EMAIL));
```

```
var_dump(filter_var('john\@example.com', FILTER_SANITIZE_EMAIL));
var_dump(filter_var('joh n@example.com', FILTER_SANITIZE_EMAIL));
```

Results:

```
string(16) "john@example.com"
string(33) "!#$%&'*+-=?^_`{|}~.[]@example.com"
string(16) "john@example.com"
string(16) "john@example.com"
string(16) "john@example.com"
```

Section 46.5: Sanitize Integers

Remove all characters except digits, plus and minus sign.

```
var_dump(filter_var(1, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(-1, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(+1, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(1.00, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(+1.00, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(-1.00, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('1', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('-1', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('+1', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('1.00', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('+1.00', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('-1.00', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('1 unicorn', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('-1 unicorn', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('+1 unicorn', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var("!#$%&'*+-=?^_`{|}~@.[]0123456789abcdefghijklmnopqrstuvwxyz",
FILTER_SANITIZE_NUMBER_INT));
```

Results:

```
string(1) "1"
string(2) "-1"
string(1) "1"
string(1) "1"
string(1) "1"
string(2) "-1"
string(1) "1"
string(2) "-1"
string(2) "+1"
string(3) "100"
string(4) "+100"
string(4) "-100"
string(1) "1"
string(2) "-1"
string(2) "+1"
string(12) "+-0123456789"
```

Section 46.6: Sanitize URLs

Sanitze URLs

Remove all characters except letters, digits and $-_.+!*'(),{}|\^~[]`<>#%";/?:@&=

```
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y', FILTER_SANITIZE_URL));
var_dump(filter_var("http://www.example.com/path/to/dir/index.php?test=y!#$%&'*+-=?^_`{|}~.[]",
FILTER_SANITIZE_URL));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=a b c',
FILTER_SANITIZE_URL));
```

Results:

```
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(72) "http://www.example.com/path/to/dir/index.php?test=y!#$%&'*+-=?^_`{|}~.[]"
string(53) "http://www.example.com/path/to/dir/index.php?test=abc"
```

Section 46.7: Validate Email Address

When filtering an email address `filter_var()` will return the filtered data, in this case the email address, or false if a valid email address cannot be found:

```
var_dump(filter_var('john@example.com', FILTER_VALIDATE_EMAIL));
var_dump(filter_var('notValidEmail', FILTER_VALIDATE_EMAIL));
```

Results:

```
string(16) "john@example.com"
bool(false)
```

This function doesn't validate not-latin characters. Internationalized domain name can be validated in their `xn--` form.

Note that you cannot know if the email address is correct before sending an email to it. You may want to do some extra checks such as checking for a MX record, but this is not necessary. If you send a confirmation email, don't forget to remove unused accounts after a short period.

Section 46.8: Validating A Value Is An Integer

When filtering a value that should be an integer `filter_var()` will return the filtered data, in this case the integer, or false if the value is not an integer. Floats are *not* integers:

```
var_dump(filter_var('10', FILTER_VALIDATE_INT));
var_dump(filter_var('a10', FILTER_VALIDATE_INT));
var_dump(filter_var('10a', FILTER_VALIDATE_INT));
var_dump(filter_var(' ', FILTER_VALIDATE_INT));
var_dump(filter_var('10.00', FILTER_VALIDATE_INT));
var_dump(filter_var('10,000', FILTER_VALIDATE_INT));
var_dump(filter_var('-5', FILTER_VALIDATE_INT));
var_dump(filter_var('+7', FILTER_VALIDATE_INT));
```

Results:

```
int(10)
bool(false)
bool(false)
bool(false)
bool(false)
bool(false)
int(-5)
```

```
int(7)
```

If you are expecting only digits, you can use a regular expression:

```
if(is_string($_GET['entry']) && preg_match('#^[0-9]+$#', $_GET['entry']))
    // this is a digit (positive) integer
else
    // entry is incorrect
```

If you convert this value into an integer, you don't have to do this check and so you can use `filter_var`.

Section 46.9: Validating An Integer Falls In A Range

When validating that an integer falls in a range the check includes the minimum and maximum bounds:

```
$options = array(
    'options' => array(
        'min_range' => 5,
        'max_range' => 10,
    )
);
var_dump(filter_var('5', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('10', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('8', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('4', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('11', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('-6', FILTER_VALIDATE_INT, $options));
```

Results:

```
int(5)
int(10)
int(8)
bool(false)
bool(false)
bool(false)
```

Section 46.10: Validate a URL

When filtering a URL `filter_var()` will return the filtered data, in this case the URL, or false if a valid URL cannot be found:

URL: `example.com`

```
var_dump(filter_var('example.com', FILTER_VALIDATE_URL));
var_dump(filter_var('example.com', FILTER_VALIDATE_URL, FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('example.com', FILTER_VALIDATE_URL, FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('example.com', FILTER_VALIDATE_URL, FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('example.com', FILTER_VALIDATE_URL, FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
bool(false)
bool(false)
bool(false)
bool(false)
bool(false)
```

URL: `http://example.com`

```
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL));
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL, FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL, FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL, FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL, FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(18) "http://example.com"
string(18) "http://example.com"
string(18) "http://example.com"
bool(false)
bool(false)
```

URL: `http://www.example.com`

```
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL));
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL, FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL, FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL, FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL, FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(22) "http://www.example.com"
string(22) "http://www.example.com"
string(22) "http://www.example.com"
bool(false)
bool(false)
```

URL: `http://www.example.com/path/to/dir/`

```
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL));
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL,
FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL,
FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL,
FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL,
FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(35) "http://www.example.com/path/to/dir/"
string(35) "http://www.example.com/path/to/dir/"
string(35) "http://www.example.com/path/to/dir/"
string(35) "http://www.example.com/path/to/dir/"
bool(false)
```

URL: `http://www.example.com/path/to/dir/index.php`

```
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL,
FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL,
```

```
FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL,
FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL,
FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(44) "http://www.example.com/path/to/dir/index.php"
string(44) "http://www.example.com/path/to/dir/index.php"
string(44) "http://www.example.com/path/to/dir/index.php"
string(44) "http://www.example.com/path/to/dir/index.php"
bool(false)
```

URL: `http://www.example.com/path/to/dir/index.php?test=y`

```
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y', FILTER_VALIDATE_URL));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y', FILTER_VALIDATE_URL,
FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y', FILTER_VALIDATE_URL,
FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y', FILTER_VALIDATE_URL,
FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y', FILTER_VALIDATE_URL,
FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
```

Warning: You must check the protocol to protect you against an XSS attack:

```
var_dump(filter_var('javascript://comment%0Aalert(1)', FILTER_VALIDATE_URL));
// string(31) "javascript://comment%0Aalert(1)"
```

Section 46.11: Sanitize Floats

Remove all characters except digits, +- and optionally .,eE.

```
var_dump(filter_var(1, FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var(1.0, FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var(1.0000, FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var(1.00001, FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1.0', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1.0000', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1.00001', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1,000', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1,000.0', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1,000.0000', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1,000.00001', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1.8281e-009', FILTER_SANITIZE_NUMBER_FLOAT));
```

Results:

```
string(1) "1"
string(1) "1"
string(1) "1"
string(6) "100001"
string(1) "1"
string(2) "10"
string(5) "10000"
string(6) "100001"
string(4) "1000"
string(5) "10000"
string(8) "10000000"
string(9) "100000001"
string(9) "18281-009"
```

With the `FILTER_FLAG_ALLOW_THOUSAND` option:

```
var_dump(filter_var(1, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.0, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.0000, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.00001, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.0', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.0000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.00001', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.0', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.0000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.00001', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.8281e-009', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
```

Results:

```
string(1) "1"
string(1) "1"
string(6) "100001"
string(1) "1"
string(2) "10"
string(5) "10000"
string(6) "100001"
string(5) "1,000"
string(6) "1,0000"
string(9) "1,0000000"
string(10) "1,00000001"
string(9) "18281-009"
```

With the `FILTER_FLAG_ALLOW_SCIENTIFIC` option:

```
var_dump(filter_var(1, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var(1.0, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var(1.0000, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var(1.00001, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1.0', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1.0000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1.00001', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1,000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1,000.0', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1,000.0000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1,000.00001', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
```

```
var_dump(filter_var('1.8281e-009', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
```

Results:

```
string(1) "1"
string(1) "1"
string(1) "1"
string(6) "100001"
string(1) "1"
string(2) "10"
string(5) "10000"
string(6) "100001"
string(4) "1000"
string(5) "10000"
string(8) "10000000"
string(9) "100000001"
string(10) "18281e-009"
```

Section 46.12: Validate IP Addresses

Validates a value is a valid IP address

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP));
```

Results:

```
string(13) "185.158.24.24"
string(39) "2001:0db8:0a0b:12f0:0000:0000:0000:0001"
string(11) "192.168.0.1"
string(9) "127.0.0.1"
```

Validate an valid IPv4 IP address:

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP, FILTER_FLAG_IPV4));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP,
FILTER_FLAG_IPV4));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_IPV4));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_IPV4));
```

Results:

```
string(13) "185.158.24.24"
bool(false)
string(11) "192.168.0.1"
string(9) "127.0.0.1"
```

Validate an valid IPv6 IP address:

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP, FILTER_FLAG_IPV6));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP,
FILTER_FLAG_IPV6));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_IPV6));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_IPV6));
```

Results:

```
bool(false)
string(39) "2001:0db8:0a0b:12f0:0000:0000:0000:0001"
bool(false)
bool(false)
```

Validate an IP address is not in a private range:

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP, FILTER_FLAG_NO_PRIV_RANGE));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP,
FILTER_FLAG_NO_PRIV_RANGE));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_NO_PRIV_RANGE));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_NO_PRIV_RANGE));
```

Results:

```
string(13) "185.158.24.24"
string(39) "2001:0db8:0a0b:12f0:0000:0000:0000:0001"
bool(false)
string(9) "127.0.0.1"
```

Validate an IP address is not in a reserved range:

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP, FILTER_FLAG_NO_RES_RANGE));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP,
FILTER_FLAG_NO_RES_RANGE));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_NO_RES_RANGE));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_NO_RES_RANGE));
```

Results:

```
string(13) "185.158.24.24"
bool(false)
string(11) "192.168.0.1"
bool(false)
```

Section 46.13: Sanitize filters

we can use filters to sanitize our variable according to our need.

Example

```
$string = "<p>Example</p>";
$newstring = filter_var($string, FILTER_SANITIZE_STRING);
var_dump($newstring); // string(7) "Example"
```

above will remove the html tags from `$string` variable.

Chapter 47: Generators

Section 47.1: The Yield Keyword

A `yield` statement is similar to a return statement, except that instead of stopping execution of the function and returning, yield instead returns a Generator object and pauses execution of the generator function.

Here is an example of the range function, written as a generator:

```
function gen_one_to_three() {
    for ($i = 1; $i <= 3; $i++) {
        // Note that $i is preserved between yields.
        yield $i;
    }
}
```

You can see that this function returns a Generator object by inspecting the output of `var_dump`:

```
var_dump(gen_one_to_three())

# Outputs:
class Generator (0) {
}
```

Yielding Values

The Generator object can then be iterated over like an array.

```
foreach (gen_one_to_three() as $value) {
    echo "$value\n";
}
```

The above example will output:

```
1
2
3
```

Yielding Values with Keys

In addition to yielding values, you can also yield key/value pairs.

```
function gen_one_to_three() {
    $keys = ["first", "second", "third"];

    for ($i = 1; $i <= 3; $i++) {
        // Note that $i is preserved between yields.
        yield $keys[$i - 1] => $i;
    }
}

foreach (gen_one_to_three() as $key => $value) {
    echo "$key: $value\n";
}
```

The above example will output:

```
first: 1
second: 2
third: 3
```

Section 47.2: Reading a large file with a generator

One common use case for generators is reading a file from disk and iterating over its contents. Below is a class that allows you to iterate over a CSV file. The memory usage for this script is very predictable, and will not fluctuate depending on the size of the CSV file.

```
<?php

class CsvReader
{
    protected $file;

    public function __construct($filePath) {
        $this->file = fopen($filePath, 'r');
    }

    public function rows()
    {
        while (!feof($this->file)) {
            $row = fgetcsv($this->file, 4096);

            yield $row;
        }

        return;
    }
}

$csv = new CsvReader('/path/to/huge/csv/file.csv');

foreach ($csv->rows() as $row) {
    // Do something with the CSV row.
}
```

Section 47.3: Why use a generator?

Generators are useful when you need to generate a large collection to later iterate over. They're a simpler alternative to creating a class that implements an Iterator, which is often overkill.

For example, consider the below function.

```
function randomNumbers(int $length)
{
    $array = [];

    for ($i = 0; $i < $length; $i++) {
        $array[] = mt_rand(1, 10);
    }

    return $array;
}
```

All this function does is generates an array that's filled with random numbers. To use it, we might do

`randomNumbers(10)`, which will give us an array of 10 random numbers. What if we want to generate one million random numbers? `randomNumbers(1000000)` will do that for us, but at a cost of memory. One million integers stored in an array uses approximately **33 megabytes** of memory.

```
$startMemory = memory_get_usage();

$randomNumbers = randomNumbers(1000000);

echo memory_get_usage() - $startMemory, ' bytes';
```

This is due to the entire one million random numbers being generated and returned at once, rather than one at a time. Generators are an easy way to solve this issue.

Section 47.4: Using the send()-function to pass values to a generator

Generators are fast coded and in many cases a slim alternative to heavy iterator-implementations. With the fast implementation comes a little lack of control when a generator should stop generating or if it should generate something else. However this can be achieved with the usage of the `send()` function, enabling the requesting function to send parameters to the generator after every loop.

```
//Imagining accessing a large amount of data from a server, here is the generator for this:
function generateDataFromServerDemo()
{
    $indexCurrentRun = 0; //In this example in place of data from the server, I just send feedback
every time a loop ran through.

    $timeout = false;
    while (!$timeout)
    {
        $timeout = yield $indexCurrentRun; // Values are passed to caller. The next time the
generator is called, it will start at this statement. If send() is used, $timeout will take this
value.
        $indexCurrentRun++;
    }

    yield 'X of bytes are missing. </br>';
}

// Start using the generator
$generatorDataFromServer = generateDataFromServerDemo ();
foreach($generatorDataFromServer as $numberOfRuns)
{
    if ($numberOfRuns < 10)
    {
        echo $numberOfRuns . "</br>";
    }
    else
    {
        $generatorDataFromServer->send(true); //sending data to the generator
        echo $generatorDataFromServer->current(); //accessing the latest element (hinting how many
bytes are still missing.
    }
}
```

Resulting in this Output:

0
1
2
3
4
5
6
7
8
9
X bytes are missing.

Chapter 48: UTF-8

Section 48.1: Input

- You should verify every received string as being valid UTF-8 before you try to store it or use it anywhere. PHP's `mb_check_encoding()` does the trick, but you have to use it consistently. There's really no way around this, as malicious clients can submit data in whatever encoding they want.

```
$string = $_REQUEST['user_comment'];
if (!mb_check_encoding($string, 'UTF-8')) {
    // the string is not UTF-8, so re-encode it.
    $actualEncoding = mb_detect_encoding($string);
    $string = mb_convert_encoding($string, 'UTF-8', $actualEncoding);
}
```

- **If you're using HTML5 then you can ignore this last point.** You want all data sent to you by browsers to be in UTF-8. The only reliable way to do this is to add the `accept-charset` attribute to all of your **`<form>`** tags like so:

```
<form action="somepage.php" accept-charset="UTF-8">
```

Section 48.2: Output

- If your application transmits text to other systems, they will also need to be informed of the character encoding. In PHP, you can use the `default_charset` option in `php.ini`, or manually issue the `Content-Type` MIME header yourself. This is the preferred method when targeting modern browsers.

```
header('Content-Type: text/html; charset=utf-8');
```

- If you are unable to set the response headers, then you can also set the encoding in an HTML document with HTML metadata.

    - HTML5

    ```
    <meta charset="utf-8">
    ```

    - Older versions of HTML

    ```
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    ```

Section 48.3: Data Storage and Access

> This topic specifically talks about UTF-8 and considerations for using it with a database. If you want more information about using databases in PHP then checkout this topic.

Storing Data in a MySQL Database:

- Specify the `utf8mb4` character set on all tables and text columns in your database. This makes MySQL physically store and retrieve values encoded natively in UTF-8.

> MySQL will implicitly use `utf8mb4` encoding if a `utf8mb4_*` collation is specified (without any explicit character set).

- Older versions of MySQL (< 5.5.3) do not support `utf8mb4` so you'll be forced to use `utf8`, which only supports a subset of Unicode characters.

Accessing Data in a MySQL Database:

- In your application code (e.g. PHP), in whatever DB access method you use, you'll need to set the connection charset to `utf8mb4`. This way, MySQL does no conversion from its native UTF-8 when it hands data off to your application and vice versa.
- Some drivers provide their own mechanism for configuring the connection character set, which both updates its own internal state and informs MySQL of the encoding to be used on the connection. This is usually the preferred approach.

  For Example (The same consideration regarding `utf8mb4`/`utf8` applies as above):

  - If you're using the PDO abstraction layer with PHP ≥ 5.3.6, you can specify `charset` in the DSN:

    ```
    $handle = new PDO('mysql:charset=utf8mb4');
    ```

  - If you're using mysqli, you can call `set_charset()`:

    ```
    $conn = mysqli_connect('localhost', 'my_user', 'my_password', 'my_db');

    $conn->set_charset('utf8mb4');         // object oriented style
    mysqli_set_charset($conn, 'utf8mb4'); // procedural style
    ```

  - If you're stuck with plain mysql but happen to be running PHP ≥ 5.2.3, you can call `mysql_set_charset`.

    ```
    $conn = mysql_connect('localhost', 'my_user', 'my_password');

    $conn->set_charset('utf8mb4');        // object oriented style
    mysql_set_charset($conn, 'utf8mb4'); // procedural style
    ```

  - If the database driver does not provide its own mechanism for setting the connection character set, you may have to issue a query to tell MySQL how your application expects data on the connection to be encoded: `SET NAMES 'utf8mb4'`.

Chapter 49: Unicode Support in PHP

Section 49.1: Converting Unicode characters to “\uxxxx” format using PHP

You can use the following code for going back and forward.

```
if (!function_exists('codepoint_encode')) {
    function codepoint_encode($str) {
        return substr(json_encode($str), 1, -1);
    }
}

if (!function_exists('codepoint_decode')) {
    function codepoint_decode($str) {
        return json_decode(sprintf('"%s"', $str));
    }
}
```

How to use:

```
echo "\\nUse JSON encoding / decoding\\n";
var_dump(codepoint_encode("我好"));
var_dump(codepoint_decode('\\u6211\\u597d'));
```

Output:

```
Use JSON encoding / decoding
string(12) "\\u6211\\u597d"
string(6) "我好"
```

Section 49.2: Converting Unicode characters to their numeric value and/or HTML entities using PHP

You can use the following code for going back and forward.

```
if (!function_exists('mb_internal_encoding')) {
    function mb_internal_encoding($encoding = NULL) {
        return ($from_encoding === NULL) ? iconv_get_encoding() : iconv_set_encoding($encoding);
    }
}

if (!function_exists('mb_convert_encoding')) {
    function mb_convert_encoding($str, $to_encoding, $from_encoding = NULL) {
        return iconv(($from_encoding === NULL) ? mb_internal_encoding() : $from_encoding,
$to_encoding, $str);
    }
}

if (!function_exists('mb_chr')) {
    function mb_chr($ord, $encoding = 'UTF-8') {
        if ($encoding === 'UCS-4BE') {
            return pack("N", $ord);
        } else {
            return mb_convert_encoding(mb_chr($ord, 'UCS-4BE'), $encoding, 'UCS-4BE');
        }
    }
}
```

```php
if (!function_exists('mb_ord')) {
    function mb_ord($char, $encoding = 'UTF-8') {
        if ($encoding === 'UCS-4BE') {
            list(, $ord) = (strlen($char) === 4) ? @unpack('N', $char) : @unpack('n', $char);
            return $ord;
        } else {
            return mb_ord(mb_convert_encoding($char, 'UCS-4BE', $encoding), 'UCS-4BE');
        }
    }
}

if (!function_exists('mb_htmlentities')) {
    function mb_htmlentities($string, $hex = true, $encoding = 'UTF-8') {
        return preg_replace_callback('/[\x{80}-\x{10FFFF}]/u', function ($match) use ($hex) {
            return sprintf($hex ? '&#x%X;' : '&#%d;', mb_ord($match[0]));
        }, $string);
    }
}

if (!function_exists('mb_html_entity_decode')) {
    function mb_html_entity_decode($string, $flags = null, $encoding = 'UTF-8') {
        return html_entity_decode($string, ($flags === NULL) ? ENT_COMPAT | ENT_HTML401 : $flags,
$encoding);
    }
}
```

How to use :

```php
echo "Get string from numeric DEC value\n";
var_dump(mb_chr(50319, 'UCS-4BE'));
var_dump(mb_chr(271));

echo "\nGet string from numeric HEX value\n";
var_dump(mb_chr(0xC48F, 'UCS-4BE'));
var_dump(mb_chr(0x010F));

echo "\nGet numeric value of character as DEC string\n";
var_dump(mb_ord('ď', 'UCS-4BE'));
var_dump(mb_ord('ď'));

echo "\nGet numeric value of character as HEX string\n";
var_dump(dechex(mb_ord('ď', 'UCS-4BE')));
var_dump(dechex(mb_ord('ď')));

echo "\nEncode / decode to DEC based HTML entities\n";
var_dump(mb_htmlentities('tchüß', false));
var_dump(mb_html_entity_decode('tch&#252;&#223;'));

echo "\nEncode / decode to HEX based HTML entities\n";
var_dump(mb_htmlentities('tchüß'));
var_dump(mb_html_entity_decode('tch&#xFC;&#xDF;'));
```

Output :

```
Get string from numeric DEC value
string(4) "ď"
string(2) "ď"

Get string from numeric HEX value
string(4) "ď"
string(2) "ď"
```

```
Get numeric value of character as DEC int
int(50319)
int(271)

Get numeric value of character as HEX string
string(4) "c48f"
string(3) "10f"

Encode / decode to DEC based HTML entities
string(15) "tch&#252;&#223;"
string(7) "tchüß"

Encode / decode to HEX based HTML entities
string(15) "tch&#xFC;&#xDF;"
string(7) "tchüß"
```

Section 49.3: Intl extention for Unicode support

Native string functions are mapped to single byte functions, they do not work well with Unicode. The extentions iconv and mbstring offer some support for Unicode, while the Intl-extention offers full support. Intl is a wrapper for the *facto de standard* ICU library, see http://site.icu-project.org for detailed information that is not available on http://php.net/manual/en/book.intl.php . If you can not install the extention, have a look at an alternative implementation of Intl from the Symfony framework.

ICU offers full Internationalization of which Unicode is only a smaller part. You can do transcoding easily:

```
\UConverter::transcode($sString, 'UTF-8', 'UTF-8');  // strip bad bytes against attacks
```

But, do not dismiss **iconv** just yet, consider:

```
\iconv('UTF-8', 'ASCII//TRANSLIT', "Cliënt"); // output: "Client"
```

Chapter 50: URLs

Section 50.1: Parsing a URL

To separate a URL into its individual components, use `parse_url()`:

```
$url = 'http://www.example.com/page?foo=1&bar=baz#anchor';
$parts = parse_url($url);
```

After executing the above, the contents of `$parts` would be:

```
Array
(
    [scheme] => http
    [host] => www.example.com
    [path] => /page
    [query] => foo=1&bar=baz
    [fragment] => anchor
)
```

You can also selectively return just one component of the url. To return just the querystring:

```
$url = 'http://www.example.com/page?foo=1&bar=baz#anchor';
$queryString = parse_url($url, PHP_URL_QUERY);
```

Any of the following constants are accepted: `PHP_URL_SCHEME`, `PHP_URL_HOST`, `PHP_URL_PORT`, `PHP_URL_USER`, `PHP_URL_PASS`, `PHP_URL_PATH`, `PHP_URL_QUERY` and `PHP_URL_FRAGMENT`.

To further parse a query string into key value pairs use `parse_str()`:

```
$params = [];
parse_str($queryString, $params);
```

After execution of the above, the `$params` array would be populated with the following:

```
Array
(
    [foo] => 1
    [bar] => baz
)
```

Section 50.2: Build an URL-encoded query string from an array

The `http_build_query()` will create a query string from an array or object. These strings can be appended to a URL to create a GET request, or used in a POST request with, for example, cURL.

```
$parameters = array(
    'parameter1' => 'foo',
    'parameter2' => 'bar',
);
$queryString = http_build_query($parameters);
```

`$queryString` will have the following value:

```
parameter1=foo&parameter2=bar
```

`http_build_query()` will also work with multi-dimensional arrays:

```
$parameters = array(
    "parameter3" => array(
        "sub1" => "foo",
        "sub2" => "bar",
    ),
    "parameter4" => "baz",
);
$queryString = http_build_query($parameters);
```

`$queryString` will have this value:

```
parameter3%5Bsub1%5D=foo&parameter3%5Bsub2%5D=bar&parameter4=baz
```

which is the URL-encoded version of

```
parameter3[sub1]=foo&parameter3[sub2]=bar&parameter4=baz
```

Section 50.3: Redirecting to another URL

You can use the `header()` function to instruct the browser to redirect to a different URL:

```
$url = 'https://example.org/foo/bar';
if (!headers_sent()) { // check headers - you can not send headers if they already sent
  header('Location: ' . $url);
  exit; // protects from code being executed after redirect request
} else {
  throw new Exception('Cannot redirect, headers already sent');
}
```

You can also redirect to a relative URL (this is not part of the official HTTP specification, but it does work in all browsers):

```
$url = 'foo/bar';
if (!headers_sent()) {
  header('Location: ' . $url);
  exit;
} else {
  throw new Exception('Cannot redirect, headers already sent');
}
```

If headers have been sent, you can alternatively send a `meta refresh` HTML tag.

WARNING: The meta refresh tag relies on HTML being properly processed by the client, and some will not do this. In general, it only works in web browsers. Also, consider that if headers have been sent, you may have a bug and this should trigger an exception.

You may also print a link for users to click, for clients that ignore the meta refresh tag:

```
$url = 'https://example.org/foo/bar';
if (!headers_sent()) {
  header('Location: ' . $url);
} else {
```

```
  $saveUrl = htmlspecialchars($url); // protects from browser seeing url as HTML
  // tells browser to redirect page to $saveUrl after 0 seconds
  print '<meta http-equiv="refresh" content="0; url=' . $saveUrl . '">';
  // shows link for user
  print '<p>Please continue to <a href="' . $saveUrl . '">' . $saveUrl . '</a></p>';
}
exit;
```

Chapter 51: How to break down an URL

As you code PHP you will most likely get your self in a position where you need to break down an URL into several pieces. There's obviously more than one way of doing it depending on your needs. This article will explain those ways for you so you can find what works best for you.

Section 51.1: Using parse_url()

> parse_url(): This function parses a URL and returns an associative array containing any of the various components of the URL that are present.

```
$url = parse_url('http://example.com/project/controller/action/param1/param2');

Array
(
    [scheme] => http
    [host] => example.com
    [path] => /project/controller/action/param1/param2
)
```

If you need the path separated you can use explode

```
$url = parse_url('http://example.com/project/controller/action/param1/param2');
$url['sections'] = explode('/', $url['path']);

Array
(
    [scheme] => http
    [host] => example.com
    [path] => /project/controller/action/param1/param2
    [sections] => Array
        (
            [0] =>
            [1] => project
            [2] => controller
            [3] => action
            [4] => param1
            [5] => param2
        )

)
```

If you need the last part of the section you can use end() like this:

```
$last = end($url['sections']);
```

If the URL contains GET vars you can retrieve those as well

```
$url = parse_url('http://example.com?var1=value1&var2=value2');

Array
(
    [scheme] => http
    [host] => example.com
    [query] => var1=value1&var2=value2
```

```
)
```

If you wish to break down the query vars you can use parse_str() like this:

```
$url = parse_url('http://example.com?var1=value1&var2=value2');
parse_str($url['query'], $parts);

Array
(
    [var1] => value1
    [var2] => value2
)
```

Section 51.2: Using explode()

> explode(): Returns an array of strings, each of which is a substring of string formed by splitting it on boundaries formed by the string delimiter.

This function is pretty much straight forward.

```
$url = "http://example.com/project/controller/action/param1/param2";
$parts = explode('/', $url);

Array
(
    [0] => http:
    [1] =>
    [2] => example.com
    [3] => project
    [4] => controller
    [5] => action
    [6] => param1
    [7] => param2
)
```

You can retrieve the last part of the URL by doing this:

```
$last = end($parts);
// Output: param2
```

You can also navigate inside the array by using sizeof() in combination with a math operator like this:

```
echo $parts[sizeof($parts)-2];
// Output: param1
```

Section 51.3: Using basename()

> basename(): Given a string containing the path to a file or directory, this function will return the trailing name component.

This function will return only the last part of an URL

```
$url = "http://example.com/project/controller/action/param1/param2";
```

```php
$parts = basename($url);
// Output: param2
```

If your URL has more stuff to it and what you need is the dir name containing the file you can use it with dirname() like this:

```php
$url = "http://example.com/project/controller/action/param1/param2/index.php";
$parts = basename(dirname($url));
// Output: param2
```

Chapter 52: Object Serialization

Section 52.1: Serialize / Unserialize

`serialize()` returns a string containing a byte-stream representation of any value that can be stored in PHP. `unserialize()` can use this string to recreate the original variable values.

To serialize an object

```
serialize($object);
```

To Unserialize an object

```
unserialize($object)
```

Example

```
$array = array();
$array["a"] = "Foo";
$array["b"] = "Bar";
$array["c"] = "Baz";
$array["d"] = "Wom";

$serializedArray = serialize($array);
echo $serializedArray; //output:
a:4:{s:1:"a";s:3:"Foo";s:1:"b";s:3:"Bar";s:1:"c";s:3:"Baz";s:1:"d";s:3:"Wom";}
```

Section 52.2: The Serializable interface

Introduction

> Classes that implement this interface no longer support `__sleep()` and `__wakeup()`. The method serialize is called whenever an instance needs to be serialized. This does not invoke `__destruct()` or has any other side effect unless programmed inside the method. When the data is `unserialized` the class is known and the appropriate `unserialize()` method is called as a constructor instead of calling `__construct()`. If you need to execute the standard constructor you may do so in the method.

Basic usage

```
class obj implements Serializable {
    private $data;
    public function __construct() {
        $this->data = "My private data";
    }
    public function serialize() {
        return serialize($this->data);
    }
    public function unserialize($data) {
        $this->data = unserialize($data);
    }
    public function getData() {
        return $this->data;
    }
}
```

```php
$obj = new obj;
$ser = serialize($obj);

var_dump($ser); // Output: string(38) "C:3:"obj":23:{s:15:"My private data";}"

$newobj = unserialize($ser);

var_dump($newobj->getData()); // Output: string(15) "My private data"
```

Chapter 53: Serialization

Parameter	Details
value	The value to be serialized. serialize() handles all types, except the resource-type. You can even serialize() arrays that contain references to itself. Circular references inside the array/object you are serializing will also be stored. Any other reference will be lost. When serializing objects, PHP will attempt to call the member function __sleep() prior to serialization. This is to allow the object to do any last minute clean-up, etc. prior to being serialized. Likewise, when the object is restored using unserialize() the __wakeup() member function is called. Object's private members have the class name prepended to the member name; protected members have a '*' prepended to the member name. These prepended values have null bytes on either side.

Section 53.1: Serialization of different types

Generates a storable representation of a value.

This is useful for storing or passing PHP values around without losing their type and structure.

To make the serialized string into a PHP value again, use **unserialize()**.

Serializing a string

```
$string = "Hello world";
echo serialize($string);

// Output:
// s:11:"Hello world";
```

Serializing a double

```
$double = 1.5;
echo serialize($double);

// Output:
// d:1.5;
```

Serializing a float

Float get serialized as doubles.

Serializing an integer

```
$integer = 65;
echo serialize($integer);

// Output:
// i:65;
```

Serializing a boolean

```
$boolean = true;
echo serialize($boolean);

// Output:
// b:1;

$boolean = false;
echo serialize($boolean);

// Output:
// b:0;
```

Serializing null

```php
$null = null;
echo serialize($null);

// Output:
// N;
```

Serializing an array

```php
$array = array(
    25,
    'String',
    'Array'=> ['Multi Dimension','Array'],
    'boolean'=> true,
    'Object'=>$obj, // $obj from above Example
    null,
    3.445
);


// This will throw Fatal Error
// $array['function'] = function() { return "function"; };

echo serialize($array);

// Output:
// a:7:{i:0;i:25;i:1;s:6:"String";s:5:"Array";a:2:{i:0;s:15:"Multi
Dimension";i:1;s:5:"Array";}s:7:"boolean";b:1;s:6:"Object";O:3:"abc":1:{s:1:"i";i:1;}i:2;N;i:3;d:3.44
4999999999998;}
```

Serializing an object

You can also serialize Objects.

When serializing objects, PHP will attempt to call the member function **__sleep()** prior to serialization. This is to allow the object to do any last minute clean-up, etc. prior to being serialized. Likewise, when the object is restored using **unserialize()** the **__wakeup()** member function is called.

```php
class abc {
    var $i = 1;
    function foo() {
        return 'hello world';
    }
}

$object = new abc();
echo serialize($object);

// Output:
// O:3:"abc":1:{s:1:"i";i:1;}
```

Note that Closures cannot be serialized:

```php
$function = function () { echo 'Hello World!'; };
$function(); // prints "hello!"

$serializedResult = serialize($function);  // Fatal error: Uncaught exception 'Exception' with
message 'Serialization of 'Closure' is not allowed'
```

Section 53.2: Security Issues with unserialize

Using `unserialize` function to unserialize data from user input can be dangerous.

A Warning from php.net

> **Warning** Do not pass untrusted user input to unserialize(). Unserialization can result in code being loaded and executed due to object instantiation and autoloading, and a malicious user may be able to exploit this. Use a safe, standard data interchange format such as JSON (via json_decode() and json_encode()) if you need to pass serialized data to the user.

Possible Attacks

- PHP Object Injection

PHP Object Injection

PHP Object Injection is an application level vulnerability that could allow an attacker to perform different kinds of malicious attacks, such as Code Injection, SQL Injection, Path Traversal and Application Denial of Service, depending on the context. The vulnerability occurs when user-supplied input is not properly sanitized before being passed to the unserialize() PHP function. Since PHP allows object serialization, attackers could pass ad-hoc serialized strings to a vulnerable unserialize() call, resulting in an arbitrary PHP object(s) injection into the application scope.

In order to successfully exploit a PHP Object Injection vulnerability two conditions must be met:

- The application must have a class which implements a PHP magic method (such as `__wakeup` or `__destruct`) that can be used to carry out malicious attacks, or to start a "POP chain".
- All of the classes used during the attack must be declared when the vulnerable `unserialize()` is being called, otherwise object autoloading must be supported for such classes.

Example 1 - Path Traversal Attack

The example below shows a PHP class with an exploitable `__destruct` method:

```
class Example1
{
   public $cache_file;

   function __construct()
   {
      // some PHP code...
   }

   function __destruct()
   {
      $file = "/var/www/cache/tmp/{$this->cache_file}";
      if (file_exists($file)) @unlink($file);
   }
}

// some PHP code...

$user_data = unserialize($_GET['data']);

// some PHP code...
```

In this example an attacker might be able to delete an arbitrary file via a Path Traversal attack, for e.g. requesting the following URL:

```
http://testsite.com/vuln.php?data=O:8:"Example1":1:{s:10:"cache_file";s:15:"../../index.php";}
```

Example 2 - Code Injection attack

The example below shows a PHP class with an exploitable __wakeup method:

```
class Example2
{
   private $hook;

   function __construct()
   {
      // some PHP code...
   }

   function __wakeup()
   {
      if (isset($this->hook)) eval($this->hook);
   }
}

// some PHP code...

$user_data = unserialize($_COOKIE['data']);

// some PHP code...
```

In this example an attacker might be able to perform a Code Injection attack by sending an HTTP request like this:

```
GET /vuln.php HTTP/1.0
Host: testsite.com
Cookie:
data=O%3A8%3A%22Example2%22%3A1%3A%7Bs%3A14%3A%22%00Example2%00hook%22%3Bs%3A10%3A%22phpinfo%28%29%
3B%22%3B%7D
Connection: close
```

Where the cookie parameter "data" has been generated by the following script:

```
class Example2
{
   private $hook = "phpinfo();";
}

print urlencode(serialize(new Example2));
```

Chapter 54: Closure

Section 54.1: Basic usage of a closure

A **closure** is the PHP equivalent of an anonymous function, eg. a function that does not have a name. Even if that is technically not correct, the behavior of a closure remains the same as a function's, with a few extra features.

A closure is nothing but an object of the Closure class which is created by declaring a function without a name. For example:

```
<?php

$myClosure = function() {
    echo 'Hello world!';
};

$myClosure(); // Shows "Hello world!"
```

Keep in mind that `$myClosure` is an instance of `Closure` so that you are aware of what you can truly do with it (cf. http://fr2.php.net/manual/en/class.closure.php)

The classic case you would need a Closure is when you have to give a `callable` to a function, for instance usort.

Here is an example where an array is sorted by the number of siblings of each person:

```
<?php

$data = [
    [
        'name' => 'John',
        'nbrOfSiblings' => 2,
    ],
    [
        'name' => 'Stan',
        'nbrOfSiblings' => 1,
    ],
    [
        'name' => 'Tom',
        'nbrOfSiblings' => 3,
    ]
];

usort($data, function($e1, $e2) {
    if ($e1['nbrOfSiblings'] == $e2['nbrOfSiblings']) {
        return 0;
    }

    return $e1['nbrOfSiblings'] < $e2['nbrOfSiblings'] ? -1 : 1;
});

var_dump($data); // Will show Stan first, then John and finally Tom
```

Section 54.2: Using external variables

It is possible, inside a closure, to use an external variable with the special keyword **use**. For instance:

```
<?php
```

```php
$quantity = 1;

$calculator = function($number) use($quantity) {
    return $number + $quantity;
};

var_dump($calculator(2)); // Shows "3"
```

You can go further by creating "dynamic" closures. It is possible to create a function that returns a specific calculator, depending on the quantity you want to add. For example:

```php
<?php

function createCalculator($quantity) {
    return function($number) use($quantity) {
        return $number + $quantity;
    };
}

$calculator1 = createCalculator(1);
$calculator2 = createCalculator(2);

var_dump($calculator1(2)); // Shows "3"
var_dump($calculator2(2)); // Shows "4"
```

Section 54.3: Basic closure binding

As seen previously, a closure is nothing but an instance of the Closure class, and different methods can be invoked on them. One of them is `bindTo`, which, given a closure, will return a new one that is bound to a given object. For example:

```php
<?php

$myClosure = function() {
    echo $this->property;
};

class MyClass
{
    public $property;

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

$myInstance = new MyClass('Hello world!');
$myBoundClosure = $myClosure->bindTo($myInstance);

$myBoundClosure(); // Shows "Hello world!"
```

Section 54.4: Closure binding and scope

Let's consider this example:

```php
<?php
```

```php
$myClosure = function() {
    echo $this->property;
};

class MyClass
{
    public $property;

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

$myInstance = new MyClass('Hello world!');
$myBoundClosure = $myClosure->bindTo($myInstance);

$myBoundClosure(); // Shows "Hello world!"
```

Try to change the `property` visibility to either **`protected`** or **`private`**. You get a fatal error indicating that you do not have access to this property. Indeed, even if the closure has been bound to the object, the scope in which the closure is invoked is not the one needed to have that access. That is what the second argument of `bindTo` is for.

The only way for a property to be accessed if it's **`private`** is that it is accessed from a scope that allows it, ie. the class's scope. In the just previous code example, the scope has not been specified, which means that the closure has been invoked in the same scope as the one used where the closure has been created. Let's change that:

```php
<?php

$myClosure = function() {
    echo $this->property;
};

class MyClass
{
    private $property; // $property is now private

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

$myInstance = new MyClass('Hello world!');
$myBoundClosure = $myClosure->bindTo($myInstance, MyClass::class);

$myBoundClosure(); // Shows "Hello world!"
```

As just said, if this second parameter is not used, the closure is invoked in the same context as the one used where the closure has been created. For example, a closure created inside a method's class which is invoked in an object context will have the same scope as the method's:

```php
<?php

class MyClass
{
    private $property;

    public function __construct($propertyValue)
```

```
    {
        $this->property = $propertyValue;
    }

    public function getDisplayer()
      {
        return function() {
              echo $this->property;
        };
      }
}

$myInstance = new MyClass('Hello world!');

$displayer = $myInstance->getDisplayer();
$displayer(); // Shows "Hello world!"
```

Section 54.5: Binding a closure for one call

Since PHP7, it is possible to bind a closure just for one call, thanks to the `call` method. For instance:

```
<?php

class MyClass
{
    private $property;

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

$myClosure = function() {
    echo $this->property;
};

$myInstance = new MyClass('Hello world!');

$myClosure->call($myInstance); // Shows "Hello world!"
```

As opposed to the `bindTo` method, there is no scope to worry about. The scope used for this call is the same as the one used when accessing or invoking a property of `$myInstance`.

Section 54.6: Use closures to implement observer pattern

In general, an observer is a class with a specific method being called when an action on the observed object occurs. In certain situations, closures can be enough to implement the observer design pattern.

Here is a detailed example of such an implementation. Let's first declare a class whose purpose is to notify observers when its property is changed.

```
<?php

class ObservedStuff implements SplSubject
{
    protected $property;
    protected $observers = [];
```