CSE3026: Web Application Development PHP

Scott Uk-Jin Lee

5.1: Server-Side Basics

URLs and web servers

`http://`*server*`/`*path*`/`*file*

- usually when you type a URL in your browser:
  - your computer looks up the server's IP address using DNS
  - your browser connects to that IP address and requests the given file
  - the web server software (e.g. Apache) grabs that file from the server's local file system, and sends back its contents to you

- some URLs actually specify *programs* that the web server should run, and then send their output back to you as the result:

  ```
  http://selab.hanyang.ac.kr/courses/cse326/2017/.../quote.php
  ```

  - the above URL tells the server `selab.hanyang.ac.kr` to run the program `quote.php` and send back its output

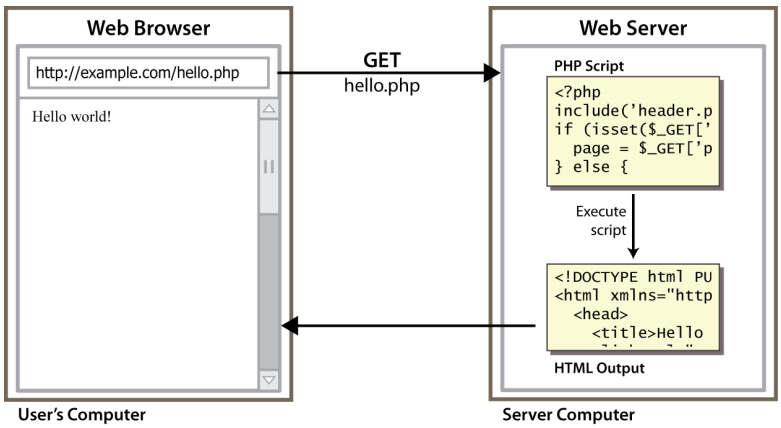
Server-Side web programming

- server-side pages are programs written using one of many web programming languages/frameworks
  - examples: PHP, Java/JSP, Ruby on Rails, ASP.NET, Python, Perl
- the web server contains software that allows it to run those programs and send back their output
- each language/framework has its pros and cons
  - we use PHP for server-side programming in this textbook

What is PHP?

- **PHP** stands for "PHP Hypertext Preprocessor"
- a server-side scripting language
- used to make web pages dynamic:
  - provide different content depending on context
  - interface with other services: database, e-mail, etc
  - authenticate users
  - process form information
- PHP code can be embedded in HTML code

Lifecycle of a PHP web request



- browser requests a `.html` file (**static content**): server just sends that file
- browser requests a `.php` file (**dynamic content**): server reads it, runs any script code inside it, then sends result across the network
  - script produces output that becomes the response sent back

Why PHP?

There are many other options for server-side languages: Ruby on Rails, JSP, ASP.NET, etc. Why choose PHP?

- free and open source: anyone can run a PHP-enabled server free of charge
- **compatible:** supported by most popular web servers
- **simple:** lots of built-in functionality; familiar syntax
- **available:** can easily be installed on your own computer and installed on most commercial web hosts
- **well-documented:** type `php.net/`*functionName* in browser Address bar to get docs for any function

Hello, World!

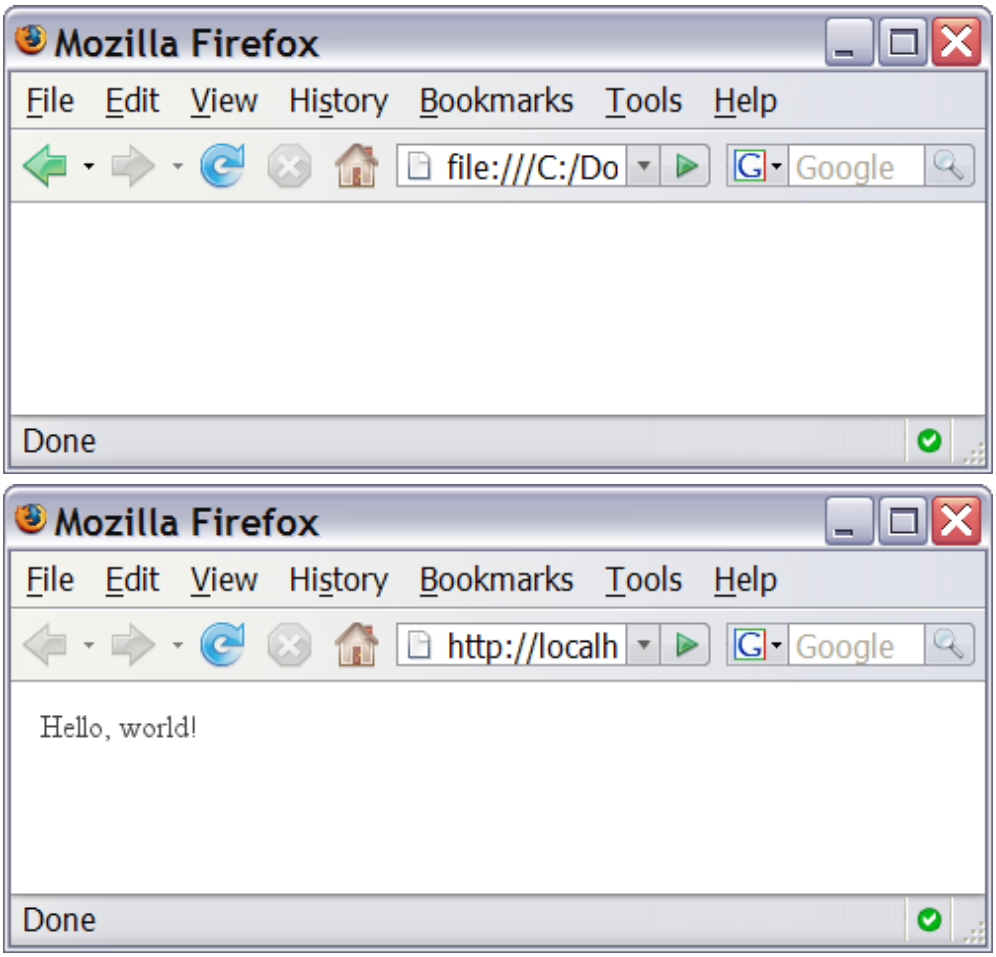
The following contents could go into a file `hello.php`:

```
<?php
print "Hello, world!";
?>
```

```
Hello, world!
```

- a block or file of PHP code begins with `<?php` and ends with `?>`
- PHP statements, function declarations, etc. appear between these endpoints

Viewing PHP output



- you can't view your `.php` page on your local hard drive; you'll either see nothing or see the PHP source code
- if you upload the file to a PHP-enabled web server, requesting the `.php` file will run the program and send you back its output

5.2: PHP Basic Syntax

- 5.1: Server-Side Basics
- **5.2: PHP Basic Syntax**
- 5.3: Embedded PHP
- 5.4: Advanced PHP Syntax

PHP syntax template

```
HTML content

  <?php
    PHP code
  ?>

HTML content

  <?php
    PHP code
  ?>

HTML content ...
```

- any contents of a `.php` file between `<?php` and `?>` are executed as PHP code
- all other contents are output as pure HTML
- can switch back and forth between HTML and PHP "modes"

Comments

```
# single-line comment

// single-line comment

/*
multi-line comment
*/
```

- like Java, but # is also allowed
  - a lot of PHP code uses # comments instead of `//`
  - we recommend # and will use it in our examples

Console output: `print`

```
print "text";

print "Hello, World!\n";
print "Escape \"chars\" are the SAME as in Java!\n";

print "You can have
line breaks in a string.";

print 'A string can use "single-quotes".  It\'s cool!';
```

```
Hello, World! Escape "chars" are the SAME as in Java! You can have line breaks in a string.
A string can use "single-quotes". It's cool!
```

- some PHP programmers use the equivalent `echo` instead of `print`

Arithmetic operators

- `+ - * / %`
  `. ++ --`
  `= += -= *= /= %= .=`
- many operators auto-convert types: `5 + "7"` is `12`

Math operations

```
$a = 3;
$b = 4;
$c = sqrt(pow($a, 2) + pow($b, 2));
```

abs	ceil	cos	floor	log	log10	max
min	pow	rand	round	sin	sqrt	tan

math functions

M_PI	M_E	M_LN2

math constants

- the syntax for method calls, parameters, returns is the same as Java

Variables

```
$name = expression;

$user_name = "PinkHeartLuvr78";
$age = 16;
$drinking_age = $age + 5;
$this_class_rocks = TRUE;
```

- names are case sensitive; separate multiple words with _
- names always begin with $, on both declaration and usage
- implicitly declared by assignment (type is not written; a "loosely typed" language)

Types

- basic types: `int`, `float`, `boolean`, `string`, `array`, `object`, `NULL`
  - test what type a variable is with `is_`*type* functions, e.g. `is_string`
  - `gettype` function returns a variable's type as a string (not often needed)
- PHP converts between types automatically in many cases:
  - `string` → `int` auto-conversion on `+` (`"1" + 1 == 2`)
  - `int` → `float` auto-conversion on `/` (`3 / 2 == 1.5`)
- type-cast with `(`*type*`)`:
  - `$age = (int) "21";`

`int` and `float` types

```
$a = 7 / 2;                 # float: 3.5
$b = (int) $a;              # int: 3
$c = round($a);             # float: 4.0
$d = "123";                 # string: "123"
$e = (int) $d;              # int: 123
```

- `int` for integers and `float` for reals
- division between two `int` values can produce a `float`

`String` type

```
$favorite_food = "Ethiopian";
print $favorite_food[2];              # h
```

- zero-based indexing using bracket notation
- string concatenation operator is `.` (period), not `+`
  - `5 + "2 turtle doves"` produces `7`
  - `5 . "2 turtle doves"` produces `"52 turtle doves"`
- can be specified with `""` or `''`

Interpreted strings

```
$age = 16;
print "You are " . $age . " years old.\n";
print "You are $age years old.\n";     # You are 16 years old.
```

- strings inside `" "` are **interpreted**
  - variables that appear inside them will have their values inserted into the string
- strings inside `' '` are *not* interpreted:

```
print 'You are $age years old.\n';     # You are $age years old.\n
```

- if necessary to avoid ambiguity, can enclose variable in `{}`:

```
print "Today is your $ageth birthday.\n";      # $ageth not found
print "Today is your {$age}th birthday.\n";
```

String functions

```
# index  0123456789012345
$name = "Stefanie Hatcher";
$length = strlen($name);               # 16
$cmp = strcmp($name, "Brian Le");      # > 0
$index = strpos($name, "e");           # 2
$first = substr($name, 9, 5);          # "Hatch"
$name = strtoupper($name);             # "STEFANIE HATCHER"
```

Name	Java Equivalent
strlen	length
strpos	indexOf
substr	substring
strtolower, strtoupper	toLowerCase, toUpperCase
trim	trim
explode, implode	split, join
strcmp	compareTo

bool (Boolean) type

```
$feels_like_summer = FALSE;
$php_is_rad = TRUE;

$student_count = 217;
$nonzero = (bool) $student_count;      # TRUE
```

- the following values are considered to be FALSE (all others are TRUE):
  - 0 and 0.0
  - "", "0", and NULL (includes unset variables)
  - arrays with 0 elements
- can cast to boolean using (bool)
- FALSE prints as an empty string (no output); TRUE prints as a 1

- TRUE and FALSE keywords are case insensitive

`for` loop

```
for (initialization; condition; update) {
  statements;
}
```

```
for ($i = 0; $i < 10; $i++) {
  print "$i squared is " . $i * $i . ".\n";
}
```

`if/else` statement

```
if (condition) {
  statements;
} elseif (condition) {
  statements;
} else {
  statements;
}
```

- NOTE: although `elseif` keyword is much more common, `else if` is also supported

`while` loop (same as Java)

```
while (condition) {
  statements;
}
```

```
do {
  statements;
} while (condition);
```

- `break` and `continue` keywords also behave as in Java

5.3: Embedded PHP

- 5.1: Server-Side Basics
- 5.2: PHP Basic Syntax
- **5.3: Embedded PHP**
- 5.4: Advanced PHP Syntax

Printing HTML tags in PHP = bad style

```
<?php
print "<!DOCTYPE html>\n";
print "<html>\n";
print "  <head>\n";
print "    <title>Geneva's web page</title>\n";
...
for ($i = 1; $i <= 10; $i++) {
  print "<p class=\"count\"> I can count to $i! </p>\n";
}
?>
```

- printing HTML tags with `print` statements is bad style and error-prone:
  - must quote the HTML and escape special characters, e.g. `\"`
- but without `print`, how do we insert dynamic content into the page?

PHP expression blocks

```
<?= expression ?>
```

```
<h2> The answer is <?= 6 * 7 ?> </h2>
```

The answer is 42

- **PHP expression block**: evaluates and embeds an expression's value into HTML
- `<?= expr ?>` is equivalent to `<?php print expr; ?>`

Expression block example

```
<!DOCTYPE html >
<html>
  <head><title>CSE 3026s: Embedded PHP</title></head>
  <body>
    <?php for ($i = 99; $i >= 1; $i--) { ?>
      <p> <?= $i ?> bottles of beer on the wall, <br />
          <?= $i ?> bottles of beer. <br />
          Take one down, pass it around, <br />
          <?= $i - 1 ?> bottles of beer on the wall. </p>
    <?php } ?>
  </body>
</html>
```

Common errors: unclosed braces, missing = sign

```
  <body>
    <p>Watch how high I can count:
      <?php for ($i = 1; $i <= 10; $i++) { ?>
        <? $i ?>
    </p>
  </body>
</html>
```

- `</body>` and `</html>` above are inside the `for` loop, which is never closed
- if you forget to close your braces, you'll see an error about 'unexpected `$end`'
- if you forget = in `<?=`, the expression does not produce any output

Complex expression blocks

```
<body>
  <?php for ($i = 1; $i <= 3; $i++) { ?>
    <h<?= $i ?>>This is a level <?= $i ?> heading.</h<?= $i ?>>
  <?php } ?>
</body>
```

- expression blocks can even go inside HTML tags and attributes

6.1: Parameterized Pages

- 5.1: Server-Side Basics
- 5.2: PHP Basic Syntax
- 5.3: Embedded PHP
- 5.4: Advanced PHP Syntax
- **6.1: Parameterized Pages**

Query strings and parameters

```
URL?name=value&name=value...
```

```
http://www.google.com/search?q=Obama

http://example.com/student_login.php?username=lee&id=1234567
```

- **query string**: a set of parameters passed from a browser to a web server
  - often passed by placing name/value pairs at the end of a URL
  - above, parameter `username` has value `lee`, and `sid` has value `1234567`
- PHP code on the server can examine and utilize the value of parameters
- a way for PHP code to produce different output based on values passed by the user

Query parameters: `$_GET`

```
$user_name = $_GET["username"];
$id_number = (int) $_GET["id"];
$eats_meat = FALSE;
if (isset($_GET["meat"])) {
  $eats_meat = TRUE;
}
```

- `$_GET["`*parameter name*`"]` returns an HTTP GET parameter's value as a string
- parameters specified as `http://....?`*name*`=`*value*`&`*name*`=`*value* are GET parameters
- can test whether a given parameter was passed with `isset`

Example: Exponents

```
$base = $_GET["base"];
$exp = $_GET["exponent"];
$result = pow($base, $exp);
print "$base ^ $exp = $result";
```

```
exponent.php?base=3&exponent=4
```

```
3 ^ 4 = 81
```

Example: Print all parameters

```
<?php foreach ($_GET as $param => $value) { ?>
  <p>Parameter <?= $param ?> has value <?= $value ?></p>
<?php } ?>
```

```
print_params.php?name=Scott+Lee&sid=1234567
```

```
Parameter name has value Scott Lee

Parameter sid has value 1234567
```

- or call `print_r` or `var_dump` on `$_GET` for debugging

5.4: Advanced PHP Syntax

- 5.1: Server-Side Basics
- 5.2: PHP Basic Syntax
- 5.3: Embedded PHP
- **5.4: Advanced PHP Syntax**
- 6.1: Parameterized Pages

Arrays

```
$name = array();                                # create
$name = array(value0, value1, ..., valueN);

$name[index]                                    # get element value
$name[index] = value;                           # set element value
$name[] = value;                                # append
```

```
$a = array();      # empty array (length 0)
$a[0] = 23;        # stores 23 at index 0 (length 1)
$a2 = array("some", "strings", "in", "an", "array");
$a2[] = "Ooh!";    # add string to end (at index 5)
```

- to append, use bracket notation without specifying an index
- element type is not specified; can mix types

Array functions

function name(s)	description
count	number of elements in the array
print_r	print array's contents
array_pop, array_push, array_shift, array_unshift	using array as a stack/queue
in_array, array_search, array_reverse, sort, rsort, shuffle	searching and reordering
array_fill, array_merge, array_intersect, array_diff, array_slice, range	creating, filling, filtering
array_sum, array_product, array_unique, array_filter, array_reduce	processing elements

Array function example

```
$tas = array("MD", "BH", "KK", "HM", "JP");
for ($i = 0; $i < count($tas); $i++) {
  $tas[$i] = strtolower($tas[$i]);
}                                  # ("md", "bh", "kk", "hm", "jp")
$morgan = array_shift($tas);       # ("bh", "kk", "hm", "jp")
array_pop($tas);                   # ("bh", "kk", "hm")
array_push($tas, "ms");            # ("bh", "kk", "hm", "ms")
array_reverse($tas);               # ("ms", "hm", "kk", "bh")
sort($tas);                        # ("bh", "hm", "kk", "ms")
$best = array_slice($tas, 1, 2);   # ("hm", "kk")
```

- the array in PHP replaces many other collections in Java
  - list, stack, queue, set, map, ...

The `foreach` loop

```
foreach ($array as $variableName) {
  ...
}
```

```
$stooges = array("Larry", "Moe", "Curly", "Shemp");
for ($i = 0; $i < count($stooges); $i++) {
  print "Moe slaps {$stooges[$i]}\n";
}
foreach ($stooges as $stooge) {
  print "Moe slaps $stooge\n";  # even himself!
}
```

- a convenient way to loop over each element of an array without indexes

Splitting/joining strings

```
$array = explode(delimiter, string);
$string = implode(delimiter, array);
```

```
$s  = "CSE 3026";
$a  = explode(" ", $s);      # ("CSE", "3026")
$s2 = implode("...", $a);    # "CSE...3026"
```

- `explode` and `implode` convert between strings and arrays
- for more complex string splitting, you can use **regular expressions** (later)

Example with `explode`

s *contents of input file* `names.txt`

```
Martin D Stepp
Jessica K Miller
Victoria R Kirst
```

```
foreach (file("names.txt") as $name) {
  $tokens = explode(" ", $name);
  ?>
  <p> author: <?= $tokens[2] ?>, <?= $tokens[0] ?> </p>
  <?php
}
```

author: Stepp, Marty

author: Miller, Jessica

author: Kirst, Victoria

Functions

```
function name(parameterName, ..., parameterName) {
  statements;
}
```

```
function bmi($weight, $height) {
  $result = 703 * $weight / $height / $height;
  return $result;
}
```

- parameter types and return types are not written
- a function with no `return` statements is implicitly "void"
- can be declared in any PHP block, at start/end/middle of code

Calling functions

```
name(expression, ..., expression);
```

```
$w = 163;  # pounds
$h = 70;   # inches
$my_bmi = bmi($w, $h);
```

- if the wrong number of parameters are passed, it's an error

Variable scope: global and local vars

```
$school = "HYU";                    # global
...

function downgrade() {
  global $school;
  $suffix = "(Wisconsin)";          # local

  $school = "$school $suffix";
  print "$school\n";
}
```

- variables declared in a function are **local** to that function; others are **global**
- if a function wants to use a global variable, it must have a `global` statement
  - but don't abuse this; mostly you should use parameters

Default parameter values

```
function name(parameterName = value, ..., parameterName = value) {
  statements;
}
```

```
function print_separated($str, $separator = ", ") {
  if (strlen($str) > 0) {
    print $str[0];
    for ($i = 1; $i < strlen($str); $i++) {
      print $separator . $str[$i];
    }
  }
}
```

```
print_separated("hello");         # h, e, l, l, o
print_separated("hello", "-");    # h-e-l-l-o
```

- if no value is passed, the default will be used (defaults must come last)

NULL

```
$name = "Victoria";
$name = NULL;
if (isset($name)) {
  print "This line isn't going to be reached.\n";
}
```

- a variable is `NULL` if
  - it has not been set to any value (undefined variables)
  - it has been assigned the constant `NULL`
  - it has been deleted using the `unset` function
- can test if a variable is `NULL` using the `isset` function
- `NULL` prints as an empty string (no output)

5.4: PHP File Input

- 5.1: Server-Side Basics
- 5.2: PHP Basic Syntax
- 5.3: Embedded PHP
- **5.4: Advanced PHP Syntax**

PHP file I/O functions

function name(s)	category
file, **file_get_contents**, **file_put_contents**	reading/writing entire files
basename, file_exists, filesize, fileperms, filemtime, is_dir, is_readable, is_writable, disk_free_space	asking for information
copy, rename, unlink, chmod, chgrp, chown, mkdir, rmdir	manipulating files and directories
glob, **scandir**	reading directories

Reading/writing files

contents of foo.txt	file("foo.txt")	file_get_contents("foo.txt")
Hello how r u? I'm fine	array("Hello\n", # 0 "how r u?\n", # 1 "\n", # 2 "I'm fine\n" # 3)	"Hello\n how r u?\n # a single \n # string I'm fine\n"

- file function returns lines of a file as an array (\n at end of each)
- file_get_contents returns entire contents of a file as a single string
  - file_put_contents writes a string into a file

Reading/writing an entire file

```
# reverse a file
$text = file_get_contents("poem.txt");
$text = strrev($text);
file_put_contents("poem.txt", $text);
```

- file_get_contents returns entire contents of a file as a string
  - if the file doesn't exist, you will get a warning and an empty return string
- file_put_contents writes a string into a file, replacing its old contents
  - if the file doesn't exist, it will be created

Appending to a file

```
# add a line to a file
$new_text = "P.S. ILY, GTG TTYL!~";
file_put_contents("poem.txt", $new_text, FILE_APPEND);
```

old contents	new contents
Roses are red, Violets are blue. All my base, Are belong to you.	Roses are red, Violets are blue. All my base, Are belong to you. P.S. ILY, GTG TTYL!~

- `file_put_contents` can be called with an optional third parameter to append (add to the end) rather than overwrite

The `file` function

```
# display lines of file as a bulleted list
$lines = file("todolist.txt");
foreach ($lines as $line) {   # for ($i = 0; $i < count($lines); $i++)
  print "<li>$line</li>\n";
}
```

- `file` returns the lines of a file as an array of strings
- each ends with `\n` ; to strip it, use an optional second parameter:

```
$lines = file("todolist.txt", FILE_IGNORE_NEW_LINES);
```

- common idiom: `foreach` or `for` loop over lines of file

Unpacking an array: `list`

```
list($var1, ..., $varN) = array;
```

contents of input file `personal.txt`

```
Marty Stepp
(206) 685-2181
570-86-7326
```

```
list($name, $phone, $ssn) = file("personal.txt");
...
list($area_code, $prefix, $suffix) = explode(" ", $phone);
```

- the odd `list` function "unpacks" an array into a set of variables you declare
- when you know a file's exact length/format, use `file` and `list` to unpack it

Reading directories

function	description
`glob`	returns an array of all file names that match a given pattern (returns a file path and name, such as `"foo/bar/myfile.txt"`)
`scandir`	returns an array of all file names in a given directory (returns just the file names, such as `"myfile.txt"`)

- `glob` can accept a general path with the `*` wildcard character (more powerful)

glob example

```
# reverse all poems in the poetry directory
$poems = glob("poetry/poem*.dat");
foreach ($poems as $poemfile) {
  $text = file_get_contents($poemfile);
  file_put_contents($poemfile, strrev($text));
  print "<p>I just reversed " . basename($poemfile) . "</p>\n";
}
```

- glob can match a "wildcard" path with the * character
  - glob("foo/bar/*.doc") returns all .doc files in the foo/bar subdirectory
  - glob("food*") returns all files whose names begin with "food"
- the basename function strips any leading directory from a file path
  - basename("foo/bar/baz.txt") returns "baz.txt"

scandir example

```
<ul>
  <?php foreach (scandir("taxes/old") as $filename) { ?>
    <li><?= $filename ?></li>
  <?php } ?>
</ul>
```

- .
- ..
- 2007_w2.pdf
- 2006_1099.doc

- scandir includes current directory (".") and parent ("..") in the array
- don't need basename with scandir; returns file names only without directory

Why use classes and objects?

- PHP is a primarily procedural language
- small programs are easily written without adding any classes or objects
- larger programs, however, become cluttered with so many disorganized functions
- grouping *related data and behavior* into objects helps manage size and complexity

Constructing and using objects

```
# construct an object
$name = new ClassName(parameters);

# access an object's field (if the field is public)
$name->fieldName

# call an object's method
$name->methodName(parameters);
```

```
$zip = new ZipArchive();
$zip->open("moviefiles.zip");
$zip->extractTo("images/");
$zip->close();
```

- the above code unzips a file
- test whether a class is installed with `class_exists`

Object example: Fetch file from web

```
# create an HTTP request to fetch student.php
$req = new HttpRequest("student.php", HttpRequest::METH_GET);
$params = array("first_name" => $fname, "last_name" => $lname);
$req->addPostFields($params);

# send request and examine result
$req->send();
$http_result_code = $req->getResponseCode();    # 200 means OK
print "$http_result_code\n";
print $req->getResponseBody();
```

- PHP's `HttpRequest` object can fetch a document from the web

Class declaration syntax

```
class ClassName {
  # fields - data inside each object
  public $name;     # public field
  private $name;    # private field

  # constructor - initializes each object's state
  public function __construct(parameters) {
    statement(s);
  }

  # method - behavior of each object
  public function name(parameters) {
    statements;
  }
}
```

- inside a constructor or method, refer to the current object as `$this`

Class example

```
<?php
class Point {
  public $x;
  public $y;

  # equivalent of a Java constructor
  public function __construct($x, $y) {
    $this->x = $x;
    $this->y = $y;
  }

  public function distance($p) {
    $dx = $this->x - $p->x;
    $dy = $this->y - $p->y;
    return sqrt($dx * $dx + $dy * $dy);
  }

  # equivalent of Java's toString method
  public function __toString() {
    return "(" . $this->x . ", " . $this->y . ")";
  }
}
?>
```

Class usage example

```
<?php
# this code could go into a file named use_point.php
include("Point.php");

$p1 = new Point(0, 0);
$p2 = new Point(4, 3);
print "Distance between $p1 and $p2 is " . $p1->distance($p2) . "\n\n";

var_dump($p2);   # var_dump prints detailed state of an object
?>
```

```
Distance between (0, 0) and (4, 3) is 5

object(Point)[2]
  public 'x' => int 4
  public 'y' => int 3
```

- `$p1` and `$p2` are references to `Point` objects

Basic inheritance

```
class ClassName extends ClassName {
    ...
}
```

```
class Point3D extends Point {
  public $z;

  public function __construct($x, $y, $z) {
    parent::__construct($x, $y);
    $this->z = $z;
  }

  ...
}
```

- the given class will inherit all data and behavior from *ClassName*

Static methods, fields, and constants

```
static $name = value;     # declaring a static field
const $name = value;      # declaring a static constant
```

```
# declaring a static method
public static function name(parameters) {
  statements;
}
```

```
ClassName::methodName(parameters);   # calling a static method (outside class)
self::methodName(parameters);        # calling a static method (within class)
```

- static fields/methods are shared throughout a class rather than replicated in every object

Abstract classes and interfaces

```
interface InterfaceName {
  public function name(parameters);
  public function name(parameters);
  ...
}

class ClassName implements InterfaceName { ...
```

```
abstract class ClassName {
  abstract public function name(parameters);
  ...
}
```

- **interfaces** are supertypes that specify method headers without implementations
  - cannot be instantiated; cannot contain function bodies or fields
  - enables polymorphism between subtypes without sharing implementation code
- **abstract classes** are like interfaces, but you can specify fields, constructors, methods
  - also cannot be instantiated; enables polymorphism with sharing of implementation code