Web Programming Step by Step

Lecture 12
Object-Oriented PHP
References: PHP.net, Developer.com, KillerPHP, DevX

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

```php
# 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);
```

```php
$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

```php
# 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

```php
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
<?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
<?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

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

```php
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
```
PHP

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

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

- 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 { ...
```
PHP

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

- **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