Object Oriented Programming

About OOP

OOP

- Object Oriented Programming
- Spot it where there are `->` notations
- Allows us to keep data and functionality together
- Used to organise complex programs

OOP

- Object Oriented Programming
- Spot it where there are `->` notations
- Allows us to keep data and functionality together
- Used to organise complex programs
- Basic building block are simple (I promise!)

Classes vs Objects

Class

The class is a recipe, a blueprint

```
class Table {
}
```

Object

The object is a tangible thing

```
$coffee_table = new Table();
```

The **`$coffee_table`** is an Object, an "instance" of Table

Object Properties

Can be defined in the class

```
class Table {
  public $legs;
}
```

Or set as we go along

```
$coffee_table->legs = 4;
```

Object Methods

These are "functions", with a fancy name

```php
class Table {
    public $legs;

    public function getLegCount() {
        return $this->legs;
    }
}
```

```php
$coffee_table = new Table();
$coffee_table->legs = 4;
echo $coffee_table->getLegCount(); // 4
```

Static Methods

We can call methods without instantiating a class

- `$this` is not available in a static method
- use the `::` notation (paamayim nekudotayim)
- used where we don’t need object properties

```
class Table {
    public static function getColours() {
        return array("beech", "birch", "mahogany");
    }
}
```

```
$choices = Table::getColours();
```

Accessing Classes

Just like library files, we use **`include`** and **`require`** to bring class code into our applications.

We can also use **autoloading** if our classes are predictably named.

```
function __autoload($classname) {

        if(preg_match('/[a-zA-Z]+Controller$/',$classname)) {
                include('../controllers/' . $classname . '.php');
                return true;
        } elseif(preg_match('/[a-zA-Z]+Model$/',$classname)) {
                include('../models/' . $classname . '.php');
                return true;
        } elseif(preg_match('/[a-zA-Z]+View$/',$classname)) {
                include('../views/' . $classname . '.php');
                return true;
        }
}
```

No need to include/require if you have autoloading

Objects and References

Objects are always passed by reference

```
$red_table = new Table();
$red_table->legs = 4;

$other_table = $red_table;
$other_table->legs = 3;

echo $red_table->legs; // output: 3
```

Objects behave differently from variables

- objects are always references, when assigned and when passed around
- variables are copied when they are passed into a function
- you can pass by reference by using **&**

Copying Objects

If you actually want to copy an object, you need to use the **`clone`** keyword

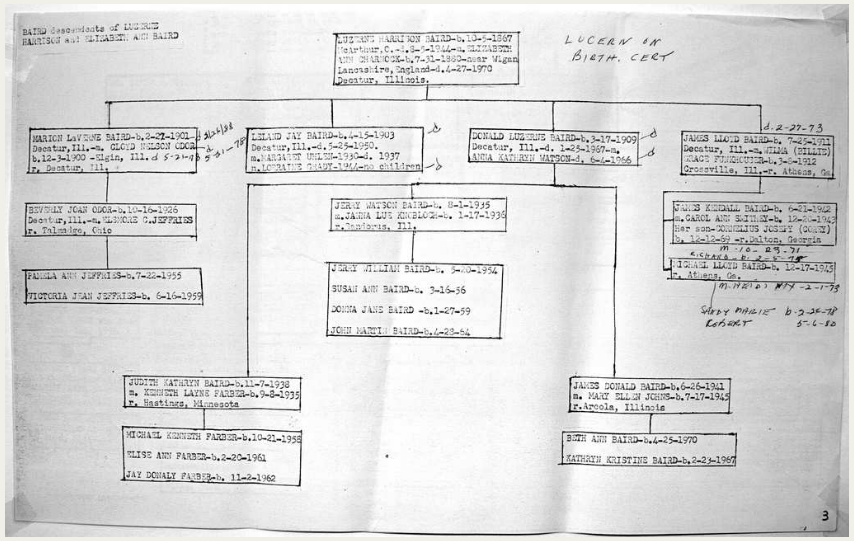
```
$red_table = new Table();
$red_table->legs = 4;

$other_table = clone $red_table;
$other_table->legs = 3;

echo $red_table->legs; // output: 4
```

Inheritance

Inheritance

OOP supports inheritance

- similar classes can share a parent and override features
- improves modularity, avoids duplication
- classes can only have one parent (unlike some other languages)
- classes can have many children
- there can be as many generations of inheritance as we need


BAIRD descendants of LUZERNE HARRISON and ELIZABETH ANN BAIRD
LUZERNE HARRISON BAIRD-b.10-5-1867 McArthur, O.-d.8-5-1944-m. ELIZABETH ANN CHARNOCK-b.7-31-1880-near Wigan Lancashire, England-d.4-27-1970 Decatur, Illinois.
LUCERN ON BIRTH. CERT
MARION LaVERNE BAIRD-b.2-27-1901-Decatur, Ill.-m. CLOYD NELSON ODOR-b.12-3-1900 -Elgin, Ill. d 5-21-78 r. Decatur, Ill.
d 3/26/88
d 5-31-78
LELAND JAY BAIRD-b.4-15-1903 Decatur, Ill.-d.5-25-1950. m. MARGARET UHLEN-1930-d. 1937 m. LORRAINE GRADY-1944-no children
DONALD LUZERNE BAIRD-b.3-17-1909 Decatur, Ill.-d. 1-25-1967-m. ANNA KATHRYN WATSON-d. 6-4-1966
d.2-27-73
JAMES LLOYD BAIRD-b. 7-25-1911 Decatur, Ill.-m. WILMA (BILLIE) GRACE FUNKHOUSER-b.3-8-1912 Crossville, Ill.-r. Athens, Ga.
BEVERLY JOAN ODOR-b.10-16-1926 Decatur, Ill.-m. ELMORE C.JEFFRIES r. Talmadge, Ohio
PAMELA ANN JEFFRIES-b.7-22-1955
VICTORIA JEAN JEFFRIES-b. 6-16-1959
JERRY WATSON BAIRD-b. 8-1-1935 m. JANNA LUE KNOBLOCH-b. 1-17-1936 r. Sandoval, Ill.
JERRY WILLIAM BAIRD-b. 5-20-1954
SUSAN ANN BAIRD-b. 3-16-56
DONNA JANE BAIRD -b.1-27-59
JOHN MARTIN BAIRD-b.4-28-64
JAMES KENDALL BAIRD-b. 6-21-1942 m. CAROL ANN SMITHEY-b. 12-20-1943 Her son-CORNELIUS JOSEPH (COREY) b. 12-12-69 -r. Dalton, Georgia
m-10-23-71
RICHARD-b. 2-5-75
MICHAEL LLOYD BAIRD-b. 12-17-1945 r. Athens, Ga.
m. HEIDI MAY-2-1-73
SANDY MARIE b.3-24-78
ROBERT 5-6-80
JUDITH KATHRYN BAIRD-b.11-7-1938 m. KENNETH LAYNE FARBER-b.9-8-1935 r. Hastings, Minnesota
MICHAEL KENNETH FARBER-b.10-21-1958
ELISE ANN FARBER-b.2-20-1961
JAY DONALY FARBER-b. 11-2-1962
JAMES DONALD BAIRD-b.6-26-1941 m. MARY ELLEN JOHNS-b.7-17-1945 r. Arcola, Illinois
BETH ANN BAIRD-b.4-25-1970
KATHRYN KRISTINE BAIRD-b.2-23-1967
3

How NOT to Design Class Hierarchies

- create one class per database table
- instantiate one object of each class
- pass all required data in as parameters (or globals!)
- never use **`$this`**

... note that the title says **"NOT"**

Designing Class Hierarchies

- identify "things" in your system (users, posts, orders)
- what qualities do they share?
- where are objects similar? Or different?
- in MVC systems, remember that controllers and views are often objects too

Inheritance Examples

```
class Table {

    public $legs;

    public function getLegCount() {
        return $this->legs;
    }
}

class DiningTable extends Table {}
```

```
$newtable = new DiningTable();
$newtable->legs = 6;
echo $newtable->getLegCount(); // 6
```

Access Modifiers

Visibility

We can control which parts of a class are available and where:

- **`public`**: always available, everywhere
- **`private`**:* only available inside this class
- **`protected`**: only available inside this class and descendants

This applies to both methods and properties

* use with caution! Protected is almost always a better choice

Protected Properties

```
class Table {
    protected $legs;

    public function getLegCount() {
        return $this->legs;
    }

    public function setLegCount($legs) {
        $this->legs = $legs;
        return true;
    }
}
```

```
$table = new Table();
$table->legs = 4;

// Fatal error: Cannot access protected property Table::$legs in /.../
```

Protected Properties

```
class Table {
    protected $legs;

    public function getLegCount() {
        return $this->legs;
    }

    public function setLegCount($legs) {
        $this->legs = $legs;
        return true;
    }
}
```

```
$table = new Table();
$table->setLegCount(4);

echo $table->getLegCount();
```

It is common to use "getters" and "setters" in this way, especially if you are a Java programmer

Protected Methods

Access modifiers for methods work exactly the same way:

```php
class Table {
    protected function getColours() {
        return array("beech", "birch", "mahogany");
    }
}
```

```php
class DiningTable extends Table {
    public function colourChoice() {
        return parent::getColours();
    }
}
```

If **`Table::getColours()`** were private, DiningTable would think that method was undefined

Object Keywords

- **`parent`**: the class this class extends

Object Keywords

- **`parent`**: the class this class extends
- **`self`**: this class, usually used in a static context, instead of **`$this`**
  - **WARNING:** in extending classes, this resolves to where it was declared
  - This was fixed in PHP 5.3 by "late static binding"

Object Keywords

- **`parent`**: the class this class extends
- **`self`**: this class, usually used in a static context, instead of **`$this`**
    - **WARNING:** in extending classes, this resolves to where it was declared
    - This was fixed in PHP 5.3 by "late static binding"
- **`static`**: the class in which the code is being used
    - Just like **`self`** but actually works :)

Identifying Objects

The instanceOf Operator

To check whether an object is of a particular class, use **`instanceOf`**

```
$table = new DiningTable();

if($table instanceOf DiningTable) {
    echo "a dining table\n";
}

if($table instanceOf Table) {
    echo "a table\n";
}
```

InstanceOf will return true if the object:

- is of that class
- is of a child of that class
- implements that interface (more on interfaces later)

Type Hinting

We have type hinting in PHP for complex types. So we can do:

```
function moveFurniture(Table $table) {
    $table->move();
    // imagine something more exciting
    return true;
}
```

PHP will error unless the argument:

- is of that class
- is of a child of that class
- implements that class (more on interfaces later)

... look familiar?

Comparing Objects

- Comparison ==
  - objects must be of the (exact) same class
  - objects must have identical properties
- Strict comparison ===
  - both arguments must refer to the same object

Magical Mystery Tour (of Magic Methods)

Magic Methods

- two underscores in method name
- allow us to override default object behaviour
- really useful for Design Patterns and solving tricky problems

Constructors

- **`__construct`**: called when a new object is instantiated
  - declare any parameters you like
  - usually we inject dependencies
  - perform any other setup

```
$blue_table = new BlueTable();
```

Destructors

- `__destruct`: called when the object is destroyed
    - good time to close resource handles

Fake Properties

When we access a property that doesn't exist, PHP calls **`__get()`** or **`__set()`** for us

```php
class Table {

    public function __get($property) {
        // called if we are reading
        echo "you asked for $property\n";
    }

    public function __set($property, $value) {
        // called if we are writing
        echo "you tried to set $property to $value\n";
    }
}

$table = new Table();

$table->legs = 5;

echo "table has: " . $table->legs . "legs\n";
```

Fake Methods

PHP calls **`__call`** when we call a method that doesn’t exist

```php
class Table {
    public function shift($x, $y) {
        // the table moves
        echo "shift table by $x and $y\n";
    }

    public function __call($method, $arguments) {
        // look out for calls to move(), these should be shift()
        if($method == "move") {
            return $this->shift($arguments[0], $arguments[1]);
        }
    }
}

$table = new Table();
$table->shift(3,5); // shift table by 3 and 5
$table->move(4,9); // shift table by 4 and 9
```

There is an equivalent function for static calls, **`__callStatic()`**

Serialising Objects

We can control what happens when we **`serialize`** and **`unserialize`** objects

```php
class Table {
}

$table = new Table();
$table->legs = 4;
$table->colour = "red";

echo serialize($table);
// O:5:"Table":2:{s:4:"legs";i:4;s:6:"colour";s:3:"red";}
```

Serialising Objects

- **`__sleep()`** to specify which properties to store
- **`__wakeup()`** to put in place any additional items on unserialize

```
class Table {
    public function __sleep() {
        return array("legs");
    }
}

$table = new Table();
$table->legs = 7;
$table->colour = "red";

$data =  serialize($table);
echo $data;
// O:5:"Table":1:{s:4:"legs";i:7;}
```

Serialising Objects

- `__sleep()` to specify which properties to store
- `__wakeup()` to put in place any additional items on unserialize

```
class Table {
    public function __wakeup() {
        $this->colour = "wood";
    }
}

echo $data;
$other_table = unserialize($data);
print_r($other_table);

/* Table Object
(
    [legs] => 7
    [colour] => wood
) */
```

Magic Tricks: clone

Control the behaviour of cloning an object by defining **`__clone()`**

- make it return false to prevent cloning (for a Singleton)
- recreate resources that shouldn’t be shared

Magic Tricks: toString

Control what happens when an object cast to a string. E.g. for an exception

```
class TableException extends Exception {
    public function __toString() {
        return '** ' . $this->getMessage() . ' **';
    }
}

try {
    throw new TableException("it wobbles!");
} catch (TableException $e) {
    echo $e;
}

// output: ** it wobbles! **
```

The default output would be

```
exception ’TableException’ with message ’it wobbles!’  in
/.../tostring.php:7 Stack trace:
```

Exceptions

Exceptions are Objects

Exceptions are **fabulous**

Exceptions are Objects

Exceptions are **fabulous**
They are pretty good use of OOP too!

- Exceptions are objects
- Exceptions have properties
- We can extend exceptions to make our own

Raising Exceptions

In PHP, we can throw any exception, any time.

```php
function addTwoNumbers($a, $b) {
    if(($a == 0) || ($b == 0)) {
        throw new Exception("Zero is Boring!");
    }

    return $a + $b;
}

echo addTwoNumbers(3,2); // 5
echo addTwoNumbers(5,0); // error!!
```

```
Fatal error: Uncaught exception ’Exception’ with message ’Zero is Boring!’ in /
Stack trace:
#0 /.../exception.php(12): addTwoNumbers(5, 0)
#1 {main}
    thrown in /.../exception.php on line 5
```

Catching Exceptions

Exceptions are thrown, and should be caught by our code - avoid Fatal Errors!

```php
function addTwoNumbers($a, $b) {
    if(($a == 0) || ($b == 0)) {
        throw new Exception("Zero is Boring!");
    }

    return $a + $b;
}

try {
    echo addTwoNumbers(3,2);
    echo addTwoNumbers(5,0);
} catch (Exception $e) {
    echo "FAIL!! (" . $e->getMessage() . ")\n";
}
// there is no "finally"

// output: 5FAIL!! (Zero is Boring!)
```

Catching Exceptions

Exceptions are thrown, and should be caught by our code - avoid Fatal Errors!

```
function addTwoNumbers($a, $b) {
    if(($a == 0) || ($b == 0)) {
        throw new Exception("Zero is Boring!");
    }

    return $a + $b;
}

try {
    echo addTwoNumbers(3,2);
    echo addTwoNumbers(5,0);
} catch (Exception $e) {
    echo "FAIL!! (" . $e->getMessage() . ")\n";
}
// there is no "finally"

// output: 5FAIL!! (Zero is Boring!)
```

Did you spot the **typehinting**??

Extending Exceptions

Make your own exceptions, and be specific when you catch

```
class DontBeDaftException extends Exception {
}

function tableColour($colour) {
    if($colour == "orange" || $colour == "spotty") {
        throw new DontBeDaftException($colour . 'is not acceptable');
    }
    echo "The table is $colour\n";
}

try {
    tableColour("blue");
    tableColour("orange");
} catch (DontBeDaftException $e) {
    echo "Don't be daft! " . $e->getMessage();
} catch (Exception $e) {
    echo "The sky is falling in! " . $e->getMessage();
}
```

Hands up if you’re still alive

Abstraction

Abstract Classes

Abstract classes are

- incomplete
- at least partially incomplete

Abstract Classes

Abstract classes are

- incomplete
- at least partially incomplete
- we cannot instantiate them
- if a class has an abstract method, the class must be marked abstract too
- common in parent classes

Abstract Examples

An abstract class:

```
abstract class Shape {
    abstract function draw($x, $y);
}
```

We can build on it, but must implement any abstract methods

```
class Circle extends Shape {
    public function draw($x, $y) {
        // imagine funky geometry stuff
        echo "circle drawn at $x, $y\n";
    }
}
```

Any non-abstract methods are inherited as normal

Interfaces

Interfaces

- prototypes of class methods
- classes "implement" an interface
- they must implement all these methods
- the object equivalent of a contract

PHP does not have multiple inheritance

Example Interface: Countable

This interface is defined in SPL, and it looks like this:

```
Interface Countable {
    public function count();
}
```

RTFM: `http://uk2.php.net/manual/en/class.countable.php`

Implementing Countable Interface

We can implement this interface in a class, so long as our class has a **`count()`** method

```
class Table implements Countable {
    public function personArrived() {
        $this->people++;
        return true;
    }

    public function personLeft() {
        $this->people--;
        return true;
    }

    public function count() {
        return $this->people;
    }
}

$my_table = new Table();
$my_table->personArrived();
$my_table->personArrived();
echo count($my_table); // 2
```

Object Design by Composition

This is where interfaces come into their own

- class hierarchy is more than inheritance
- identify little blocks of functionality
- each of these becomes an interface
- objects implement as appropriate

Polymorphism

A big word representing a small concept!

- we saw typehints and **`instanceOf`**
- classes identify as themselves, their parents or anything they implement
- word roots:
  - poly: many
  - morph: body

Our object can appear to be more than one thing

Questions?

Resources

- PHP Manual `http://php.net`
  - start here: `http://php.net/manual/en/language.oop5.php`
- Think Vitamin (Disclaimer: my posts!)
  - `http://thinkvitamin.com/code/oop-with-php-finishing-tou`
- PHP Objects, Patterns and Practice *Matt Zandstra* (book)

Further Reading

Knwoing OOP means you can go on to learn/use:

- PDO
- SPL
- Design Patterns

Thanks

Image Credits

- http://www.flickr.com/photos/amiefedora/3897320577
- http://www.flickr.com/photos/silkroadcollection/4886280049
- http://www.flickr.com/photos/dalbera/5336748810
- http://www.flickr.com/photos/stml/3625386561/
- http://www.flickr.com/photos/mikebaird/1958287914
- http://www.flickr.com/photos/dalbera/5336748810
- http://www.flickr.com/photos/mr_git/1118330301/
- http://www.flickr.com/photos/mafleen/1782135825