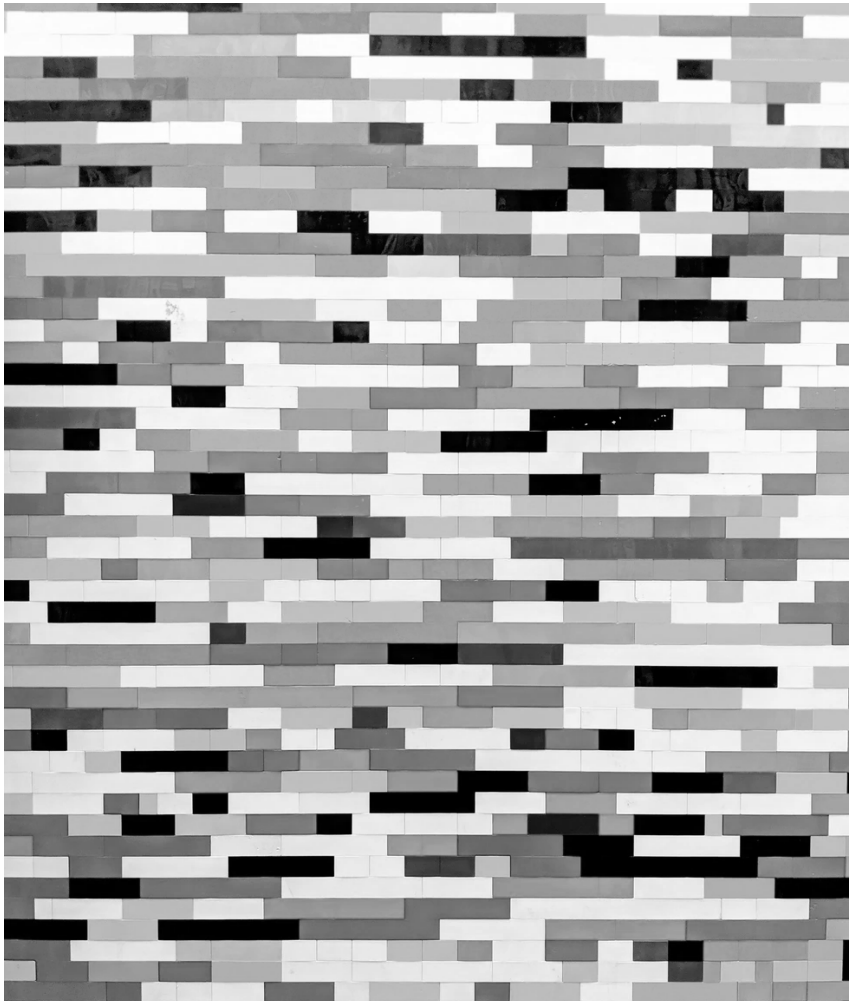
DEEPer

Back-end Development

Week 3 Session 2

TODAY'S SESSION



01 **PHP Continued**

Committing & Pushing

- It’s important to push work!
- Pushing means we can see your code on BitBucket, and we can track your progress and help when you’re stuck
- Think of it like a backup of your work – we can see a history of changes and easily revert if something breaks

```
git add . && git commit -m "Task complete" && git push
```

PHP++

01

OO Continued

- In the last session we learned the basics of using classes in PHP
- We learned how to define classes with only public properties, and how to set and get data from instances
- Today we are going to explore more functionality classes provide

```php
<?php

class TestClass
{
  public $title;
  public $description;
  public $datePublished;
}

$instance = new TestClass();
$instance->title = 'My Title';
// ...
$title = $instance->title;
```

Class Methods

- As well as properties to hold data, classes can also contain functions containing runnable code, known as **methods**
- These methods can access any properties in the class through the use of the `$this` keyword
- Similarly to properties, methods also have a visibility modifier

Class Methods - Example


```php
<?php

class Product
{
  public $title = 'Macbook Pro';

  public function getFormattedTitle()
  {
    return strtoupper($this->title);
  }
}

$product = new Product();
echo $product->getFormattedTitle(); // 'MACBOOK PRO'
```

Class Methods – Magic Methods

- PHP also provides you with a number of **magic methods** you may optionally implement in classes
- Each **magic method** provides a hook into a behaviour of a class
- For example custom logic can be run in the following cases when a class (note the double-underscore prefix);
  - Is created – `__construct($arg1, $arg2...)`
  - Is serialised to return the data desired to be serialised – `__serialize()`
  - Is used as a string – `__toString()`

Basic Inheritance

- When multiple classes share common functionality, we can avoid duplicating that functionality with inheritance
- A class can optionally **extend** one other to **inherit** its exposed behaviour
- A single class can be extended by multiple **children**, with the base class referred to as the **parent**

Basic Inheritance - Example


```php
<?php

/**
 * Parent class
 */
class Duck
{
  // Properties shared by all children
  public $name;
  public $headColour;
}


/**
 * Child classes
 */
class Mallard extends Duck
{
  public $headColour = 'green/brown';
}

class Redhead extends Duck
{
  public $headColour = 'red';
}


$myDuck = new Mallard();
$myDuck->name = 'Webster';

echo $myDuck->name; // 'Webster'
echo $myDuck->headColour; // 'green/brown'
```

- The Duck class contains properties shared by all child classes
- Child classes can override properties of their parent
- When interfacing with an object instance with a parent, the syntax is the same, e.g. setting/getting property values

Basic Inheritance – Abstract Classes

- In our previous example we don't ever want a developer to create an instance of the base class – Duck
- We only want to allow for instances of the **child** classes
- We can achieve this by setting the parent class as **abstract**

Basic Inheritance – Abstract Classes


```php
<?php

abstract class Duck
{
  public $name;
  public $headColour;
}

class Mallard extends Duck {
  public $headColour = 'green/brown';
}

// Successful!
$myDuck = new Mallard();

// Fatal error: Uncaught Error: Cannot instantiate abstract class Duck
$badDuck = new Duck();
```

Basic Inheritance – Abstract Methods

- If the class itself is abstract, methods within can also be made abstract
- Abstract methods **must be implemented by any non-abstract child**
- Failing to implement abstract methods will result in a fatal (but very helpful!) error

Basic Inheritance – Interfaces

- We have seen how we can force child classes to implement methods by adding abstract methods
- If the parent class does not provide any properties or concrete methods, we can create an interface instead
- Interfaces define public methods which children must implement
- Interfaces can be thought of as **contracts**
- Interfaces are used through the **implements** keyword

Basic Inheritance – Interfaces


```php
<?php

interface Duck
{
  public function quack();
}

class Mallard implements Duck
{
  public function quack()
  {
    echo 'Mallard quack!';
  }
}

class Redhead implements Duck
{
  // Fatal error: Class Redhead contains 1 abstract method and must therefore
  // be declared abstract or implement the remaining methods (Duck::quack)
}
```

Visibility Modifiers

- So far, all properties and methods in classes we have seen have been **public**
- Public properties and methods are accessible outside of the class
- Parent methods are accessed within a child using `parent::methodName()` syntax
- There are two other visibility modifiers we can use;
  - `protected` – only accessible in the **current class, or children**
  - `private` – available only in the **current class**, not children

Typed Properties

- Until now, all properties have been defined with the following syntax – `visibility $propertyName`
- For example `public $title`
- As of PHP 7.4 a PHP type can be added to a property's definition
- This type will enforce that no data of any other type is set against a class property

Typed Properties - Example


```php
<?php

class User
{
  public string $name;
  public int $age;

  public function __construct($name, $age)
  {
    $this->name = $name;
    $this->age = $age;
  }
}

// Success!
$user = new User('Joe Bloggs', 50);

// Fatal error: Uncaught TypeError: Typed property User::$age must be int,
// string used
$user2 = new User('Jane Doe', 'invalid');
```

Typed Properties – Casting Problem

- In this latest example our code is now safer – for example, the name cannot contain an integer
- However we still have one potential issue with this implementation
- Remember, despite the existence of functionality to implement strict types in PHP, the language itself is still **loosely typed**
- If you provide PHP with the wrong type, **PHP will try to change the type for you**
- There is a PHP setting available which allows us to define whether type checks should be strict: **strict_types**

Typed Properties – Casting Problem

- Strict types can be declared in one of two ways:
  - Via the PHP configuration file (discussed in later weeks)
  - At the top of the file, with `<?php declare(strict_types=1);`

Typed Properties – Casting Problem


```php
<?php

class User
{
  public string $name;
  public int $age;

  public function __construct($name, $age)
  {
    $this->name = $name;
    $this->age = $age;
  }
}

// We don't want to allow false as an age - that isn't an integer...
$user = new User('Joe Bloggs', false);

// ...but it doesn't error

// There are two possible outcomes here depending on the strict_types setting
// if strict types is off:
//   PHP will try to cast "false" to an integer value, the outcome being 0
// if strict types is on:
//   PHP will throw a fatal error:
//   Fatal error: Uncaught TypeError: Argument 1 passed to test() must be of the type int, bool given
echo $user->age; // 0
```

Type Hinting Parameters

- Like against properties, types can also be added to method parameters
- All common PHP types are allowed, e.g. string, int, array
- We can also provide type declarations for properties using **classes** or **interfaces**
- Any typed parameter can be marked to allow **null** to be provided
- Any parameter can be assigned a default value, making it optional

Type Hinting Parameters


```php
<?php

class User
{
  public string $name;
  public DateTime $dateOfBirth;

  public function __construct(string $name, DateTime $dateOfBirth)
  {
    $this->name = $name;
    $this->dateOfBirth = $dateOfBirth;
  }
}

// Success!
$dateOfBirth = new DateTime('1990-05-20');
$user = new User('Joe Bloggs', $dateOfBirth);

// Again, PHP will cast 9 to a string of '9'
// But it cannot cast a string to an object instance, so we get a Fatal;

// Fatal error: Uncaught TypeError: Argument 2 passed to User::__construct()
// must be an instance of DateTime, string given
$user2 = new User(9, 'hello');
```

Nullable Types & Default Values


```php
<?php

class TestClass
{
  public function nullableParameter(?string $nullable)
  {
    echo $nullable; // Can be any string, or null
  }

  public function defaultedParameter(string $defaulted = 'Hello World')
  {
    echo $defaulted;
  }
}

$object = new TestClass();
$object->nullableParameter(null); // null
$object->nullableParameter('A string this time'); // 'A string this time'

$object->defaultedParameter(); // 'Hello World'
$object->defaultedParameter('I provided my own'); // 'I provided my own'
```

Function/Method Return Types

- Like how we can add type hints to parameters, we can also specify the data type a function or method should return
- This again makes our code safer and more predictable
- The same types supported by parameter types are supported by return types
- There is one additional type supported however – void
- Void indicates that the method should return **no value at all**
- Equally, return types can be nullable

Function/Method Return Types


```php
<?php

class TestClass
{
  public function stringReturnType(): string
  {
    return 'I am a string!';
  }

  public function nullableReturnType(): ?string
  {
    return null;
  }

  public function incorrectReturnType(): int
  {
    return 'I am not an integer!';
  }
}

$object = new TestClass();

echo $object->stringReturnType(); // 'I am a string!'
echo $object->nullableReturnType(); // null

// Fatal error: Uncaught TypeError: Return value of
// TestClass::incorrectReturnType() must be of the type int, string returned
echo $object->incorrectReturnType();
```

Basic Security

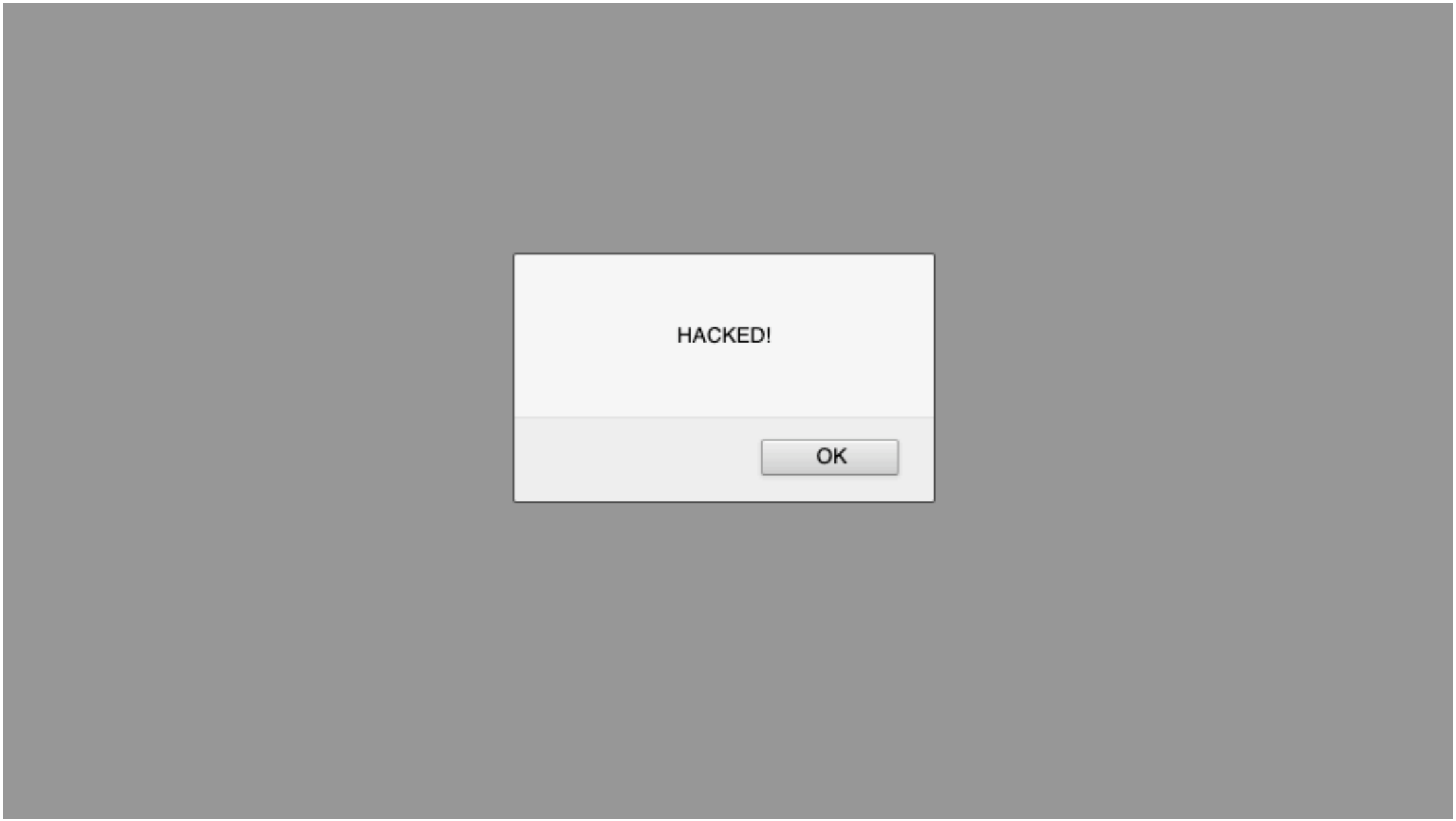
- Foreign data loaded by your application can **never** be trusted
- This includes data from users (e.g. form submissions) and from other sources like files or external services
- All data inbound to your application should be **filtered and validated**. This will be covered in a future session
- All output from your application should be **escaped**
- Failure to adhere to these steps can render your application vulnerable to attack
- Two attack vectors you may have heard of are SQL Injection and Cross Site Scripting (XSS) attacks

Basic Security – Filtering Input

- Before inputs are stored, we should filter them
- For example, a user may attempt to input some text in a number field
- Hackers may also want to submit malicious code
- Forms are a common attack vector
- `filter_var` can be used to filter inputs
- Reference:
  - https://www.php.net/manual/en/function.filter-var.php
  - https://www.php.net/manual/en/filter.filters.php

Basic Security – Escaping Output

- Cross Site Scripting (XSS) is one of the most common vulnerabilities related to not escaping output
- This generally occurs when user input is displayed on a web page
- If a user enters code into an input and it is outputted to the page unescaped, any valid code such as HTML will be parsed
- This can be exploited to inject custom JavaScript code into a page with effects ranging from nuisance to data theft or worse
- In our case escaping output means ensuring nothing unwanted is rendered in the HTML output of a page


HACKED!
OK

Exceptions

- Exceptions are objects which are like errors, but are triggered manually by code, not PHP due to parsing errors
- They are generally used when the code detects some invalid state that it cannot reliably handle or recover from
- Exceptions can be “thrown” anywhere in code and “caught” by calling code
- When an exception is caught it can be handled by the code, for example redirecting a user or displaying a useful error
- Custom Exception classes can be created for contextual errors, e.g. an `AuthenticationException`

Exceptions


```php
<?php

class TestClass
{
  public function doThing()
  {
    $userIsAuthenticated = false;

    if (!$userIsAuthenticated) {
      throw new Exception('You must be logged in to do that!');
    }
  }
}

try {
  $object = new TestClass();
  $object->doThing()
} catch (Exception $e) {
  // Some error handling here
}
```

Exceptions – Custom Exception


```php
<?php

class AuthenticationException extends Exception
{
}

class TestClass
{
  public function doThing()
  {
    $userIsAuthenticated = false;

    if (!$userIsAuthenticated) {
      throw new AuthenticationException('You must be logged in to do that!');
    }
  }
}

try {
  $object = new TestClass();
  $object->doThing();
} catch (AuthenticationException $e) {
  // An AuthenticationException specifically was caught
} catch (Exception $e) {
  // Any other Exception thrown by the code within try{} was caught
}
```