Programmation orientée objet (avancé)

Lien vers le cours

L. Delafontaine, avec l'aide de GitHub Copilot.

Retrouvez plus de détails dans le support de cours

Cette présentation est un résumé du support de cours. Pour plus de détails, consultez le support de cours.

Objectifs

- Rappeler les concepts de base de la programmation orientée objet.
- Appliquer les notions d'interface, d'héritage et d'abstraction avec la programmation orientée objet.

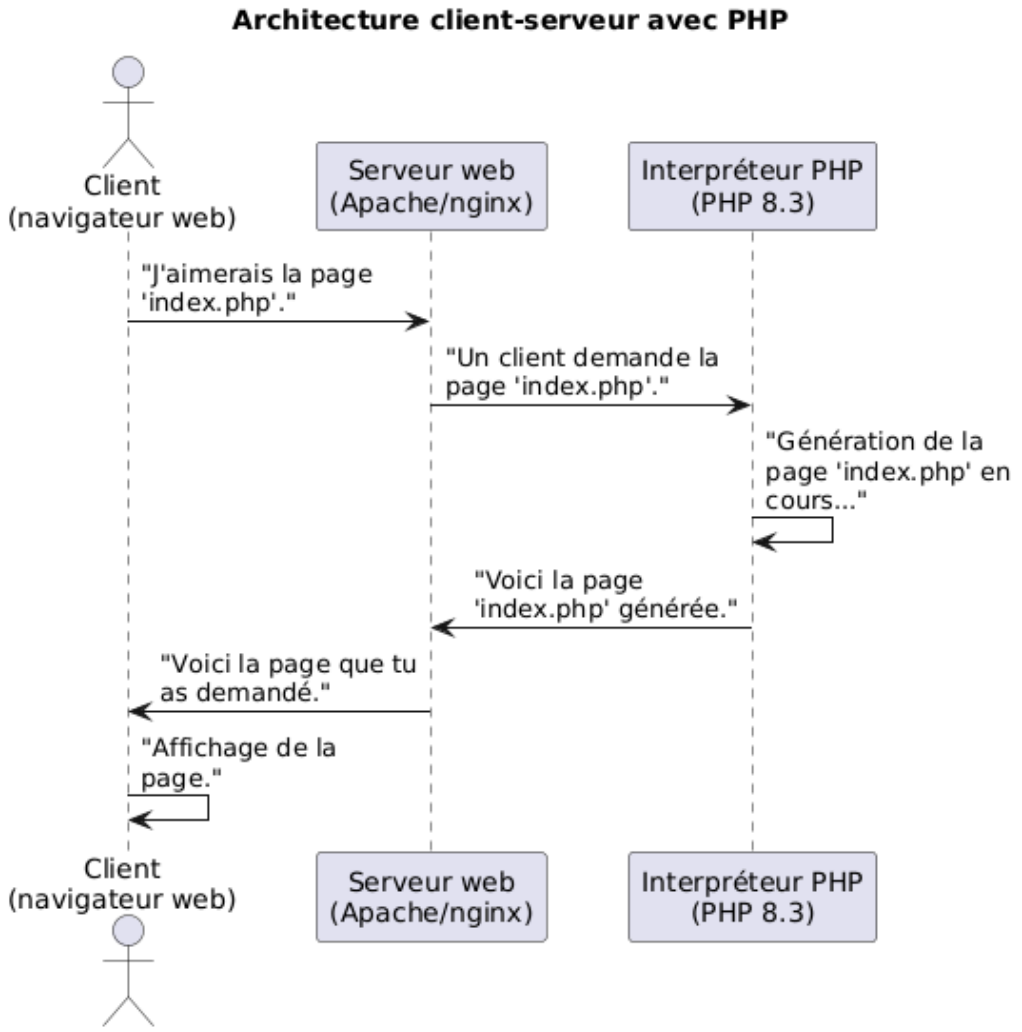
PHP, un rappel

Architecture client-serveur (1/2)

- La plupart des applications web modernes reposent sur une architecture dite *"client-serveur"* :
  1. Un client (navigateur web) envoie une requête à un serveur.
  2. Le serveur répond aux requêtes des différents clients.
  3. Le client affiche le résultat de la requête.
- PHP repose sur cette même architecture.

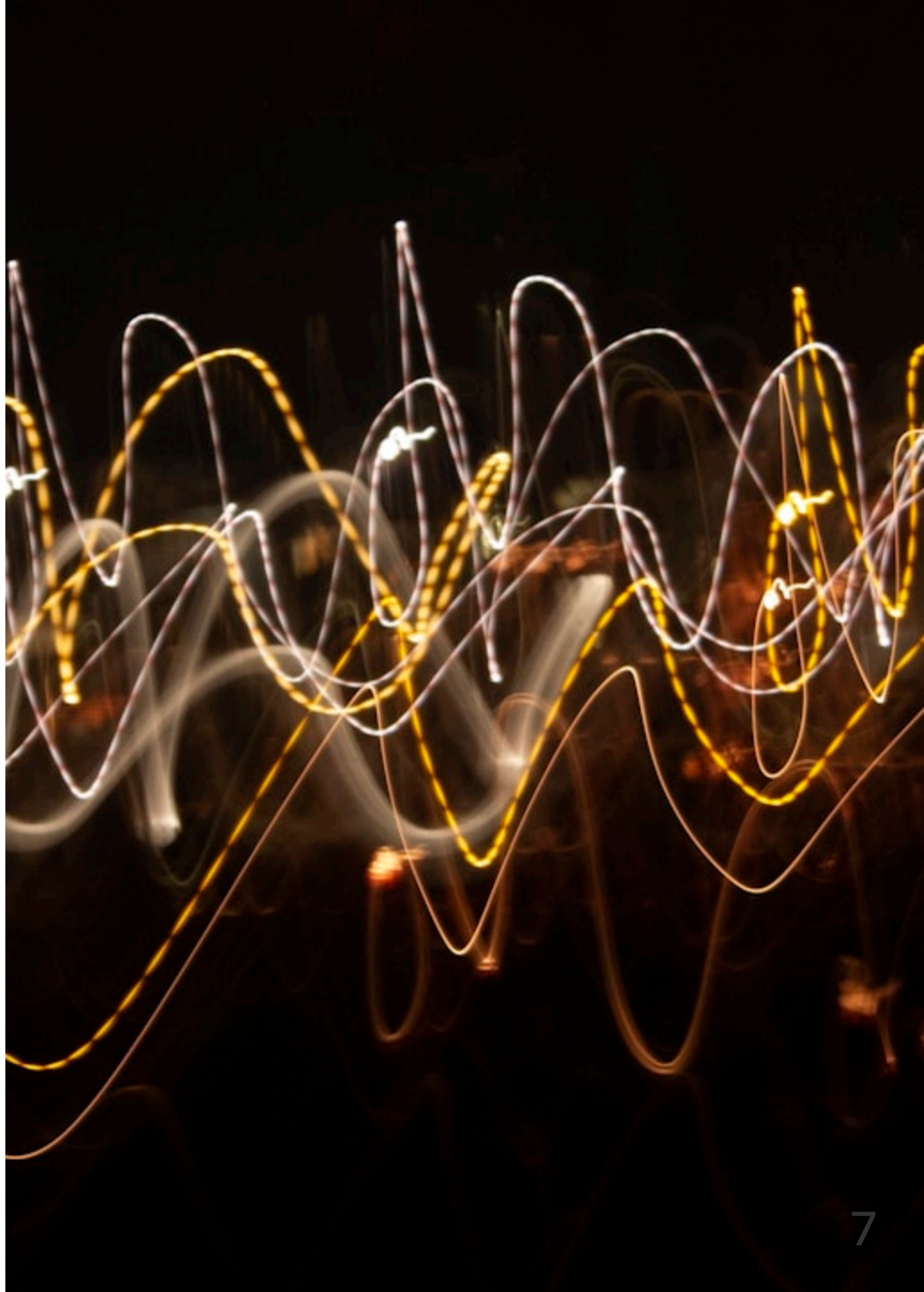
Architecture client-serveur (2/2)

- PHP fonctionne grâce aux outils suivants :
  - Un serveur web.
  - PHP installé sur le serveur web.
  - Un navigateur web.
  - Un éditeur de code (pour le développement).



Variables (1/2)

- PHP est un language de programmation à typage dynamique.
- Il n'y a pas besoin de déclarer le type de données d'une variable.
- Le type de données d'une variable est déterminé par la valeur qui lui est assignée.



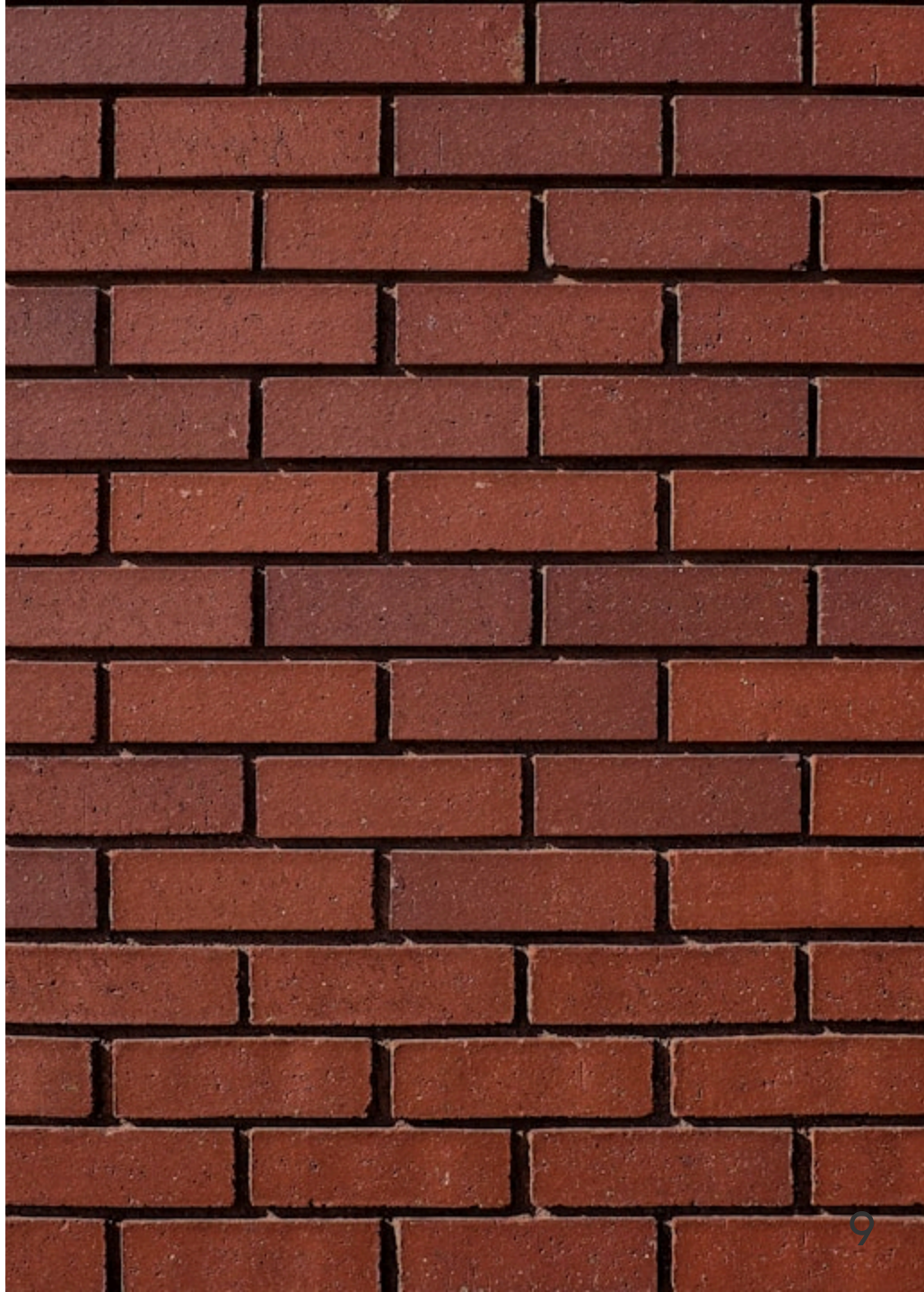
Variables (2/2)

- Une variable commence toujours par le symbole `$` en PHP.
- Une valeur lui est affectée (= donnée) avec l'opérateur `=`.
- Une variable peut changer de type en cours d'exécution.

```php
<?php
$variable = "Hello";                          // string
$variable = 42;                               // int
$variable = 3.14;                             // float
$variable = true;                             // bool
$variable = [true, 2, "3", 4 => [5, 6, 7]]; // array
$variable = null;                             // null
```

Constantes (1/2)

- Les constantes sont des valeurs qui ne peuvent pas être modifiées.
- Les constantes sont déclarées avec le mot-clé `const` ou avec la fonction `define()`.
- La convention veut que les constantes soient écrites en majuscules.



Constantes (2/2)

```php
<?php
define("PI", 3.14159); // Définition d'une constante
const EULER = 2.71828; // Définition d'une constante

echo PI;    // Affiche 3.14159
echo EULER; // Affiche 2.71828

PI = 3.14; // Erreur : les constantes ne peuvent pas être modifiées
```

Opérateurs (1/2)

- Permet d'effectuer des opérations sur des variables et des valeurs.
- Opérateurs arithmétiques : `+`, `-`, `*`, `/`, `%` (modulo)
- Opérateurs de comparaison : `==` (égal), `!=` (différent), `>` (supérieur), `<` (inférieur)
- Opérateurs logiques : `&&` (et), `||` (ou), `!` (non/inversion)

Opérateurs (2/2)

```php
<?php

$a = 10;
$b = 5;
$c = 15;
$d = 15;

// L'opérateur `===` permet de vérifier la valeur et le type (à préférer).
if ($a > $b && $c === $d) {
  echo "Condition met!";
} else {
  echo "Condition not met!";
}
```

Structures conditionnelles (1/4)

- Permettent de contrôler le flux d'exécution d'un programme.
- Utilisent les opérateurs de comparaison et logiques.
- Elles se composent de `if`, `else`, `elseif` et `switch`.

Structures conditionnelles (2/4)

```php
<?php
$age = 20;

if ($age < 18) {
    echo "You are a minor.";
} elseif ($age >= 18 && $age < 65) {
    echo "You are an adult.";
} else {
    echo "You are a senior.";
}
```

Structures conditionnelles (3/4)

```php
<?php
$day = "Monday";

switch ($day) {
    case "Monday":
        echo "It's Monday!";
        break;
    case "Tuesday":
        echo "It's Tuesday!";
        break;
    case "Wednesday":
        echo "It's Wednesday!";
        break;
// ...
```

```php
// ...
    case "Thursday":
        echo "It's Thursday!";
        break;
    case "Friday":
        echo "It's Friday!";
        break;
    case "Saturday":
        echo "It's Saturday!";
        break;
    case "Sunday":
        echo "It's Sunday!";
        break;
}
```

Structures conditionnelles (4/4)

```php
<?php
$day = "Monday";

switch ($day) {
    case "Monday":
    case "Tuesday":
    case "Wednesday":
    case "Thursday":
    case "Friday":
        echo "It's a weekday!";
        break;
// ...
```

```php
// ...
    case "Saturday":
    case "Sunday":
        echo "It's the weekend!";
        break;
    default:
        echo "Invalid day!";
        break;
}
```

Fonctions

- Ensemble d'instructions pour effectuer une tâche spécifique
- Inspirée des fonctions mathématiques :
  - $f(x) = x^2$
  - où x est un paramètre
  - $f(2) = 4$, $f(3) = 9$, etc.
- Permettent de structurer le code en blocs réutilisables.

Fonctions sans paramètres

```php
<?php
function greet() {
    return "Hello, World!";
}

// 1. Exécute la fonction `greet()`
// 2. Récupère la valeur de retour
// 3. Affecte (= donne) cette valeur à la variable `$greetings`
$greetings = greet();

// Affiche le résultat ("Hello, World!")
echo $greetings;
```

Fonctions avec paramètres

```php
<?php
function greet($name) {
    return "Hello, " . $name . "!";
}

$greetings = greet("Alice");
echo $greetings . "<br>";       // "Hello, Alice!"
echo greet("Bob") . "<br>";     // "Hello, Bob!"
```

Fonctions avec des paramètres par défaut

```php
<?php
function greet($name = "World") {
    return "Hello, " . $name . "!";
}

echo greet() . "<br>";          // "Hello, World!" (utilise la valeur par défaut)
echo greet("Alice") . "<br>";   // "Hello, Alice!" (utilise l'argument fourni)
```

Fonctions avec typage des paramètres et du retour (1/2)

- Depuis la version 7.1 de PHP, il est possible de typer les paramètres et le retour des fonctions.

```php
<?php
function greet(string $name = "World"): string {
    return "Hello, " . $name . "!";
}

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

Fonctions avec typage des paramètres et du retour (2/2)

- Le typage permet de s'assurer que les arguments passés à une fonction sont du bon type.

```php
echo greet() . "<br>";           // "Hello, World!"
echo greet("Alice") . "<br>";    // "Hello, Alice!"
echo greet(42) . "<br>";         // "Hello, 42!" (conversion implicite)
echo add(2, 3) . "<br>";         // 5

// Erreur 1 : Implicit conversion from float 2.5 to int loses precision
// Erreur 2 : Argument #2 ($b) must be of type int, string given
echo add(2.5, "Hello") . "<br>";
```

Importation de fichiers (1/2)

- L'importation permet de réutiliser du code défini dans d'autres fichiers.
- Il est recommandé d'utiliser `require` (erreur et arrête l'exécution) plutôt que `include` (avertissement mais continue l'exécution).

Importation de fichiers (2/2)

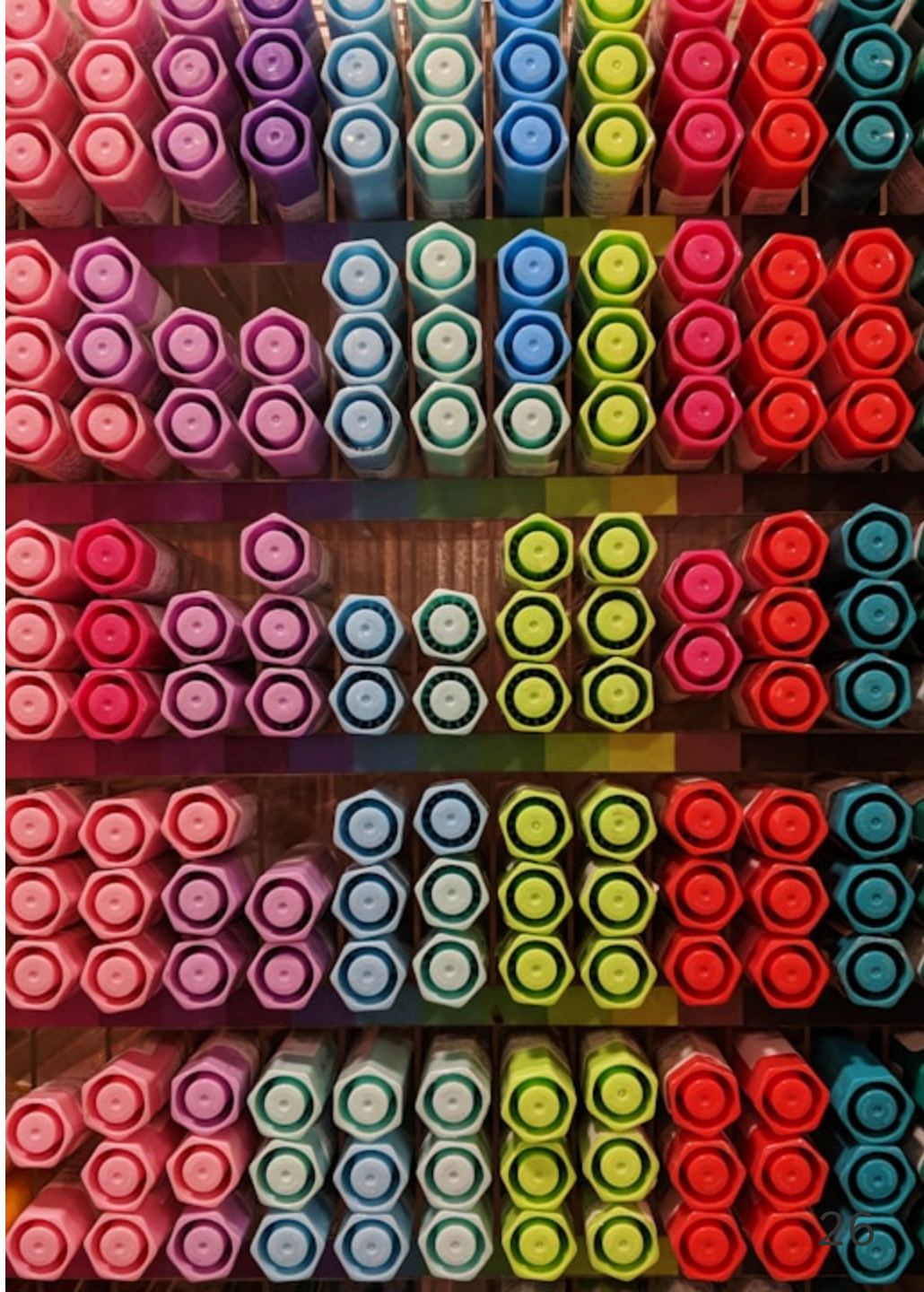
```php
<?php
// Fichier functions.php
function greet(string $name = "World"): string {
    return "Hello, " . $name . "!";
}
```

```php
<?php
// Fichier index.php
require "functions.php";

echo greet("Alice"); // "Hello, Alice!"
```

Tableaux et boucles

Tableaux (1/2)

- Les tableaux sont des collections de valeurs.
- Les tableaux sont déclarés entre des crochets (`[]`) ou avec la fonction `array()`.
- Les valeurs peuvent être de n'importe quel type.
- Il existe trois types de tableaux en PHP : indexés, associatifs et multidimensionnels.



Tableaux (2/2)

```php
<?php
// Tableau indexé numériquement
$fruits = ['apple', 'banana', 'orange'];

echo $fruits[0] . "<br>"; // "apple"

// Tableau associatif
$person = [
    'name' => 'Alice',
    'age' => 30,
    'city' => 'New York'
];

echo $person['name'] . "<br>"; // "Alice"
```

Boucles (1/5)

- Les boucles sont des structures de contrôle qui permettent d'exécuter un bloc de code plusieurs fois.
- Elles sont utilisées pour parcourir des tableaux ou des collections de données.
- Il existe plusieurs types de boucles en PHP : `for`, `while`, `do...while` et `foreach`.

Boucles (2/5)

```php
<?php
// Affiche les nombres de 0 à 9
for ($i = 0; $i < 10; $i++) {
    echo "$i<br>";
}
```

Boucles (3/5)

```php
<?php
$i = 0;

// Affiche les nombres de 0 à 9
while ($i < 10) {
    echo "$i<br>";
    $i++;
}
```

Boucles (4/5)

```php
<?php
$randomNumber = null;

do {
    // La fonction `rand()` génère un nombre aléatoire entre 1 et 10
    $randomNumber = rand(1, 10);
    echo "The random number is $randomNumber<br>";
} while ($randomNumber < 8);
```

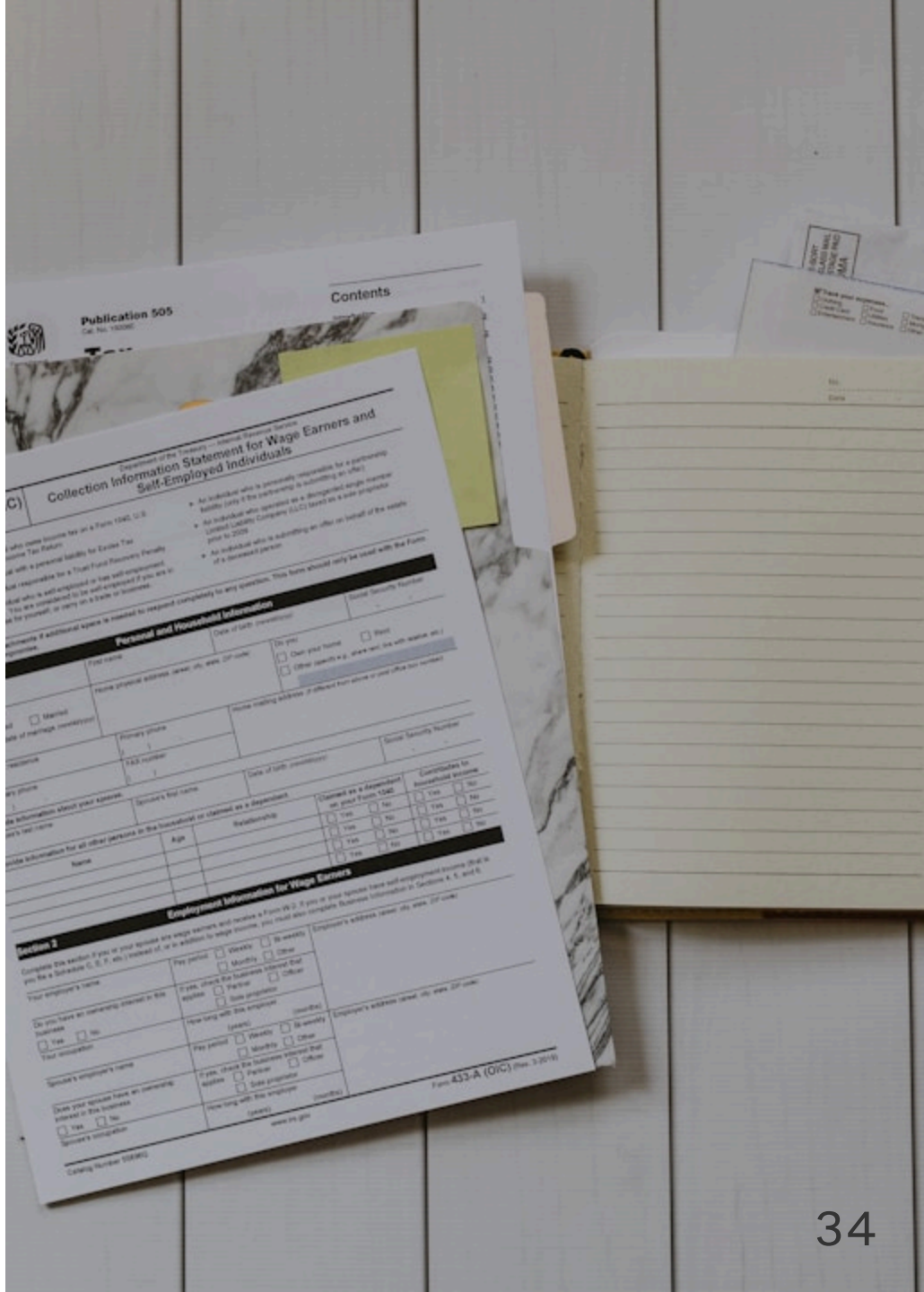
Boucles (5/5)

```php
<?php
$users = [
    'john' => [
        'name' => 'John Doe',
        'age' => 30,
        'city' => 'New York',
    ],
    'jane' => [
        'name' => 'Jane Doe',
        'age' => 25,
        'city' => 'Los Angeles',
    ],
];
```

```php
// `$user` contient la valeur de l'élément du tableau
foreach ($users as $user) {
    echo "Name: {$user['name']}<br>";
    echo "Age: {$user['age']}<br>";
    echo "City: {$user['city']}<br>";
    echo "<br>";
}
```

Formulaires HTML, validation et sécurité

- Les formulaires HTML permettent de collecter des données.
- Les données des formulaires sont stockées dans les superglobales `$_POST` et `$_GET`.
- Il est nécessaire de traiter et valider les données pour éviter des vulnérabilités.



Programmation orientée objet (base) (1/2)

- Paradigme de programmation qui utilise des "objets" pour modéliser des concepts du monde réel.
- Une classe est un modèle pour créer des objets. Un objet est une instance (= un exemplaire) d'une classe.

Programmation orientée objet (base) (2/2)

```php
<?php
class User {
    // Propriétés (attributs)
    private string $firstName;
    private string $lastName;

    // Constructeur
    public function __construct(string $firstName, string $lastName) {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
    }
```

```php
    // Méthodes
    public function getFirstName(): string {
        return $this->firstName;
    }

    public function setFirstName(string $firstName): void {
        $this->firstName = $firstName;
    }

    public function getLastName(): string {
        return $this->lastName;
    }

    public function setLastName(string $lastName): void {
        $this->lastName = $lastName;
    }

    public function getFullName(): string {
        return "{$this->firstName} {$this->lastName}";
    }
}
```

```php
// Création d'objets (instanciation)
$user1 = new User("Alice", "Smith");
$user2 = new User("Bob", "Johnson");

// Utilisation des méthodes
echo $user1->getFirstName() . "<br>";    // "Alice"
echo $user2->getFullName() . "<br>";     // "Bob Johnson"
$user2->setLastName("Doe");              // Modifie le nom de famille de Bob
echo $user2->getFullName() . "<br>";     // "Bob Doe"
```

Programmation orientée objet (avancé)

Interfaces (1/2)

- Les interfaces définissent un contrat que les classes doivent respecter.
- Chaque classe qui implémente une interface doit définir toutes les méthodes déclarées dans l'interface.
- Cela permet de garantir que certaines méthodes sont toujours présentes dans les classes qui implémentent l'interface.

Interfaces (2/2)

```php
<?php
interface AnimalInterface {
    public function makeSound(): string;
    public function getHabitat(): string;
}
```

```php
class Lion implements AnimalInterface {
    public function makeSound(): string {
        return "Roar!";
    }

    public function getHabitat(): string {
        return "Savannah";
    }
}

class Penguin implements AnimalInterface {
    public function makeSound(): string {
        return "Honk!";
    }

    public function getHabitat(): string {
        return "Antarctica";
    }
}
```

```php
$lion = new Lion();
$penguin = new Penguin();

echo $lion->makeSound();         // "Roar!"
echo $lion->getHabitat();        // "Savannah"
echo $penguin->makeSound();      // "Honk!"
echo $penguin->getHabitat();     // "Antarctica"
```

Héritage (1/2)

- L'héritage permet à une classe fille d'hériter des propriétés et méthodes d'une classe parent.
- Cela favorise la réutilisation du code et la création de hiérarchies de classes.

Héritage (2/2)

```php
<?php
class Plant {
    protected string $englishName;
    protected string $latinName;

    public function __construct(string $englishName, string $latinName) {
        $this->englishName = $englishName;
        $this->latinName = $latinName;
    }

    public function getEnglishName(): string {
        return $this->englishName;
    }

    public function getLatinName(): int {
        return $this->latinName;
    }
}
```

```php
class Basil extends Plant {
    private string $variety;

    public function __construct(string $englishName, string $latinName, string $variety) {
        parent::__construct($englishName, $latinName);
        $this->variety = $variety;
    }

    public function getVariety(): string {
        return $this->variety;
    }
}
```

```php
class Tomato extends Plant {
    private string $color;

    public function __construct(string $englishName, string $latinName, string $color) {
        parent::__construct($englishName, $latinName);
        $this->color = $color;
    }

    public function getColor(): string {
        return $this->color;
    }
}
```

```php
$plant = new Plant("Generic Plant", "Plantae");
$basil = new Basil("Basil", "Ocimum basilicum", "Sweet Basil");
$tomato = new Tomato("Tomato", "Solanum lycopersicum", "Red");

echo $plant->getEnglishName(); // "Generic Plant"
echo $plant->getLatinName();   // "Plantae"
echo $basil->getVariety();     // "Sweet Basil"
echo $tomato->getColor();      // "Red"
```

Abstraction (1/2)

- Les classes abstraites définissent une base commune avec des méthodes partiellement implémentées.
- Une classe abstraite ne peut pas être instanciée directement.
- Les classes filles doivent implémenter les méthodes abstraites définies dans la classe abstraite.

Abstraction (2/2)

```php
<?php
abstract class Shape {
    protected string $color;

    public function __construct(string $color) {
        $this->color = $color;
    }

    // Méthode concrète (implémentée)
    public function getColor(): string {
        return $this->color;
    }

    // Méthodes abstraites (doivent être implémentées par les classes filles)
    abstract public function calculateArea(): float;
    abstract public function calculatePerimeter(): float;
```

```
    // Méthode concrète utilisant les méthodes abstraites
    public function getShapeInfo(): string {
        return sprintf(
            "Shape: %s, Color: %s, Area: %.2f, Perimeter: %.2f",
            static::class,
            $this->color,
            $this->calculateArea(),
            $this->calculatePerimeter()
        );
    }
}
```

```php
class Rectangle extends Shape {
    private float $width;
    private float $height;

    public function __construct(string $color, float $width, float $height) {
        parent::__construct($color);
        $this->width = $width;
        $this->height = $height;
    }

    public function calculateArea(): float {
        return $this->width * $this->height;
    }

    public function calculatePerimeter(): float {
        return 2 * ($this->width + $this->height);
    }
}
```

```php
class Circle extends Shape {
    private float $radius;

    public function __construct(string $color, float $radius) {
        parent::__construct($color);
        $this->radius = $radius;
    }

    public function calculateArea(): float {
        return pi() * pow($this->radius, 2);
    }

    public function calculatePerimeter(): float {
        return 2 * pi() * $this->radius;
    }
}
```

```php
$rectangle = new Rectangle("blue", 10, 5);
$circle = new Circle("red", 7);

echo $rectangle->getShapeInfo();
// Shape: Rectangle, Color: blue, Area: 50.00, Perimeter: 30.00

echo $circle->getShapeInfo();
// Shape: Circle, Color: red, Area: 153.94, Perimeter: 43.98
```

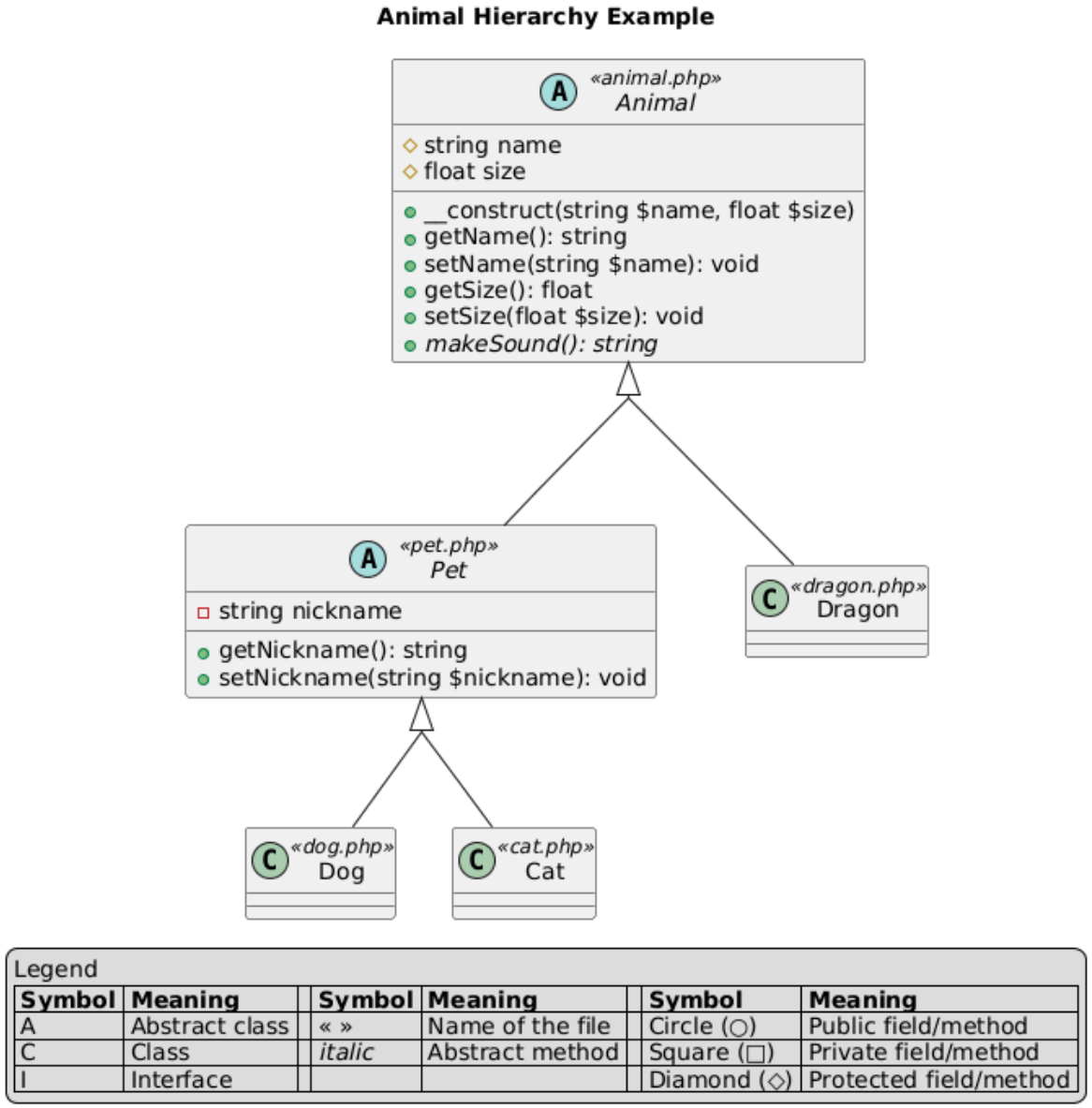
```php
$shape = new Shape("green"); // Erreur fatale : Cannot instantiate abstract class Shape
```

Inclusion des fichiers et classes

- Pour organiser le code, chaque classe peut être définie dans son propre fichier.
- Il est important de gérer correctement l'inclusion des fichiers pour éviter les erreurs.

Inclusion manuelle

- L'inclusion manuelle consiste à inclure chaque fichier de classe individuellement avec `require` ou `include`.
- Cela peut devenir fastidieux dans les projets avec de nombreuses classes.
- Il est préférable d'utiliser `require_once` ou `include_once` pour éviter les inclusions multiples.
- Illustrons les problèmes potentiels avec un exemple simple.


Animal Hierarchy Example
«animal.php»
Animal
◇ string name
◇ float size
• __construct(string $name, float $size)
• getName(): string
• setName(string $name): void
• getSize(): float
• setSize(float $size): void
• makeSound(): string
«pet.php»
Pet
▫ string nickname
• getNickname(): string
• setNickname(string $nickname): void
«dragon.php»
Dragon
«dog.php»
Dog
«cat.php»
Cat
Legend
Symbol | Meaning | Symbol | Meaning | Symbol | Meaning
A | Abstract class | « » | Name of the file | Circle (○) | Public field/method
C | Class | italic | Abstract method | Square (□) | Private field/method
I | Interface | | | Diamond (◇) | Protected field/method

```php
<?php
// Animal.php
abstract class Animal {
    protected string $name;
    protected float $size;

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

    abstract public function makeSound(): string;

    public function getName(): string {
        return $this->name;
    }

    public function setName(string $name): void {
        $this->name = $name;
    }

    public function getSize(): float {
        return $this->size;
    }

    public function setSize(float $size): void {
        $this->size = $size;
    }
}
```

```php
<?php
// Pet.php
require 'Animal.php';

abstract class Pet extends Animal {
    protected string $nickname;

    public function __construct(string $name, float $size, string $nickname) {
        parent::__construct($name, $size);
        $this->nickname = $nickname;
    }

    public function getNickname(): string {
        return $this->nickname;
    }

    public function setNickname(string $nickname): void {
        $this->nickname = $nickname;
    }
}
```

```php
<?php
// Dog.php
require 'Pet.php';

class Dog extends Pet {
    public function __construct(string $name, float $size, string $nickname) {
        parent::__construct($name, $size, $nickname);
    }

    public function makeSound(): string {
        return "Woof!";
    }
}
```

```php
<?php
// Cat.php
require 'Pet.php';

class Cat extends Pet {
    public function __construct(string $name, float $size, string $nickname) {
        parent::__construct($name, $size, $nickname);
    }

    public function makeSound(): string {
        return "Meow!";
    }
}
```

```php
<?php
// index.php
require 'Dog.php';
require 'Cat.php';

$dog = new Dog("Nalia", 30.5, "Naliouille");
$cat = new Cat("Tofu", 10.0, "Sushi");

echo $dog->getName() . " says: " . $dog->makeSound() . "<br>";
echo $cat->getName() . " says: " . $cat->makeSound() . "<br>";
```

```php
<?php
// Dog.php
require_once 'Pet.php';
...
```

```php
<?php
// Cat.php
require_once 'Pet.php';
...
```

```php
<?php
// Pet.php
require_once 'Animal.php';
...
```

Espaces de noms (namespaces) (1/2)

- Les namespaces permettent d'organiser le code et d'éviter les conflits de noms dans des projets qui utilisent de nombreuses classes.
- Ils sont définis avec le mot-clé `namespace` au début d'un fichier.
- Ils permettent de grouper des classes, interfaces, fonctions et constantes sous un même espace de noms.

```php
<?php
// src/Animals/Animal.php
namespace Animals;

abstract class Animal {
    protected string $name;
    // ...
}
```

```php
<?php
// src/Animals/Pets/Pet.php
namespace Animals\Pets;

require_once 'Animal.php';

use Animals\Animal;

abstract class Pet extends Animal {
    protected string $nickname;
    // ...
}
```

```php
<?php
// src/Animals/Pets/Dog.php
namespace Animals\Pets;

require_once 'Pet.php';

use Animals\Pets\Pet;

class Dog extends Pet {
    // ...
}
```

```php
<?php
// src/Animals/Pets/Cat.php
namespace Animals\Pets;

require_once 'Pet.php';

use Animals\Pets\Pet;

class Cat extends Pet {
    // ...
}
```

Inclusion automatique (autoloader) (1/2)

- L'autoloader permet de charger automatiquement les classes sans inclusions manuelles.
- Permet de simplifier la gestion des dépendances.
- L'autoloader sera chargé d'importer les classes au moment où elles sont utilisées en utilisant le namespace pour localiser le fichier.

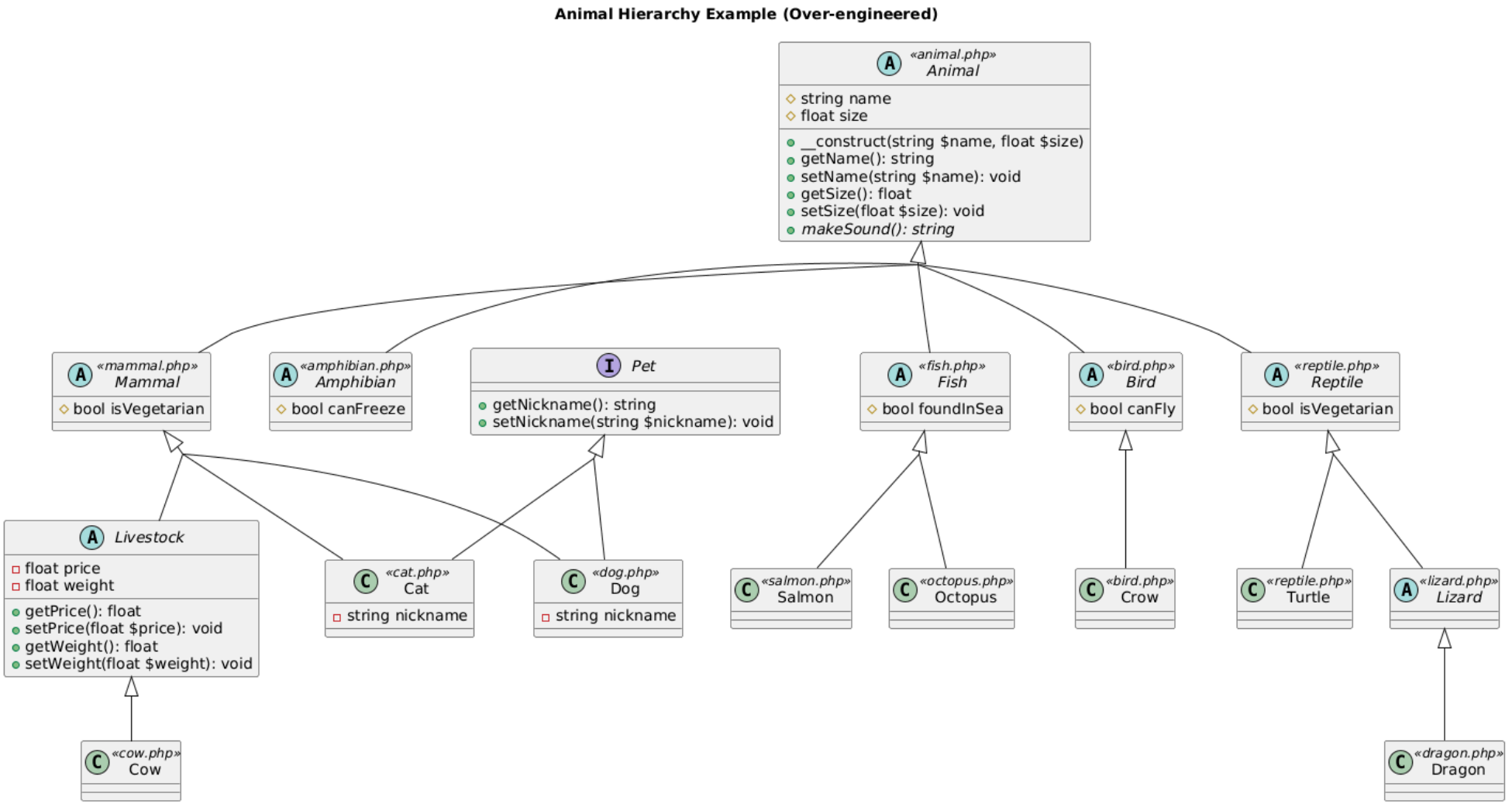
Inclusion automatique (autoloader) (2/2)

```php
<?php
// autoloader.php
// Charge les classes automatiquement
spl_autoload_register(function ($class) {
    // Convertit les séparateurs de namespace en séparateurs de répertoires
    $relativePath = str_replace('\\', '/', $class);

    // Construit le chemin complet du fichier
    $file = __DIR__ . '/../classes/' . $relativePath . '.php';

    // Vérifie si le fichier existe avant de l'inclure
    if (file_exists($file)) {
        // Inclut le fichier de classe
        require_once $file;
    }
});
```

```php
<?php
// index.php
require 'autoloader.php'; // Plus besoin d'inclure chaque fichier de classe manuellement

use Animals\Pets\Dog;
use Animals\Pets\Cat;

$dog = new Dog("Nalia", 30.5, "Naliouille");
$cat = new Cat("Tofu", 10.0, "Sushi");

echo $dog->getName() . " says: " . $dog->makeSound() . "<br>";
echo $cat->getName() . " says: " . $cat->makeSound() . "<br>";
```

Limites de l'héritage et de l'abstraction

- PHP ne supporte pas l'héritage multiple (une classe ne peut hériter que d'une seule classe parent).
- Mais une classe peut implémenter plusieurs interfaces.
- Ces concepts sont à utiliser avec parcimonie pour éviter une complexité excessive :
  - Restez simple.
  - Évitez les hiérarchies trop profondes ou complexes.


Animal Hierarchy Example (Over-engineered)
«animal.php»
Animal
◇ string name
◇ float size
• __construct(string $name, float $size)
• getName(): string
• setName(string $name): void
• getSize(): float
• setSize(float $size): void
• makeSound(): string
«mammal.php»
Mammal
◇ bool isVegetarian
«amphibian.php»
Amphibian
◇ bool canFreeze
Pet
• getNickname(): string
• setNickname(string $nickname): void
«fish.php»
Fish
◇ bool foundInSea
«bird.php»
Bird
◇ bool canFly
«reptile.php»
Reptile
◇ bool isVegetarian
Livestock
▫ float price
▫ float weight
• getPrice(): float
• setPrice(float $price): void
• getWeight(): float
• setWeight(float $weight): void
«cat.php»
Cat
▫ string nickname
«dog.php»
Dog
▫ string nickname
«salmon.php»
Salmon
«octopus.php»
Octopus
«bird.php»
Crow
«reptile.php»
Turtle
«lizard.php»
Lizard
«cow.php»
Cow
«dragon.php»
Dragon
Legend
Symbol | Meaning | Symbol | Meaning | Symbol | Meaning
A | Abstract class | « » | Name of the file | Circle (○) | Public field/method
C | Class | italic | Abstract method | Square (□) | Private field/method
I | Interface | | | Diamond (◇) | Protected field/method

Conclusion

- PHP est un langage de programmation à typage dynamique, largement utilisé pour le développement web.
- La programmation orientée objet (POO) est un paradigme puissant pour structurer le code.
- Les concepts avancés de la POO, tels que les interfaces, l'héritage et l'abstraction, permettent de créer des applications modulaires et maintenables.
- Utilisez les namespaces et l'autoloading pour organiser et gérer vos classes efficacement.

Questions

Est-ce que vous avez des questions ?

À vous de jouer !

- (Re)lire le support de cours.
- Explorer les exemples de code.
- Faire les exercices.
- Poser des questions si nécessaire.

➡️ Lien vers le cours

N'hésitez pas à vous entraidez si vous avez des difficultés !

Sources (1/2)

- Illustration principale par Richard Jacobs sur Unsplash
- Illustration par Aline de Nadai sur Unsplash
- Illustration par Jan Huber sur Unsplash
- Illustration par Kenny Eliason sur Unsplash
- Illustration par charlesdeluvio sur Unsplash
- Illustration par Arham Jain sur Unsplash
- Illustration par Birmingham Museums Trust sur Unsplash
- Illustration par Jack Church sur Unsplash

Sources (2/2)

- Illustration par Faris Mohammed sur Unsplash
- Illustration par Justin sur Unsplash
- Illustration par Kelly Sikkema sur Unsplash
- Illustration par Eric Prouzet sur Unsplash
- Illustration par Nikita Kachanovsky sur Unsplash