Programmation orientée objet

Cyril Rabat
cyril.rabat@univ-reims.fr

Licence 2 Informatique - Info0303 - Programmation Web 2

2023-2024

Cours n°3

Programmation orientée objet
Gestion des erreurs

Version 9 septembre 2023

Table des matières

Les classes (1/2)

- **Classe** : modèle décrivant...
  ↪ ...des caractéristiques communes
  ↪ ...des comportements communs d'un ensemble d'éléments
- **Objet** : instance d'une classe
  ↪ Généré à partir de la classe
- **Membres** :
  - Attributs (données) (ou propriétés, champs)
    ↪ Variables propres
  - Méthodes
    ↪ Fonctions propres

Les classes (2/2)

Structure générale d'une classe en PHP

```
class Personne {
  /* Attributs */
  ...
  /* Constructeur */
  ...
  /* Getters/Setters */
  ...
  /* Autres méthodes */
  ...
}
```

- Classe `Personne` contenue dans le fichier "`Personne.php`" :
  ↪ **AVEC UNE MAJUSCULE AU DÉBUT**, **SANS ACCENT !!!**
- Une classe par fichier (sauf classées privées, etc.)
  ↪ Permet de réaliser des `autoload`

Attributs

- Correspondent à des variables, propres à un objet
- Définition en début de classe (de préférence)
- Syntaxe :
  - Modificateur de portée : `private`, `public`, `protected`
  - Typage (avec un `?` si nécessaire)
  - Le nom de l'attribut (avec le `$`)
  - Une valeur par défaut
- Accès via la pseudo-variable `$this`
  ↪ Obligatoire (contrairement à *Java*, ...)

```
class Personne {

    private string $prenom;
    private string $nom;
    ...
}
```

Attributs en lecture seule

- Possible depuis PHP 8.1 de spécifier un attribut en lecture seule
- Utilisation du mot-clef `readonly`
- Permet de spécifier une valeur qui ne peut plus être modifiée une fois initialisée
  ↪ L'initialisation doit être effectuée dans la classe !

Attention

- `readonly` ne peut pas être utilisé en spécifiant une valeur par défaut...
  ↪ ...sinon, c'est une constante qu'il faut utiliser !

Méthodes

- Fonctions propres à une classe
- Utilisation du mot-clé `function`
- Paramètres et retour typés
- Permet d'accéder aux attributs de l'objet (mêmes privés)

```
...
public function complimenter(string $compliment) : void {
  echo "{$this->prenom} {$this->nom} {$compliment}";
}
...
```

Le constructeur

- Nom de la méthode : `__construct`
- Pas de typage du retour
- Un seul constructeur par classe
  ↪ Si nécessaire, on peut utiliser les valeurs par défaut des attributs/paramètres

```
...
public function __construct(string $prenom = "John", string $nom = "Doe") {
  $this->prenom = $prenom;
  $this->nom = $nom;
}
...
```

Instancier un objet

- Pour instancier un objet : opérateur `new`
- Appel du constructeur et retour d'une référence sur l'objet
- Accès à une méthode/attribut (publique) : ->

```
class Personne {
  ...
}
$p = new Personne("Cyril", "Rabat");
$p->afficher();
print_r($p);

// Cyril Rabat
// Personne Object ( [prenom:Personne:private] => Cyril [nom:...
```

Fichiers séparés

- Pour utiliser une classe :
  - Définition de la classe dans le script
    ↪ Problème car non réutilisable
  - Utilisation d'un script spécifique
    ↪ Nom du fichier = nom de la classe + extension `.php`
- Inclusion d'un script PHP :
  - `include` (ou `require`)
  - `include_once` (ou `require_one`) : inclusion unique
- Toutes les fonctions et variables du script sont incluses

Chargement automatique

- Définition d'une fonction de chargement automatique
- Enregistrement de cette fonction avec `spl_autoload_register`

```php
<?php
function charge(string $nomClasse) : void {
  include $nomClasse. '.php';
}
spl_autoload_register('charge');

$p = new Personne("Bob", "Bob");
```

Cette solution permet de gérer des répertoires multiples.

Retour sur les modificateurs de portée

- Permettent de protéger les attributs :
  ↪ Évite la modification non contrôlée
  ↪ Assure que l'objet est dans un état stable
- Pour récupérer les valeurs, utilisation de *getters* :
  ↪ Méthodes retournant la valeur d'un attribut
- Pour modifier les valeurs, utilisation de *setters* :
  ↪ Méthodes prenant en paramètre la nouvelle valeur

Les *getters*

- Retourne la valeur des attributs
- Nom : commencent par `get` suivi par le nom de l'attribut avec une majuscule au début

```
class Personne {
...
  public function getPrenom() : string {
    return $this->prenom;
  }
  public function getNom() : string {
    return $this->nom;
  }
...
}
```

Les *setters*

- Modifient les valeurs des attributs
- Nom : commencent par `set` suivi par le nom de l'attribut avec une majuscule au début

```
class Personne {
...
  public function setPrenom(string $prenom) : void {
    $this->prenom = $prenom;
  }
  public function setNom(string $nom) : void {
    $this->nom = $nom;
  }
...
}
```

Constantes

- Possible de définir des constantes dans la classe
- Syntaxe :
  ↪ Mot-clé `const`, nom (sans $), "=", valeur
  ↪ Possible d'utiliser un modificateur de portée
- Accès avec l'opérateur "::"

```php
class A {

  const PI = 3.14159265359;

}
```

```php
echo "La valeur de PI est ".
     A::PI."<br/>";
```

Membres de classe (1/2) : définition

- Membres d'instance :
  - Nécessite d'instancier un objet pour y accéder
  - Propres à chaque objet
- Membres de classe :
  - Communs à tous les objets de la classe
  - Pas d'accès à `$this`
- Accès avec l'opérateur "`::`"
  ↪ Utilisation de `self` au sein de la classe
- Mot-clé pour déclarer un membre de classe : `static`

Membres de classe (2/2) : exemple

```php
class Cercle {
    const PI = 3.1415;

    public static function getPerimetre(float $rayon) : float {
        return 2 * self::PI * $rayon;
        // ou return 2 * Cercle::PI * $rayon;
    }
}


echo "Périmètre␣du␣cercle␣unité␣:␣".Cercle::getPerimetre(1.0)."<br/>";
```

Qu'est-ce qu'une méthode magique ?

- Méthodes que l'on peut redéfinir et qui possèdent des comportements par défaut
- Commencent toutes par "__" (deux "_")
- Exemples :
  - `__construct`, `__destruct` : constructeur et destructeur
  - `__toString` : conversion en `string`
  - `__clone` : copie d'un objet
  - `__set`, `__get`, `__isset` et `__unset` :
    ↪ Appelées lors de la modification, récupération, etc. de propriétés inaccessibles

Dans ce cours, nous ne traiterons que les trois premiers points.

Méthode __toString

- Permet de personnaliser la conversion en chaîne de caractères
- Retourne une chaîne de caractères

```
class Personne {
...
  public function __toString() : string {
    return $this->prenom."␣".$this->nom;
  }
...
}
$p = new Personne("Cyril", "Rabat");
echo $p;

// Sortie : Cyril Rabat
```

Cloner un objet

- Utilisation de l'opérateur `clone`
- Possible de redéfinir le comportement par défaut :
  ↪ Redéfinition de la méthode magique `__clone`

```php
$p1 = new Personne("Cyril", "Rabat");
$p2 = clone($p1); // Ou clone $p1
$p1->setNom("Lignac");
echo "$p1 et $p2";

// Sortie : Cyril Lignac et Cyril Rabat
```

Détruire un objet

- Méthode `__destruct`
- Appelée dès qu'un objet est détruit
  ↪ Exemple : lorsqu'un objet temporaire est créé dans une fonction
- Utilisés dans des cas très particuliers :
  ↪ Permet de réaliser des actions de sauvegarde, de libération de mémoire...

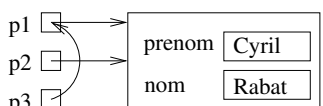
Quelques fonctions utiles

- `get_class($this)` : retourne le nom de la classe
  ↪ Fonctionne avec tout objet
- Idem : `__CLASS__` retourne le nom de la classe
- `class_exists` : vérifie si une classe a été définie
  ↪ Inutile si l'`autoload` a été bien configuré
  ↪ Même chose pour `interface_exists` et `trait_exists`
- `method_exists` : vérifie si une méthode existe
- `property_exists` : vérifie si un attribut (propriété) existe
- Toutes les méthodes pour l'introspection :
  `https://www.php.net/manual/fr/ref.classobj.php`

Affectation d'une variable avec un objet (1/2)

- En PHP, la variable référence l'objet
  ↪ Différent des types primitifs ou des tableaux
- En cas d'affectation, seule la référence vers l'objet est copiée

```php
$p1 = new Personne("Cyril", "Rabat");
$p2 = $p1;
$p3 = & $p1;
```



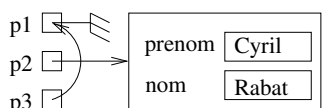
Affectation d'une variable avec un objet (2/2)

- En PHP, la variable référence l'objet
  ↪ Différent des types primitifs ou des tableaux
- En cas d'affectation, seule la référence vers l'objet est copiée

Exemple d'affectations

```
$p1 = new Personne("Cyril", "Rabat");
$p2 = $p1;
$p3 = & $p1;

$p1 = null;
```

Illustration mémoire



Passage de paramètre

- Comme pour l'affectation : passage de la "référence"
  ↪ Objet modifiable dans la fonction
- Si on passe l'adresse :
  ↪ Variable modifiable dans la fonction (nouvelle instanciation possible)

```
function modifie(Personne $p) : void {
  $p->setNom($p->getNom()."␣(modifié)");
}
$p1 = new Personne("Cyril", "Rabat");
echo "Avant␣:␣$p1␣et␣après␣:␣";
modifie($p1);
echo "$p1<br/>";

// Sortie : Avant : Cyril Rabat et après : Cyril Rabat (modifié)
```

Espaces de noms

- Possible d'organiser les classes dans des espaces de noms
  ↪ Pratique lorsque des classes portent le même nom
- Déclaration avec `namespace exemple;`
  ↪ Premier élément du script
  ↪ Non sensible à la casse
- Pour utiliser un élément dans un espace de noms :
  ↪ Préfixe `espace\element`
- Possible de créer des sous-espaces
  ↪ Exemple : `namespace exemple\sousexemple`
- La constante magique `__NAMESPACE__` indique l'espace de noms courant

Espaces de noms : exemple

```php
<?php
// script exemple.php
namespace exemple;

const EXEMPLE = 1;
class Exemple {}
function exemple() : void {}
```

```php
<?php
// Autre script
include "exemple.php";

echo "Valeur␣:␣".EXEMPLE."<br/>"; // Erreur
echo "Valeur␣:␣".exemple\EXEMPLE."<br/>"; // Fonctionne

$obj = new exemple\Exemple();
print_r($obj);
```

Utilisation et alias

- Par défaut, si on se trouve dans un espace de nom, tous les appels utilisent l'espace courant
- Pour utiliser l'espace global : \
  ↪ En cas de conflit avec les éléments de l'espace de noms
- Création d'alias à l'aide de `use`
  ↪ `use const exemple\EXEMPLE`
  ↪ `use function exemple\exemple`
  ↪ `use exemple\sousespace`
- Pour spécifier un autre nom, utilisation de `as`
  ↪ Permet de renommer une classe, par exemple

Alias : exemple

```php
<?php
include "sousespaces.php";

// Utilisation d'un sous-espace
use exemple\sousexemple2;

echo "Valeur : ".sousexemple2\EXEMPLE."<br/>";

// Utilisation d'une constante
use const exemple\sousexemple1\EXEMPLE;

echo "Valeur : ".EXEMPLE."<br/>";

// Utilisation d'une classe avec un alias
use exemple\sousexemple1\Exemple as Toto;

$obj = new Toto();
print_r($obj);
echo "<br/>";
```

L'héritage (1/2)

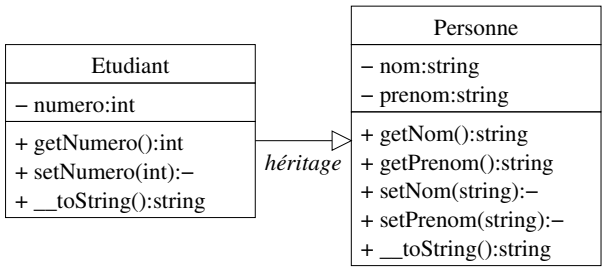
- Objectifs multiples :
  ↪ Partage du code
  ↪ Réutilisabilité
  ↪ Factorisation
- Relation de généralisation / spécialisation
- En PHP : pas d'héritage multiple
- Utilisation du mot-clé `extends`

```
class Etudiant extends Personne {
  ...
}
```

L'héritage (2/2)

- Transmission des membres :
  - Héritage de tous les membres
  - `public` : accès total par la classe fille
  - `private` : pas d'accès par la classe fille
  - `protected` :
    ↪ Accès comme s'ils étaient "`public`" pour la classe fille
    ↪ Pas d'accès pour les autres classes
- Constructeur dans la classe fille :
  - Appel du constructeur de la classe mère si nécessaire :
    ↪ `parent::__construct(...)` :
    ↪ Première instruction du constructeur (de préférence)
  - Initialisation des attributs de la classe fille

Exemple (1/2)



- Un étudiant **est une** personne
- Il possède un numéro d'étudiant en plus des nom et prénom

Exemple (2/2)

```php
class Etudiant extends Personne {
    private int $numero;

    public function __construct(string $nom, string $prenom, int $numero) {
        parent::__construct($nom, $prenom);
        $this->numero = $numero;
    }

    public function getNumero() : int {
        return $this->numero;
    }

    public function setNumero(int $numero) : void {
        $this->numero = $numero;
    }
    ...
}
```

Redéfinition de méthodes

- Possible de redéfinir une méthode existante dans la classe mère :
  ↪ Sauf si la méthode est `final`
- Même signature que dans la classe mère
  ↪ Sinon erreur car surcharge interdite !
- Depuis la classe fille, possible d'appeler celle de la classe mère :
  ↪ Instruction : `parent::nomDeLaMethode(...)`
- De l'extérieur : seule la méthode redéfinie est accessible

```
class Etudiant extends Personne {
  ...
    public function __toString() : string {
        return parent::__toString()."␣($this->numero)";
    }
  ...
```

Polymorphisme

- Lors d'un appel de méthode :
  ↪ La méthode est définie dans la classe...
  ↪ ...ou dans la classe mère (voire plus "haut" dans la hiérarchie)
  ↪ Soit les deux : redéfinition
- En cas de redéfinition, appel à la méthode la plus spécifique

```php
// Exemple de polymorphisme
$p = new Etudiant("Cyril", "Rabat", 12345);
echo $p;

// Affichage : Cyril Rabat (12345)
```

Typage dynamique

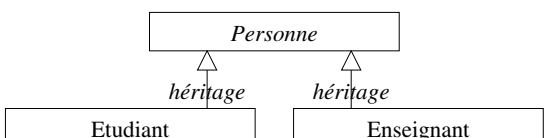
- Quand une classe est spécifiée comme type de paramètre (ou retour) :
  - Possibilité de spécifier `null`
  - Un objet de cette classe...
  - ...ou de toute classe qui en hérite

```
function compliment(Personne $p) : void {
  echo "'".$p->getNom()."' est un joli nom<br/>";
}
$etudiant = new Etudiant("Cyril", "Rabat", 123456);
compliment($etudiant);
```

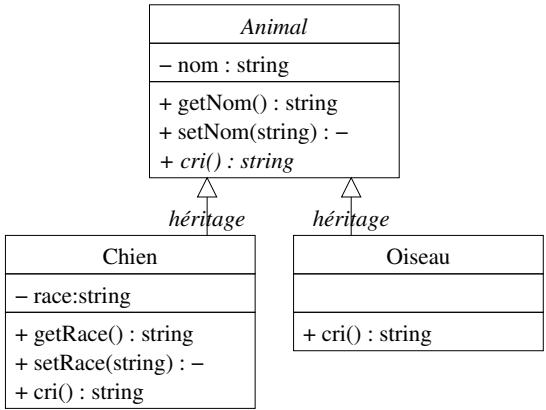
Si `null` est passé à la fonction `compliment`,
cela entraîne une erreur fatale.

Une classe abstraite

- Classes dans lesquelles des méthodes sont déclarées mais non définies
  ↪ Utilisation du mot-clé `abstract`
- Une classe abstraite ne peut pas être instanciée
- Une classe fille qui hérite d'une classe abstraite :
  - Peut utiliser le constructeur de la classe mère
  - Est abstraite si les méthodes abstraites ne sont pas définies



Exemple complet (1/4)



Exemple complet (2/4)

```php
abstract class Animal {

  private string $nom;

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

  public function getNom() : string {
    return $this->nom;
  }

  public function setNom(string $nom) : void {
    $this->nom = $nom;
  }

  public abstract function cri() : string;
}
```

Exemple complet (3/4)

```php
class Chien extends Animal {

    private string $race;

    public function __construct(string $nom, string $race) {
        parent::__construct($nom);
        $this->race = $race;
    }

    public function getRace() : string { return $this->race; }

    public function setRace(string $race) : void {
        $this->race = $race;
    }

    public function cri() : string {
        return "Ouah␣!␣Ouah␣!";
    }

}
```

Exemple complet (4/4)

```php
$animal = new Animal("Médor");

// Classe Animal abstraite => pas d'instanciation possible donc erreur !


$chien = new Chien("Médor", "Caniche");
echo $chien->cri();

// Sortie : Ouah ! Ouah !
```

Les interfaces

- Une interface est une classe abstraite sans donnée
- Intérêt : définit un contrat de programmation
  ↪ Liste de méthodes qui doivent être implémentées
  ↪ Toutes publiques !
- Peut contenir des constantes
- Mot clé : `interface`
- Pour implémenter une interface : `implements`
  ↪ Possible d'implémenter plusieurs interfaces
- Une interface peut hériter d'une autre :
  ↪ Une classe qui implémente l'interface "fille" implémentera les méthodes des deux interfaces

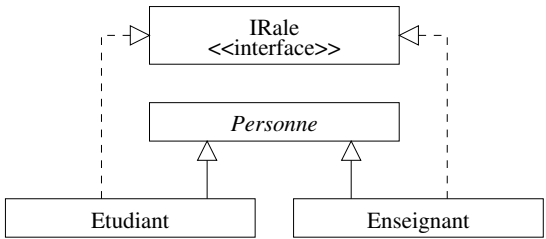
Les interfaces : exemple

```
interface IRale {

    private function raler() : void;

}


class Etudiant implements IRale {
...
    private function raler() : void {
      echo "<p>Pas content ! Pas content !</p>";
    }

}
```

Les traits

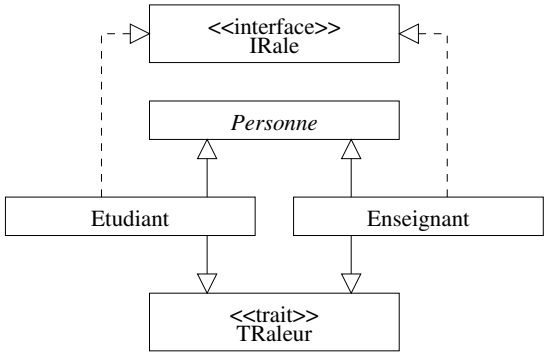
- Permettent de factoriser le code *horizontalement*
  ↪ Contrairement à l'héritage
- Mot clé : `trait`
- Dans la classe, ajout de `use` suivi du nom du trait
- Le trait peut contenir :
  - Des attributs
    ↪ Attention aux conflits
  - Des méthodes qui utilisent les attributs de la classe

Exemple d'utilisation



La méthode `raler` *doit être présente dans les classes* `Etudiant` *et* `Enseignant` *(même si le code est identique)*

Exemple d'utilisation



La méthode `raler` est présente dans le trait uniquement

Exemple

```
trait TRaleur {
    public function raler() : void {
        echo "Je␣râle␣donc␣je␣suis...<br/>";
    }
}


class Etudiant extends Personne {
    use TRaleur;

    ...
}
```

Un peu plus loin

- Plusieurs traits peuvent être utilisés :
  ↪ `use TRaleur, TJamaisContent;`
- Il est possible de modifier la portée et le nom des méthodes :
  ↪ `use TRaleur { raler as public ralerUnPeu; }`
- La composition de traits est acceptée (utilisation d'un trait dans un autre trait)
- Quelques subtilités avec les variables `static`
  ↪ Elles ne sont pas partagées entre les classes

Les énumérations

- Apparues depuis la version 8.1
- Mot-clef `enum`
- Accès aux éléments avec `::`

```php
<?php
enum Couleur {
    case Rouge;
    case Bleu;
    case Blouge;
}

function afficher(Couleur $couleur) {
    echo match($couleur) {
        Couleur::Rouge => "Je veux une voiture rouge.",
        Couleur::Bleu => "Je veux une voiture bleue.",
        Couleur::Blouge => "J'hésite entre une voiture blue et rouge."
    };
}
afficher(Couleur::Blouge);
```

Le typage (*Backed enumerations*) (1/2)

- Possible de spécifier un type
- Attribution des valeurs avec '='
- Accès à la valeur avec «l'attribut» `value`

```php
<?php
enum Couleur : string {
    case Rouge = "rouge";
    case Bleu = "bleu";
    case Blouge = "blouge";
}

function afficher(Couleur $couleur) {
    echo "Je veux une voiture {$couleur->value}.";
}
afficher(Couleur::Blouge);
```

Le typage (*Backed enumerations*) (2/2)

- Les énumérations implémentent une interface `BackedEnum`
- Deux méthodes :
  - `from` : récupère l'énumération en fonction d'une valeur
  - `tryFrom` : idem mais retourne `null` si la valeur n'existe pas

```php
$t = "rouge";
$couleur = Couleur::from($t);
afficher($couleur);
// Affiche "Je veux une voiture rouge."

$t = "autre";
$couleur = Couleur::tryFrom($t) ?? Couleur::Blouge;
afficher($couleur);
// Affiche "Je veux une voiture blouge."
```

Un peu plus loin

- Il est possible :
  - D'écrire des méthodes (`static` ou non)
  - D'implémenter des interfaces
  - De définir des constantes
  - D'utiliser des traits
- Les énumérations ne sont pas des classes
  ↪ Pas d'héritage, pas de constructeur/destructeur
- `https://www.php.net/manual/fr/language.enumerations.php`

L'union de types : *union types*

- PHP 8.0 introduit la notion d'union de types
- Possible de déclarer plusieurs types pour un attribut (ou un paramètre)

```php
class Test {

    private int|string $valeur;

    public function __construct(int|string $valeur) {
        $this->valeur = $valeur;
    }
    public function getValeur() : int|string {
        return $this->valeur;
    }
}

$test = new Test(42);
echo "Valeur : {$test->getValeur()}.<br/>"; // Affiche Valeur : 42
$test = new Test("quarante-deux");
echo "Valeur : {$test->getValeur()}.<br/>"; // Affiche Valeur : quarante-deux
```

L'intersection de types : *intersection types*

- PHP 8.1 introduit la notion d'intersection de types
- Comme pour l'union mais impose que le paramètre implémente ou hérite des interfaces/classes spécifiées

```php
function exemple(IRale&ICrie $obj) : void {
    $obj->crier();
    $obj->raler();
}

$test = new Test;
exemple($test);
// Fonctionne si la classe Test implémente les interfaces IRale et ICrie
```

Les exceptions

- Comme les autres langages, gestion des exceptions en PHP :
  ↪ Lever une exception : `throw`
  ↪ Attraper une exception : dans un bloc `try catch`
  ↪ Bloc `finally` exécuté après le `try` ou le `catch`
- L'exception est une instance de la classe `Exception` ou d'une classe fille
- En PHP, seules les extensions orientées objet utilisent les exceptions
- Autres classes d'exception : dans la bibliothèque standard PHP

Exemple de `try catch`

```
...
if(isset($_POST['valider'])) {
  try {
    if(!isset($_POST['nombre']) || ($_POST['nombre'] == ""))
      throw new Exception("Vous devez saisir un nombre.");

    $nombre = intval($_POST['nombre']);
    if(($nombre < 1) || ($nombre > 10))
      throw new Exception("Le nombre doit être compris dans l'intervalle [1; 10].");

    echo "Vous avez saisi un nombre correct : {$nombre}";
  }
  catch(Exception $e) {
    echo $e->getMessage();
  }
}
```

Créer ses propres exceptions

- Héritage de la classe `Exception`
- Possible de capturer plusieurs exceptions à l'aide de « | »

```php
try {
  // Fonction pouvant lever les exceptions
  $cafetiere->remplir();
}
catch(DebordementException | ExplosionException $e) {
  echo $e->getMessage();
}
```

Affichage des erreurs

- Affichage des erreurs :
  - Permet de déboguer plus facilement
  - À proscrire sur un environnement de production
    ↪ Expérience utilisateur !
- Réglage de l'affichage des *Warnings*, *Errors*, etc.
  ↪ Configuration de PHP (`php.ini`)
  ↪ Modification temporaire possible (fonctions de l'API)
- Messages d'erreur à deux niveaux :
  - En mode développement :
    ↪ Permet de savoir où se situe l'erreur
    ↪ Informations techniques (requêtes, classes, etc.)
  - En mode production :
    ↪ Indique à l'utilisateur que l'action n'a pu être réalisée

Niveaux d'erreur

- PHP définit des types d'erreur :
  - `E_ERROR`, `E_WARNING`, `E_PARSE`, `E_NOTICE`, ...
- Possible de définir quelles erreurs sont reportées :
  ↪ `error_reporting(...)`
  ↪ `ini_set('error_reporting', ...`

```
// Affiche toutes les erreurs
error_reporting(E_ALL);

// Toutes les erreurs sauf les E_NOTICE
// Par défaut, dans le php.ini
error_reporting(E_ALL & ~E_NOTICE);
```

Affichage des erreurs

- En mode production, les erreurs ne sont pas affichées
  ↪ Page blanche (erreur 500)
  ↪ Dépend du fichier de configuration (`php.ini` ou autre)
- Utilisation de la directive `display_errors` avec `ini_set`
  ↪ `ini_set('display_errors', 1);`
  ↪ `echo ini_get('display_errors')`
- Attention aux petites subtilités :
  ↪ Un script PHP est parsé intégralement avant l'exécution
  ↪ La directive est alors ignorée s'il y a une erreur de syntaxe dans la page !
  ↪ Solution : passer par un script tierce

```php
error_reporting(E_ALL);
ini_set("display_errors", 1);
include("script.php");
```

Log des erreurs

- Plutôt que d'afficher les erreurs à l'écran, elles sont affichées dans des logs
- Nom du fichier spécifié via la directive `error_log`
- Activation du log via la directive `log_error`
- Sous Linux, historisation gérée via le système
  ↪ Création de logs journaliers/hebdomadaires

```php
echo ini_get('error_log'); // Affiche c:/wamp64/logs/php_error.log
```

Générer une erreur utilisateur

- Utilisation de la fonction `trigger_error`
  - `trigger_error(string $msg, int $type = E_USER_NOTICE)`
- Possible de spécifier sa propre fonction comme gestionnaire :
  - `set_error_handler`
  - Cette fonction est appelée lors d'une erreur
  - Possible d'appeler le gestionnaire par défaut (`return 0`)

Opérateur de contrôle d'erreur

- Ajout de `@` avant une expression
- Permet d'ignorer les messages d'erreur d'une expression
- Message d'erreur stocké dans la variable globale `$php_errormsg`
- À utiliser avec précaution
  ↪ Ce n'est pas un joker pour faire du mauvais code !

```php
// En cas d'erreur et sans @, E_WARNING généré
if(($fichier = @fopen("resources/toto.txt", "r")) === NULL)
  echo "Oups␣!␣Impossible␣d'ouvrir␣le␣fichier...<br/>";
```