PROGRAMMATION ORIENTEE OBJET EN PHP

KRISTEN LE LIBOUX

JUILLET 2013

Introduction

Notions abordées

Classes, objets

Héritage

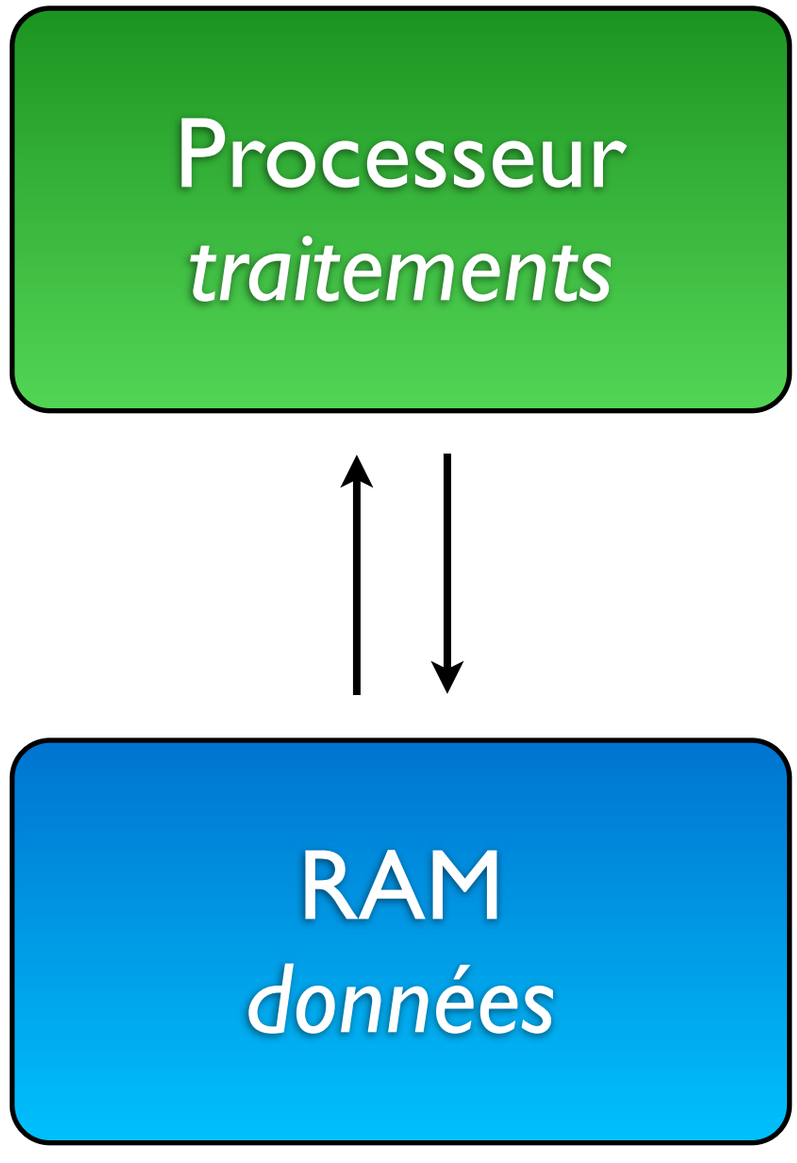
Polymorphisme

Méthodes magiques

Interfaces

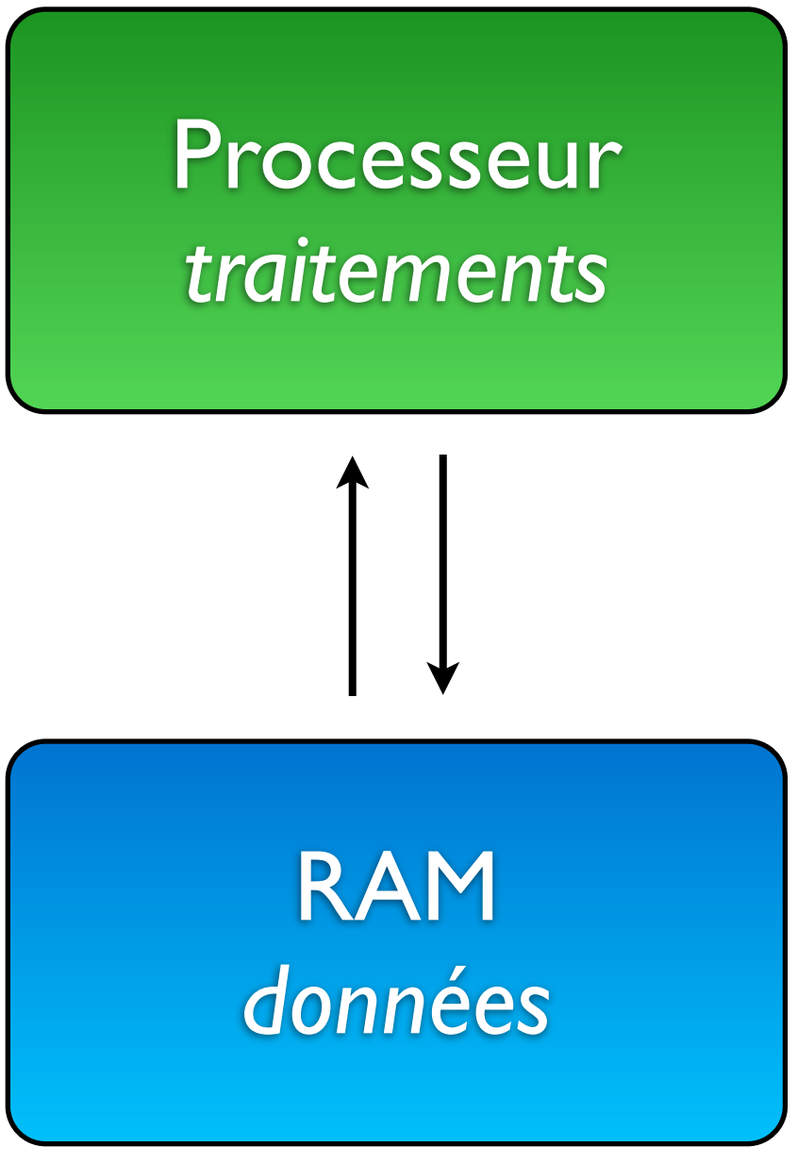
Motivation

Les deux composants essentiels d'une machine



Motivation

Il en découle deux styles de programmation



Impérative
Se concentre sur les traitements

Orientée objets (POO)
Se concentre sur les données (concepts)

Programmation impérative

```
require_once("config.php");
require_once("common.php");
$link = mysqli_connect($server, $user, $pass, $db);
$req  = mysqli_query($link, "select * from products");

displayHeader("Page d'accueil");
```

```
while( $row = mysqli_fetch_assoc($req) )
{
    echo "<li>";
    echo "    <h4>".$row["name"]."</h4>";
    echo "    <p class=\"desc\">".$row["desc"]."</h4>";
    echo "</li>";
}

displayFooter();
```

Inconvénients

Correct, fait le job, mais...

- Difficile à suivre et à lire
- Impossible de collaborer
  avec un intégrateur ou un designer
- Maintenabilité décroit à mesure que le code croit
- Ajouter une fonctionnalité, c'est souvent hacker

Programmation orientée objets

Notion centrale : la **classe**

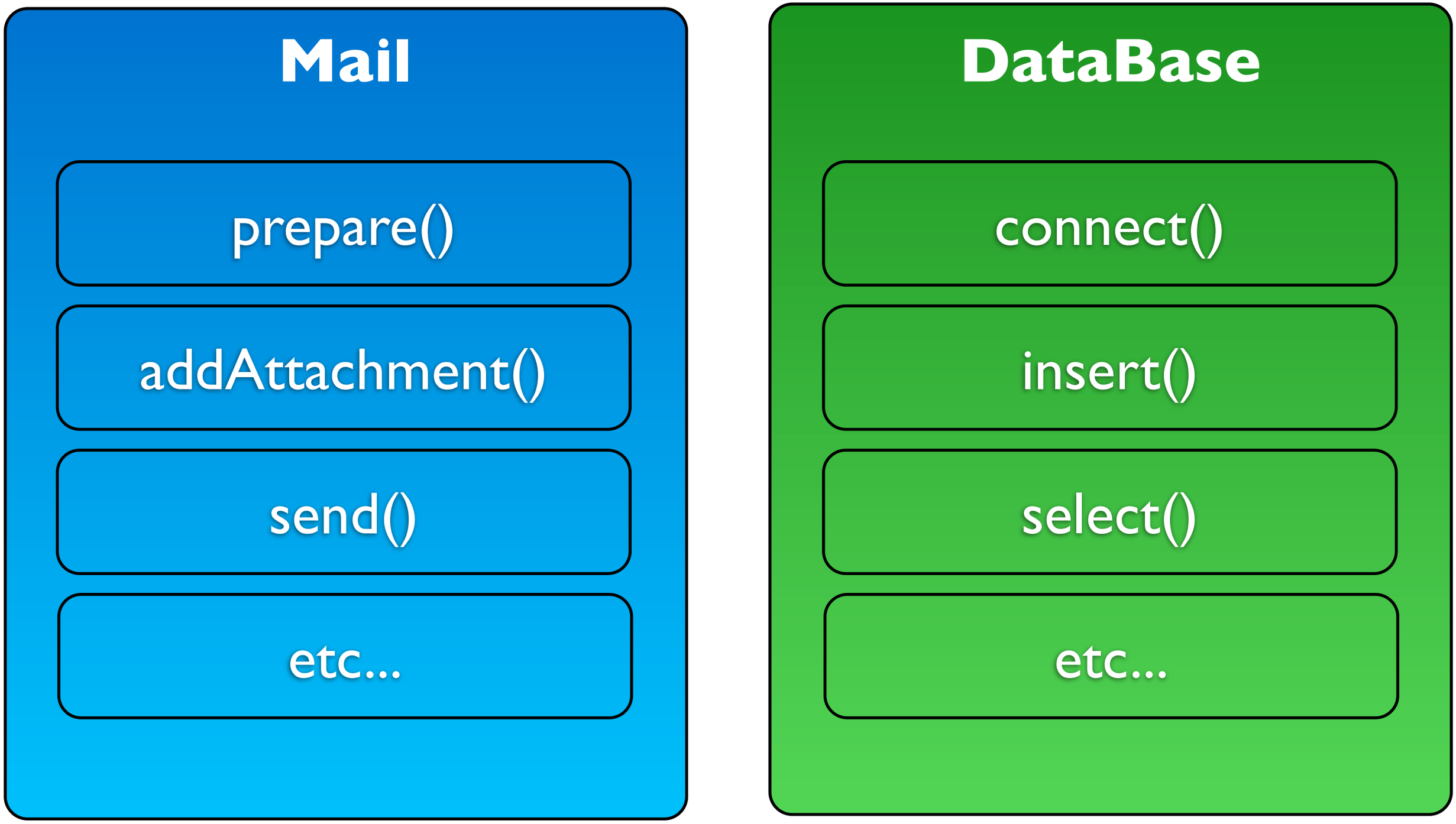
- Une classe = un concept utilisé dans un projet
  Exemple : utilisateur, base de données, produit, etc.
- On détermine des services associés à chaque classe : **méthodes**
- Comporte des données : **attributs**

Avantages

- Modularité
- Encapsulation
- Héritage
- Polymorphisme

(slides suivants)

Modularité



Séparation des données et des services par entité logique.

Encapsulation

Utilisateur

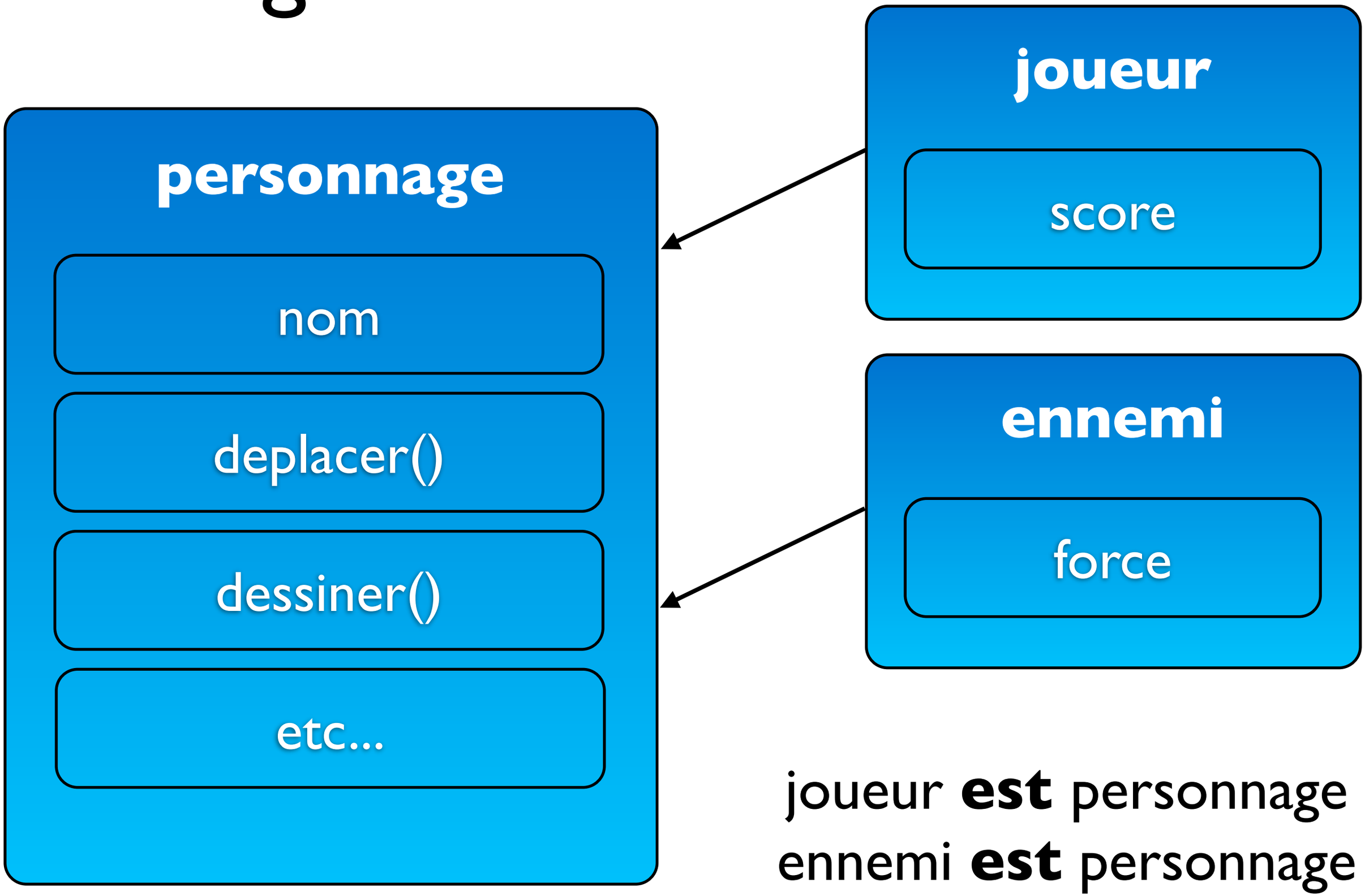
- Nom
- Mail
- Mot de passe
- etc...

Appartement

- Ville
- Nb de pièces
- Adresse
- Propriétaire

Propriétaire est une donnée de type Utilisateur associée à un Appartement

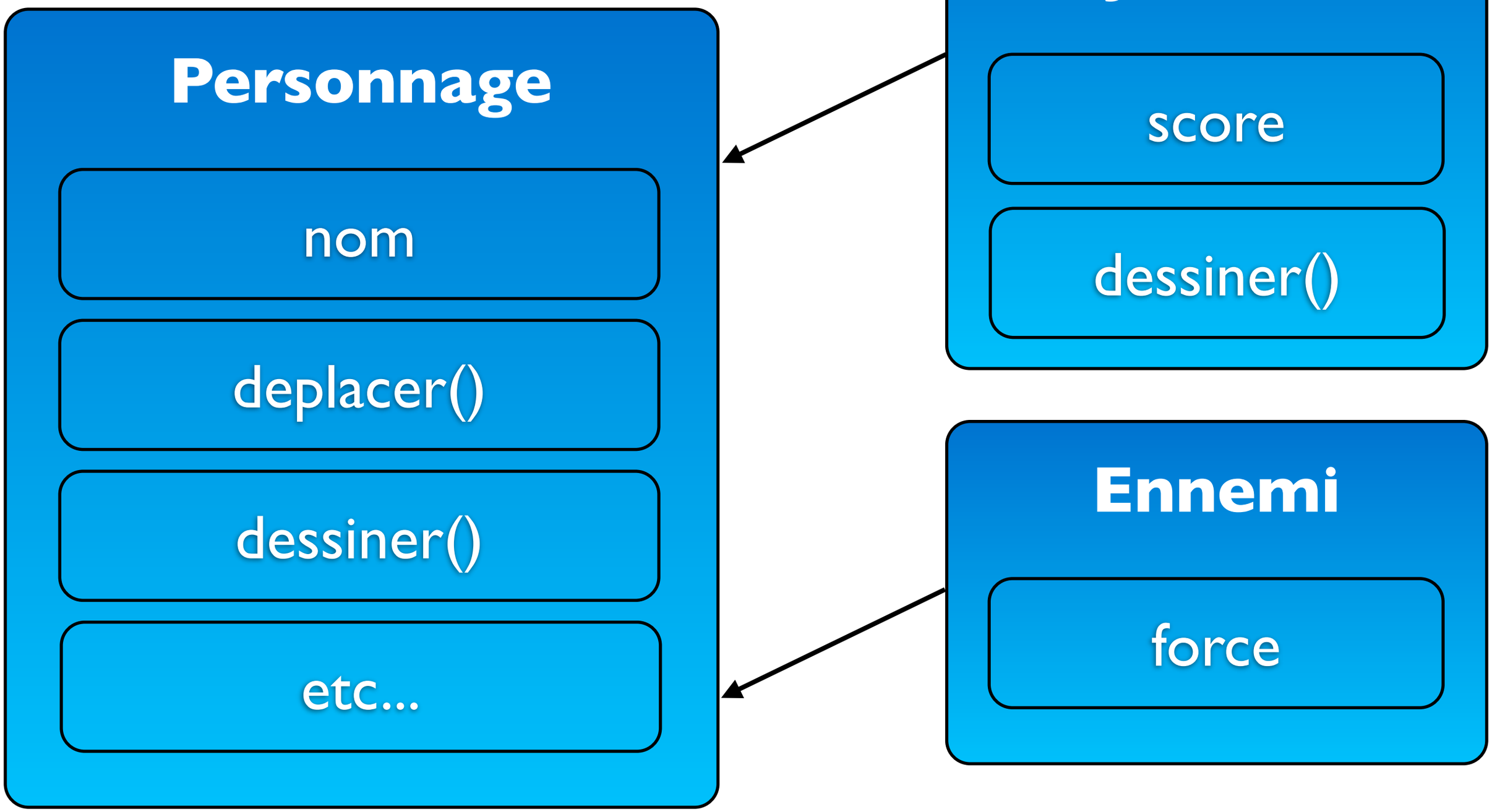
Héritage



joueur **est** personnage

ennemi **est** personnage

Polymorphisme



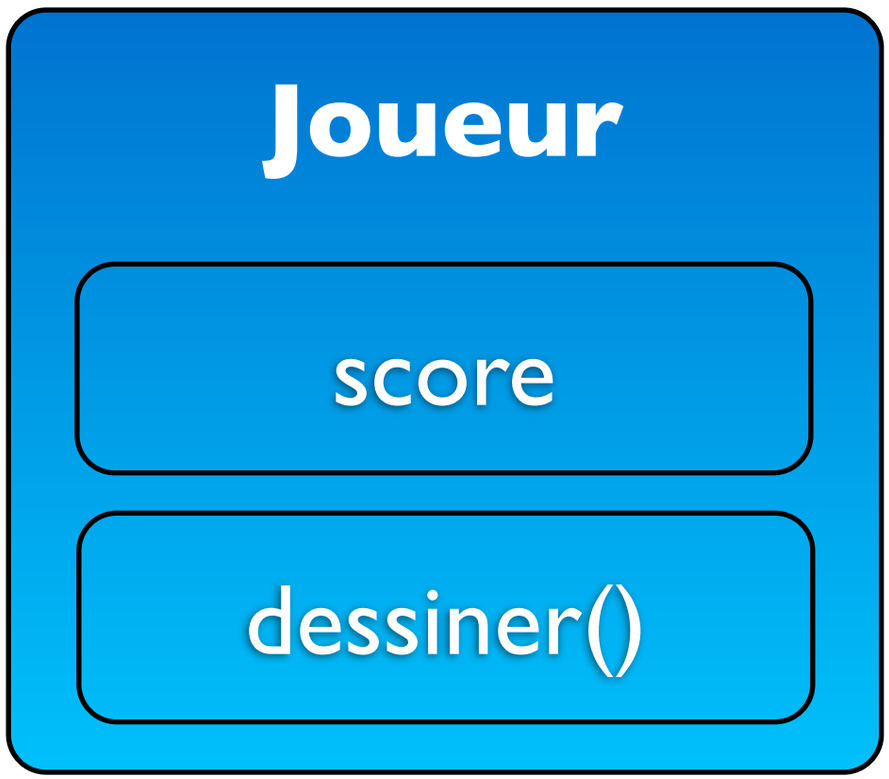
Service **dessiner()** :
générique pour un Ennemi, spécifique pour un Joueur

Vocabulaire

Classe
Un nouveau type de données,
comportant des *attributs* et des *méthodes*

Attribut
Une donnée propre à une classe

Méthode
Une fonction propre à une classe



Vocabulaire

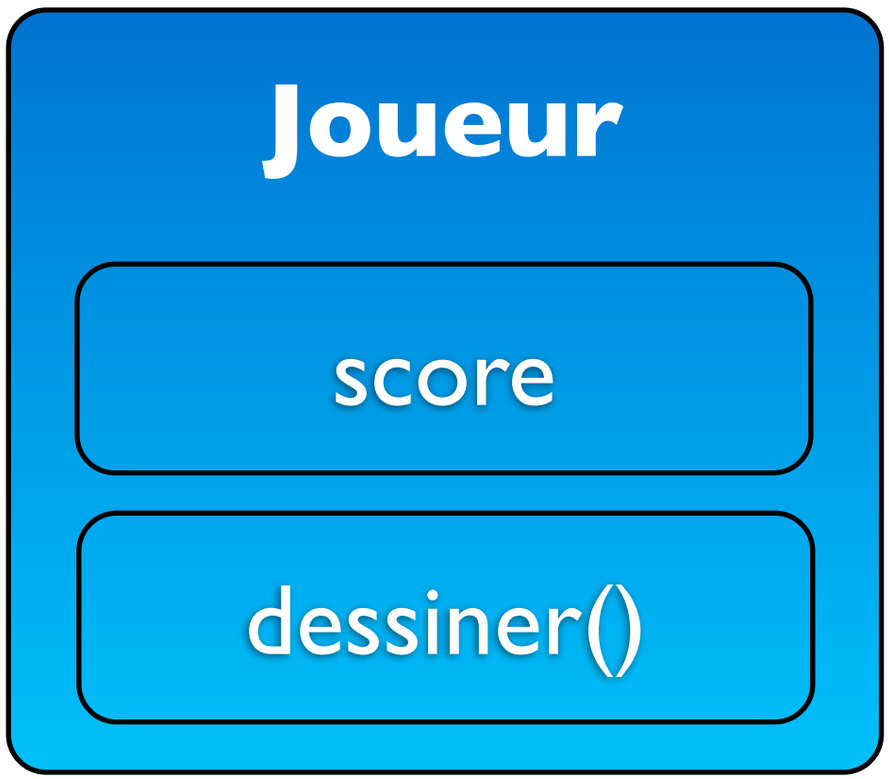
Instancier une classe
Allouer la mémoire nécessaire pour une classe

Objet
Une instance de classe

```php
$pierre = new Joueur;
$paul   = new Joueur;

$pierre->score = 10;

$paul->dessiner();
```



Vocabulaire

classe = type

Types en PHP :

- Avant : NULL, bool, int, float, string, array.
- Maintenant : Mail, Produit, Utilisateur, Voiture...

objet = instance en mémoire

En première approx. : objet = variable de type classe

Mais deux variables peuvent en fait pointer sur le même objet.

Caractéristiques

Exemple 1

```
class User {
    public $firstName = "Unknown";
    public $lastName = "";

    public function getDisplayName() {
        return $this->firstName
                ." ".$this->lastName;
    }
    public function printDisplayName() {
        echo $this->getDisplayName();
    }
}

$pers1 = new User();
$pers1->firstName = "Paul";
$pers1->lastName = "Dupont";

$pers1->printDisplayName();
```

Accès aux membres

```
$pers1->lastName = "Dupont";
$pers1->printDisplayName();
```

Membre = méthode ou attribut

Opérateur `->` pour accéder aux membres d'un objet

$this

```
echo $this->getDisplayName();
```

Une variable spéciale qui fait référence à l'instance courante.

Toujours disponible dans les méthodes, sauf pour les méthodes statiques (voir plus loin).

Instanciation

```
$pers1 = new User();
```

Crée en mémoire une variable `$pers1` de type `User` :

- Alloue la mémoire nécessaire
- Initialise les attributs

Exemple 2

```
class User {
    public $firstName = "Unknown";
    public $lastName = "";

    public function getDisplayName() {
        return $this->firstName
                ." ".$this->lastName;
    }
    public function printDisplayName() {
        echo $this->getDisplayName();
    }
}

class Admin extends User {
    public $password = "";
    public function getDisplayName() {
        return $this->firstName . " (admin)";
    }
}
```

Héritage

```
class Admin extends User
```

Les classes héritées :

- Bénéficient des membres de leurs parents
- Peuvent définir de nouveaux membres
- Peuvent redéfinir des membres (**polymorphisme**)

Exemple 2b
Polymorphisme

```
class User {
    public $firstName = "Unknown";
    public $lastName = "";

    public function getDisplayName() { ... }
    public function printDisplayName() { ... }
}

class Admin extends User {
    public $password = "";
    public function getDisplayName() { ... }
}

$me    = new Admin();
$other = new User();
echo "Message de ";
$me->printDisplayName();
echo " à ";
$other->printDisplayName();
```

⇨ `Message de Unknown (admin) à Unknown`

Appel de méthode parente

```
class Admin : extends User {
    public function getDisplayName() {
        return parent::getDisplayName() . " (admin)";
    }
}
```

- Utilisation de l'opérateur `::` de résolution de portée, avec le mot-clé `parent`

Exemple 3

```
class FormHelper {
    public $method = "post";
    private $fields = array();

    public function addField($name, $type, $label) {
        $this->fields[] = array($name, $type, $label);
    }
    public function getHTML() {
        $html = '<form method="'.$this->method.'">';
        foreach( $this->fields as $f ) {
            ...
        }
        $html.= '</form>';
        return $html;
    }
}

$f = new FormHelper();
$f->addField("name", "text", "Nom");
echo $f->getHTML();
```

Visibilité

Un membre doit être déclaré soit :

- public visible par tous, donc de l'extérieur
- protected visible par la classe et les héritiers
- private visible uniquement par la classe

Classes et méthodes finales

```
final class GlobalChecker { ... }

class User {
    final public function check($name, $pass) { ... }
}
```

- Les classes finales ne peuvent pas avoir d'héritier.
- Les méthodes finales ne peuvent pas être redéfinies par les héritiers.

Membres statiques

Une classe peut avoir des membres statiques, accessibles sans instancier la classe :

- Attributs : propres à la classe et non à l'objet
- Méthodes : `$this` interdit, ne peuvent accéder qu'aux membres statiques

Appel par l'opérateur de résolution de portée `::` et non `->`

Exemple

```
class User {
    static public function getCount() {
        ...
        mysqli_query("SELECT COUNT(*) FROM t_usr");
        ...
        return $nb;
    }
}
```

```
$count = User::getCount();
echo "Il y a $count utilisateurs inscrits !";
```

Exemple

```
class About {
    static private $version = "2.0.1";
    static public function getVersion() {
        return self::$version;
    }
}

echo "Version courante : ". About::getVersion();
```

Constantes de classe

```
class Circle extends Shape {
    const PI = 3.1415;
    public $x, $y, $r;
    public function getPerimeter() {
        return 2. * self::PI * $r
    }
}
```

- Non modifiables, pas de préfixe `$`
- Toujours publiques
- Accès interne par `self::`, externe par `Circle::`
- Valeurs littérales et types simples uniquement

Typage explicite

En anglais : *type hinting*

Permet d'imposer la classe d'un paramètre de fonction ou de méthode : à utiliser abondamment

- Auto-documente le code, limite les erreurs
- NULL interdit au moment de l'appel,
  sauf si on l'indique comme valeur par défaut
- On peut typer les objets par leur classe, les interfaces (voir plus loin), les tableaux (avec *array*), mais pas les types simples (int, float, bool, string)

Exemple typage explicite

```
class User {
    protected $id;
    public function getId() { ... }
    ...
}

class Ad {
    protected $owner;
    public function setOwner(User $user) {
        $this->owner = $user;
    }
    ...
}

function canEdit(Ad $ad, User $usr)
{
    return $usr->getId()
           === $ann->getOwner()->getId();
}
```

Méthodes magiques

Définition

- Méthodes particulières dont le nom est imposé
- Optionnellement définies par le développeur
- Nom commence par __
- Invoquées automatiquement par PHP, dans certains contextes, lorsqu'elles sont définies.

Principales méthodes magiques

__construct()

__destruct()

__get()

__isset()

__set()

__unset()

__call()

__callStatic()

__toString()

__clone()

Constructeur et destructeur

__construct()

- Appelé lors de la création de l'objet
- Sert à initialiser l'objet
- Peut avoir des paramètres :
  dans ce cas, il faut fournir des valeurs au moment du « new »

__destruct()

- Appelé lors de la destruction de l'objet
- Sert à nettoyer
- Pas de paramètres

Attributs inaccessibles ou non définis

__set($name, $value)

- Affecter une valeur

__get($name)

- Obtenir une valeur

__isset($name)

- Tester si un attribut existe (avec isset)

__unset($name)

- Appel de unset sur l'attribut

Méthodes inaccessibles ou non définies

__call($name, *array* $arg)

- Appel d'une méthode sur un objet avec ->

__callStatic($name, *array* $arg)

- Appel d'une méthode statiquement avec : :

Exemple

```
class MyReadOnlyClass
{
    protected $data = array();
    public function __construct( array $data ) {
        $this->data = $data;
    }
    public function __get( $attr ) {
        if( isset( $this->data[$attr] ) )
            return $this->data[$attr];
        trigger_error("Attribut non défini", E_USER_ERROR);
    }
    public function __set( $attr, $value ) {
        trigger_error("Lecture seule", E_USER_ERROR);
    }
    public function __isset( $attr ) {
        return isset( $this->data[$attr] );
    }
    public function __unset( $attr ) {
        trigger_error("Lecture seule", E_USER_ERROR);
    }
}
```

Utilisation

```php
$obj = new MyReadOnlyClass(
    array("login" => "ldupont", "level" => "user")
);

echo $obj->login; // (__get) affiche «ldupont»

echo isset($obj->login)?1:0; // (__isset) affiche «1»

echo isset($obj->password)?1:0; // (__isset) affiche «0»

echo $obj->password; // (__get) erreur: non défini

$obj->level = "admin"; // (__set) erreur: lecture seule
```

Copie d'objets

```
$pers2 = $pers1;
```

Ne crée pas en mémoire un nouvel objet

- Crée une nouvelle référence vers le même objet
- `$pers2` est un alias de `$pers1`

Ainsi :

```
$pers2->firstName = "Durand";
echo $pers1->firstName;  // Durand
```

Clonage

```
$pers2 = clone $pers1;
```

Crée une nouvelle instance qui vivra indépendamment

- Les attributs de `$pers1` sont copiés dans `$pers2` (au sens de l'opérateur =)
- La mémoire est allouée de nouveau
- Si la méthode magique __clone() est définie, elle sera appelée sur l'objet cloné juste après la copie

Exemple

```
class MyClass
{
    public static $nbInstances = 0;

    protected $object;

    public function __construct() {
        $this->instance = ++self::$instances;
        $this->object = new myOtherClass();
    }
    public function __clone() {
        $this->instance = ++self::$instances;
        $this->object = clone $this->object;
    }
}
```

Conversion en *string*

__toString()

- Méthode magique appelée dans tous les contextes où l'objet doit être converti en chaîne de caractères

```php
class Person
{
    public $nom = "Pierre Durand";

    public function __toString() {
        return $this->nom;
    }
}

$pers = new Person;
echo $pers; // Pierre Durand
```

Interfaces & classes abstraites

Interfaces

Un « modèle » de classe qui **spécifie** simplement les méthodes publiques à implémenter, **sans le faire**

```
interface Affichable {
    public function affiche();
}
```

```
class Person implements Affichable {
    public $nom;
    public function affiche() {
        echo $this->nom;
    }
}
```

On spécifie ici qu'un objet pourra être qualifié d'«affichable» dès lors qu'il possèdera une méthode «affiche»

Interfaces

Particularités :

- Une même classe peut implémenter plusieurs interfaces
- Peuvent hériter les unes des autres
- Peuvent définir des constantes

```
interface AA extends A {
    ...
}

class C implements AA, B, D {
    ...
}
```

Classe abstraite

Classe dont une méthode au moins est abstraite, c'est-à-dire non implémentée.

Doit être héritée, ne peut être instanciée.

Exemple

```php
abstract class BinaryOperator {
    protected $x = 0, $y = 0;
    abstract public function compute();
    public function displayResult() {
        echo $this->compute();
    }
}
```

```php
class Addition extends BinaryOperator {
    public function compute() {
        return $this->x + $this->y;
    }
}

$ope1 = new Addition(); // ok
$ope2 = new BinaryOperator(); // erreur !
```

Exemple synthétique

Boîtes d'encombrement 2D

Soit une forme géométrique en 2D, on appelle boîte d'encombrement un rectangle (idéalement minimal) qui la contient. Ici un rectangle est défini par 2 coins opposés (c'est-à-dire que les côtés sont parallèles aux axes).

Boîtes d'encombrement 2D

Définir deux interfaces :

- iShape : une forme 2D (point, droite, rectangle, etc)
- iBoundingBox : une boite d'encombrement 2D

iBoundingBox

- getCoords()
- isEmpty()
- mergeWith($bbox)
- equals($bbox, $tol)

iShape

- computeBoundingBox($shapes)
- getBoundingBox()

On souligne les méthodes statiques.

Boîtes d'encombrement 2D

Implémenter **computeBoundingBox** :

- définir une classe abstraite Shape (implements iShape)
- implémenter Shape::computeBoundingBox

Boîtes d'encombrement 2D

Implémenter des formes :

- Rectangle (à la fois Shape et BoundingBox)
- Point
- Cercle
- Triangle

Boîtes d'encombrement 2D

Tester :

```
$pt   = new Point(0, 0);
$rect = new Rectangle(20, 10, 50, -20);
$tri  = new Triangle(50, -20, 70, 0, 40, 30);
$circ = new Circle(80, 20, 50);

$shapes = array( $pt, $rect, $tri, $circ );
$bbox = Shape::computeBoundingBox( $shapes );
check( $bbox, array(0, 70, 130, -30) );

$shapes = array();
$bbox = Shape::computeBoundingBox( $shapes );
echo $bbox->isEmpty() ? "ok" : "erreur";
```

Corrigé

https://github.com/kleliboux/code-samples

Fonctions utiles

Opérateur *instanceof*

`x instanceof y` est true si :

- **x** est un objet de classe **y**
  (y compris par héritage)
- **x** est un objet dont la classe implémente l'interface **y**
  (y compris par héritage)

Remarque : **y** peut être un nom littéral ou une variable de type *string*

Exemple

```php
interface iA { }

class A implements iA { }

class AB extends A { }

class B {}

$ab = new AB();

$str = "AB";

// tout imprime TRUE sauf le 2ème
var_dump( $ab instanceof AB   );
var_dump( $ab instanceof B    );
var_dump( $ab instanceof iA   );
var_dump( $ab instanceof A    );
var_dump( $ab instanceof $str );
```

Tests d'existence

Tester l'existence d'une classe :

```
bool class_exists ( string $class_name )
```

Tester l'existence d'une méthode
(indépendamment de sa visibilité) :

```
bool method_exists ( mixed $object ,
                     string $method_name )
```

Tests d'« appelabilité »

```
bool is_callable ( callable $name )
```

- Le paramètre est soit :

  Une chaîne (vérifie l'appelabilité d'une fonction)

  Un tableau array(*objet*, *chaine*) ou (*chaine*, *chaine*) (vérifie l'appelabilité d'une méthode)

- Ce genre de paramètre s'appelle «callback» ou «callable» dans la doc PHP

Exemple

```php
function f() {}
class A {
  public function fpub() {}
  protected function fprot() {}
  static public function fstat() {}
}

class B extends A {
  public function b() {
     var_dump( is_callable( array($this, "fprot") ) );
  }
}

$b = new B();

var_dump( is_callable ( "f" ) ); // true
var_dump( is_callable ( array("A", "fstat" ) ) ); // true
var_dump( is_callable ( array("A", "fpub" ) ) ); // true!
var_dump( is_callable ( array($b, "fpub" ) ) ); // true
var_dump( is_callable ( array($b, "fprot" ) ) ); // false

$b->b(); // true
```

Exécution de *callback*

```
mixed call_user_func ( callable $callback
                       [, mixed $parameter
                       [, mixed $... ]] )
```

- Le premier paramètre est un callback
- Les suivants sont les paramètres du callback
- Le callback sera appelé, s'il existe.
- Variante :

```
mixed call_user_func_array (
            callable $callback, array $params )
```

Exemple

```php
class ActionManager {
    public function Load($id) { ... }
    public function Save($id) { ... }
    public function Delete($id) { ... }
    public function Run($id) { ... }
}

$mgr = new ActionManager();
$act = $_GET["action"];
$id  = intval( $_GET["id"] );

if( method_exists(array($mgr, $act)) ) {
    call_user_func(array($mgr, $act), $id);
}
else {
    throw new ErrorException("Action inexistante");
}
```

Remerciements

Cette présentation est partiellement issue de la traduction de celle de Jason Austin, « Object Oriented Principles in PHP5 » :

http://fr.slideshare.net/jfaustin/object-oriented-php5

Des exemples originaux ont été ajoutés.

MERCI

@NOVLANGUE SUR TWITTER

COMMENTAIRES, DISCUSSIONS, QUESTIONS