Université Montpellier-II
IUT de Béziers
Département MMI

UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

M3203 : Programmation par objets

Intervenant : Chouki TIBERMACINE
Bureau : A119
Mél. : Chouki.Tibermacine@iutbeziers.fr
Web : http://www.lirmm.fr/~tibermacin/ens/m3203/

2014-2015

Cours 2

Partie 1 : Héritage entre classes en PHP

Plan du cours

- Héritage : mécanisme de réutilisation de code
- Héritage : mécanisme d'abstraction
- Principes de l'héritage
- Restriction « *final* » et visibilité « *protected* »

Plan du cours

- Héritage : mécanisme de réutilisation de code
- Héritage : mécanisme d'abstraction
- Principes de l'héritage
- Restriction « *final* » et visibilité « *protected* »

Exemple de réutilisation

- On dispose de la classe Pile vue dans le premier cours
- On veut maintenant définir une classe PileBornee
  - La pile a donc une capacité maximale qu'elle ne pourra pas dépasser
  - Avant d'empiler, vérifier que la pile n'est pas pleine

Solution 1 : pile bornée est composée d'une pile ordinaire

- Déclarer un attribut pile dans la classe PileBornee
- Déclarer un attribut représentant la capacité maximale de la pile
- Définir un constructeur pouvant recevoir la capacité maximale de la pile
- Réécrire toutes les méthodes d'une pile :
  - empiler(...)
  - depiler()
  - est_vide()
  - sommet()

Mise en œuvre de la solution 1

```
require "Pile.php";
class PileBornee {
    public $pile; private $taille; public $capacite;
    public function __construct($capacite=10) {
        $this->pile = new Pile();
        $this->taille = 0;
        $this->capacite = $capacite;
    }
    public function empiler($elem) {
        if($this->taille > $this->capacite) print(...); return -1;
        $this->pile->empiler($elem); // Délégation
        $this->taille++;
    } ... }
```

Limites de cette solution

- Ré-écrire **toutes les méthodes** de la pile ordinaire :
  - en ajoutant les contrôles qu'il faut pour faire en sorte que la pile soit bornée (voir la méthode empiler(…), par ex.)
  - puis en déléguant les appels aux méthodes de l'attribut pile :

```
public function depiler() {
    $this->taille--;
    return $this->pile->depiler();
}
public function est_vide() {
    return $this->pile->est_vide();
}
public function sommet() {
    return $this->pile->sommet();
}
```

Solution 2 : pile bornée hérite de pile ordinaire

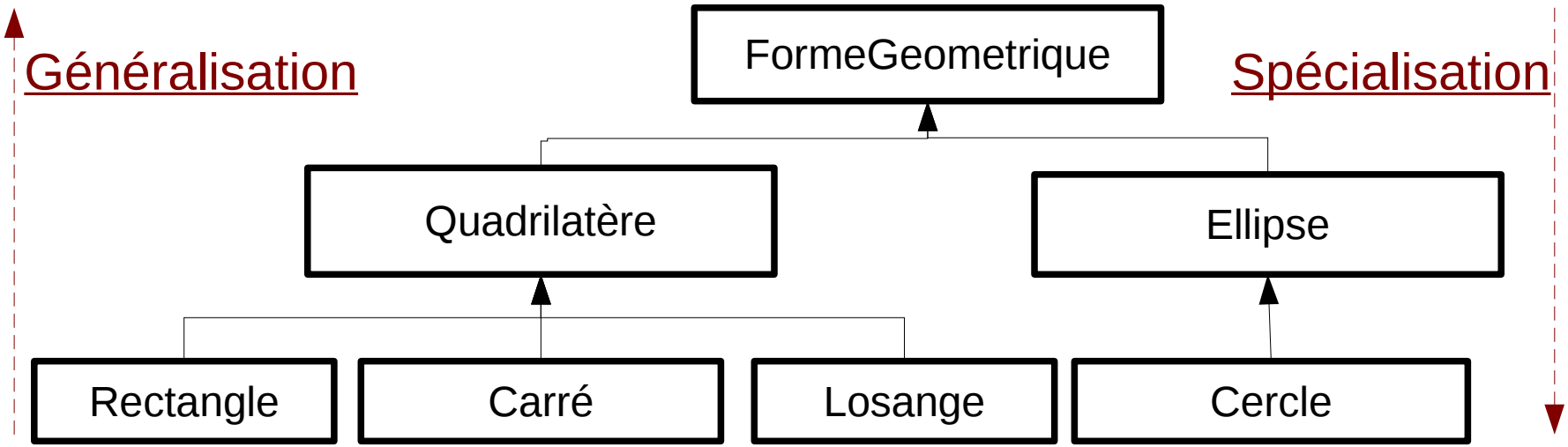
- Ici, on déclare la classe PileBornee comme étant une classe qui hérite de tous les membres de la classe Pile
- La classe PileBornee n'a donc à **re-définir** que les méthodes empiler et dépiler (parce que la 1ère fait un traitement spécifique pour les piles bornées par rapport aux piles ordinaires et la seconde met à jour l'attribut $taille)
- Les autres méthodes (où il n'y avait que de la délégation) seront héritées et donc pas besoin de les écrire
- Dans l'exemple, on a économisé environ 26 % des lignes de code de la classe PileBornee 

Plan du cours

- Héritage : mécanisme de réutilisation de code
- Héritage : mécanisme d'abstraction
- Principes de l'héritage
- Restriction « *final* » et visibilité « *protected* »

Exemple d'illustration

- Nous disposons d'un certain nombre de classes pour représenter des formes géométriques
- Une classe pour chaque concept : rectangles, cercles, triangles, carrés, ...



Mécanisme d'abstraction

- Hiérarchisation des abstractions :
  - Des classes représentant des concepts généraux aux classes représentant des concepts spécifiques :

    Forme géométrique -> Quadrilatère -> Rectangle

  - Modélisation/Compréhension plus facile du domaine

  - Classification des concepts du domaine plus proche de la réalité (la manière naturelle utilisée par les humains)

Mécanisme d'abstraction -suite-

- Factorisation de code :
  - Code commun aux concepts est factorisé dans les classes ancêtres (pas de code répété entre les classes)
  - Les classes les plus spécifiques héritent les membres des classes les plus générales :

    Toute forme géométrique a une couleur
    ->
    $couleur est un attribut de la classe FormeGeometrique (hérité par toutes les autres classes)

Plan du cours

- Héritage : mécanisme de réutilisation de code
- Héritage : mécanisme d'abstraction
- Principes de l'héritage
- Restriction « *final* » et visibilité « *protected* »

Principe général

- Une classe B qui « hérite » d'une autre classe A
  (on dit aussi B « étend » A, ou B « sous-classe » de A) :
  - B « récupère » dans sa définition tous les attributs et méthodes qui ont été définis dans A

- Ensuite, il devient possible d'ajouter des membres :
  - Raffiner un concept : Travailleur « hérite de » Personne
    (ajouter entre autres les attributs : employeur, salaire, ...)

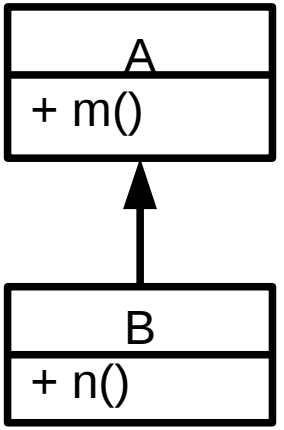
- Il devient aussi possible de redéfinir des méthodes :
  - Changer le comportement d'un concept :
    PileBornee « hérite de » Pile (empiler() vérifie d'abord la taille)

- Impossible de supprimer des attributs ou des méthodes hérités

Principe général -suite-

- Supposons que :
  - J'ai une classe A qui définit une méthode m()
  - J'ai une classe B qui hérite de A et qui définit une méthode n()

- Soit le code suivant :
  $b = new B();



- Sur l'objet $b je peux invoquer la méthode n() de B mais aussi la méthode m() héritée de A :
  $b->n(); $b->m(); // OK

- Par contre, si j'ai le code suivant :
  $a = new A();
  je peux écrire : $a->m()
  mais je ne peux pas écrire : $a->n()

Syntaxe PHP

- Une classe B qui hérite d'une autre classe A :

```
class B extends A {

    // Ici le corps de la classe B

}
```

Exemple avec la syntaxe PHP

- Classe PileBornee :

```
class PileBornee extends Pile {
    public $capacite; private $taille;
    // Autres attributs sont hérités. Pas besoin de les déclarer
    public function __construct($capacite=10) {
        // Initialisation d'une pile normale
        // ...
        // Initialisation spécifique à la pile bornée
        $this->capacite = $capacite;$this->taille=0;
    }
    // ...
    // Toutes les méthodes qui ne sont pas redéfinies ci-dessus
    // sont alors héritées
}
```

Ajout de membres : attributs et méthodes

- Dans une sous-classe, nous pouvons définir de nouveaux attributs et de nouvelles méthodes

- Ceux-ci permettent de raffiner la définition de cette sous-classe par rapport à ce qu'elle hérite de sa super-classe

- Exemple :

```
class Point3D extends Point {
    private $z;
    // Les attributs $x et $y sont hérités
    public function getZ() {return $this->z;}
    public function setZ($z) {$this->z = $z;}
    // Constructeur et autres méthodes ...
}
```

Ajout de membres : attributs et méthodes -suite-

- Avec la nouvelle classe Point3D, il devient possible d'écrire :

```
$p = new Point3D(20,10,5); // On suppose que ce constructeur existe
print($p->getZ()); // Affiche 5
$p->setZ(10);
print($p->getZ()); // Affiche 10
// On peut aussi écrire :
print($p->getX()) ; // Affiche 20
```

Spécialisation de membres : redéfinition de méthodes

- Parfois, il est nécessaire de **redéfinir** les méthodes héritées
- Celles-ci remplacent alors les méthodes définies dans la super-classe lorsqu'elles sont invoquées sur une instance de la sous-classe
- Exemple :

```
class Point3D extends Point {
    // ...
    public function distance(Point3D $autre) {
       // ... retourner la distance en prenant en compte $z
    }
}
$p1 = new Point3D(10,5,20); $p2 = new Point3D(20,10,10);
$p1->distance($p2); // C'est distance de Point3D et pas celle de Point
```

Accès aux membres des classes mères (super-classes)

- Pour accéder à un membre déclaré dans une super-classe, il suffit en PHP d'utiliser le mot clé « *parent* » (l'accès aux membres d'instance est possible avec *$this* aussi, mais *parent* est préféré)

- Ce mot clé s'utilise de la façon suivante : parent::membre

- Exemple : On est dans la sous-classe Point3D

```
public function __toString() {
    return "(".parent::getX().",".parent::getY().",".$this->getZ().")";
}
```

- Les membres privés dans une super-classe ne sont pas accessibles dans une sous-classe : parent::getX() au lieu de parent::x

Accès au constructeur des classes mères

- Il est recommandé d'appeler le constructeur de la super-classe dans le constructeur d'une sous-classe avant toute autre instruction (idem pour les méthodes)

- Objectif : faire les initialisations nécessaires des attributs hérités

- Si vous n'y voyez pas d'intérêt, mauvaise conception par objets !!!

- La syntaxe est la même que pour l'accès aux autres membres :

```
public function __construct($x = 0, $y = 0, $z = 0) {
    parent::__construct($x, $y); // Initialisation des attributs hérités
    $this->z = $z; // Initialisation de l'attribut spécifique à Point3D
}
```

parent est une référence statique

- *parent* est une référence évaluée statiquement. Elle désigne toujours la super-classe de la classe où *parent* est écrite (et non la super-classe de la classe de l'objet dans lequel elle s'exécute)
- Exemple :

```
class A {
    public function m() {print("Je suis un A. ");}
}
class B extends A {
    public function m() {parent::m(); print("Je suis un B");}
}
class C extends B {    }
$c = new C(); $c->m(); // Affiche « Je suis un A. » « Je suis un B »
```

- L'exécution de *parent* ne produit pas une récursivité (*parent* = A)

$this est une référence dynamique

- $this est une référence évaluée dynamiquement
- Exemple :

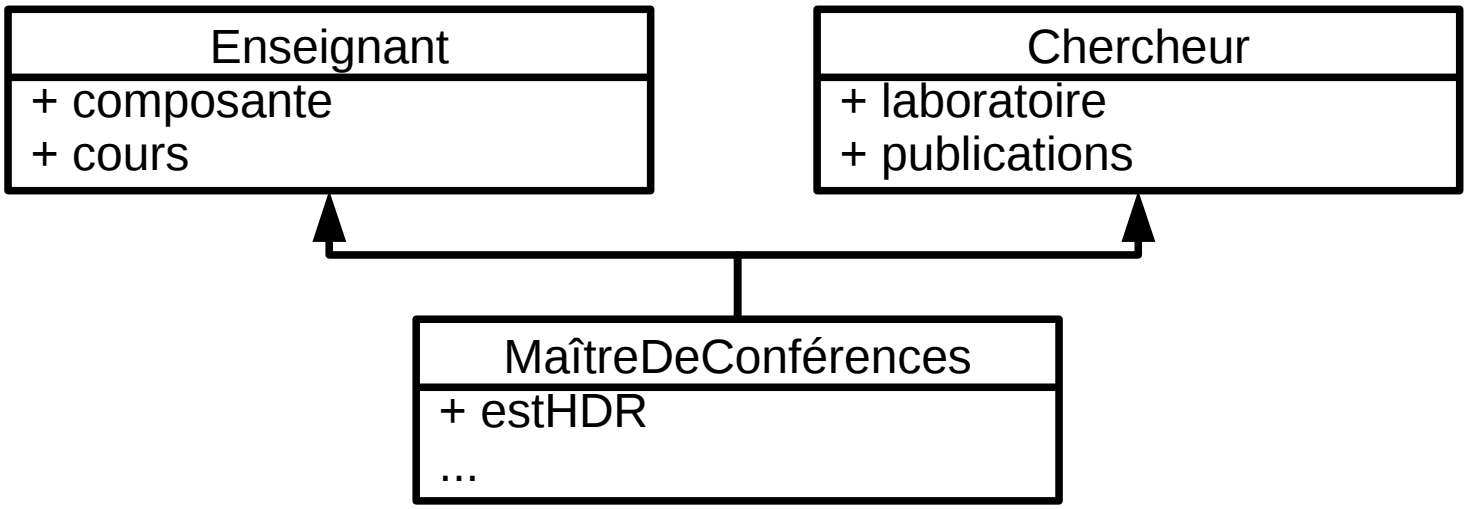
```
class A {
    public function m() {$this->afficher();}
    public function afficher() {print("Je suis un A");}
}
class B extends A {
    public function afficher() {print("Je suis un B");}
}
$obj = new A(); $obj->m(); // Affiche « Je suis un A »
$obj = new B(); $obj->m(); // Affiche « Je suis un B »
```

- Dans la dernière ligne, lorsque la méthode m() héritée de A s'exécute, $this désigne l'objet courant (instance de B) et donc c'est la méthode afficher() de B (non de A) qui est invoquée

Héritage simple versus héritage multiple

- Dans certains langages (pas PHP), il est possible de mettre en place une relation d'héritage entre une classe et plusieurs autres classes (héritage multiple)

- Exemple : un maître de conférences est un enseignant et un chercheur



Héritage simple versus héritage multiple -suite-

- Avantage de l'héritage multiple : plus d'expressivité
  - Aucune restriction liée à l'usage de l'héritage

- Inconvénient : conflits de méthodes -> Si une classe A hérite de deux classes B et C et si A utilise une méthode m() implémentée dans B et dans C, laquelle des deux méthodes s'exécutera pour une instance de A ?!!!
  - Des solutions à ce problème existent, mais la plupart des langages évitent l'héritage multiple à cause de sa complexité

Plan du cours

- Héritage : mécanisme de réutilisation de code
- Héritage : mécanisme d'abstraction
- Principes de l'héritage
- Restriction « *final* » et visibilité « *protected* »

Restriction « *final* »

- Une méthode peut être déclarée « *final* », et dans ce cas celle-ci ne peut pas être redéfinie dans une sous-classe
- Une méthode déclarée « final » est une méthode jugée comme étant une méthode qui implémente un comportement efficace et sûr (à ne pas étendre)
- Une classe peut être déclarée « *final* », et dans ce cas, celle-ci ne peut pas être étendue (héritée). Toutes ses méthodes sont donc implicitement considérées comme « final »

Exemple avec la restriction « *final* »

- Exemple d'une classe avec une méthode déclarée « final » :

```
class A {
    public final function m() {
       print("Je suis un A");
    }
}
```

- L'écriture suivante n'est pas correcte :

```
class B extends A {
    public function m() {
       print("Je suis un B");
    }
}
```

Visibilité « protected »

- Un membre d'une classe déclaré « protected » est accessible depuis la classe dans laquelle il est défini, mais aussi depuis les sous-classes
- C'est donc un niveau de visibilité intermédiaire : entre « private » et « public »
- Grâce à ce mot clé, on peut relâcher l'encapsulation des données et désigner un attribut ou une méthode comme membre pouvant être accédé depuis les sous-classes

Exemple avec la visibilité « protected »

- Classe Pile :
  - Attribut $tab contenant les éléments de la pile déclaré « protected » :

```
class Pile {
    protected $tab;
    // ...
}
```

- Classe PileBornee
  - Pas besoin de stocker la taille de la pile dans un attribut. On y accède depuis l'attribut protected $tab :

```
public function empiler($e) {
   if(count(parent::tab) < $this->capacite)
      //...
}
```

Partie 2 : Classes abstraites et interfaces en PHP

Plan du cours

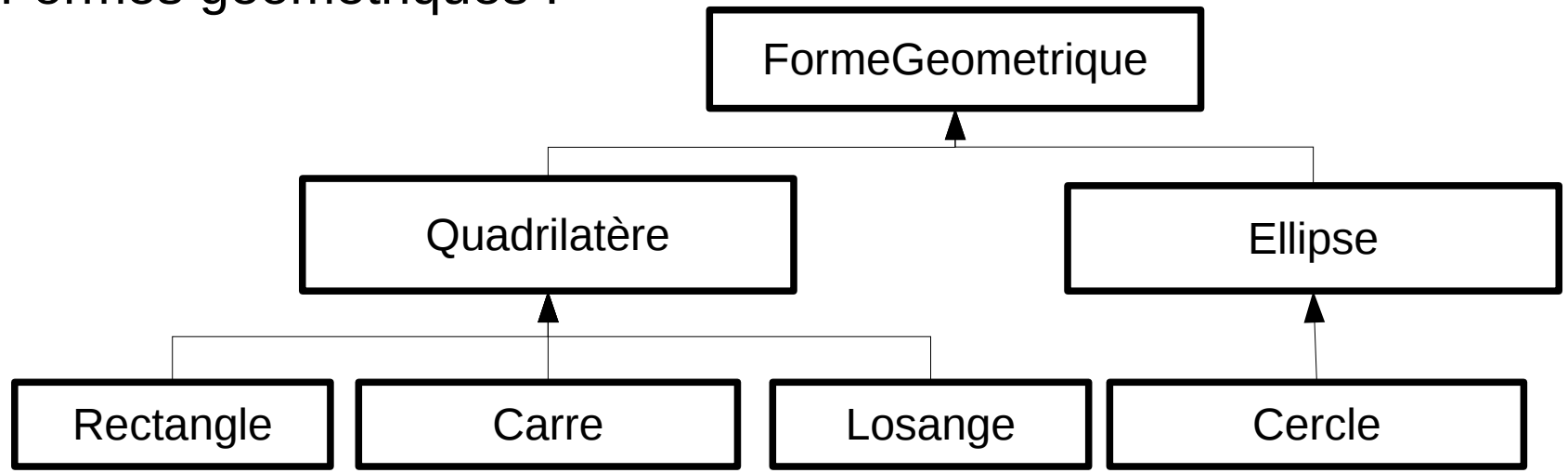
- Retour sur l'héritage : mécanisme d'abstraction
- Méthodes abstraites
- Classes abstraites
- Concept d'interface

Plan du cours

- Retour sur l'héritage : mécanisme d'abstraction
- Méthodes abstraites
- Classes abstraites
- Concept d'interface

Exemple d'illustration

- Formes géométriques :



- Est-ce qu'on sait dessiner une forme géométrique quel qu'elle soit ?
- Est-ce qu'on sait calculer la circonférence d'une forme géométrique quel qu'elle soit ?

  Non, tout dépend de la forme

Concept abstrait vs concept concret

- Il existe donc des concepts dit abstraits et des concepts dits concrets dans le domaine d'une application
- Dans notre exemple, nous retrouvons les deux sortes de concepts
- A quoi peut nous servir un objet créé à partir de la classe FormeGeometrique ?
  A rien
- Il ne peut pas répondre aux invocations de méthodes mentionnées précédemment : calculer circonférence, se dessiner, ...

Concept abstrait vs concept concret -suite-

- FormeGeometrique représente alors un concept abstrait
  - Cette classe est destinée à être héritée
  - Elle nous permet de hiérarchiser les concepts de notre domaine d'application

- Rectangle est en revanche un concept concret
  - Nous pouvons créer des instances de cette classe
  - Nous pouvons ensuite leur demander de calculer leur circonférence, leur aire, ou leur demander de se dessiner

Plan du cours

- Retour sur l'héritage : mécanisme d'abstraction
- Méthodes abstraites
- Classes abstraites
- Concept d'interface

C'est quoi une méthode abstraite ?

- Il s'agit d'une méthode qui n'a pas d'implémentation connue pour une classe donnée
- Elle est représentée par une simple signature
- Dans la signature, on précise le nom de la méthode et la liste de ces paramètres (pas de corps pour la méthode)
- Une méthode concrète est une méthode qui dispose d'une implémentation (méthodes telles qu'on les a vues pour le moment)

A quoi sert une méthode abstraite ?

- Une méthode abstraite sert à introduire la signature d'une fonctionnalité fournie par la classe qui la déclare
- Toute classe qui hérite d'une classe qui déclare une méthode abstraite « doit » fournir une implémentation à celle-ci
- Il s'agit en quelque sorte d'un contrat que doit respecter la sous-classe

Syntaxe de déclaration d'une méthode abstraite

- Pour déclarer une méthode abstraite, il suffit d'ajouter le mot clé « abstract » avant le nom de la méthode

- Syntaxe :
  public abstract function maMethode(...);

- Comme dit précédemment, une méthode abstraite n'a pas d'implémentation : pas de {, suivi d'une séquence d'instructions, terminée par }

Exemple d'une méthode abstraite

- La méthode abstraite circonference() dans la classe FormeGeometrique :
  public abstract circonference();

- Toute forme géométrique doit pouvoir calculer sa circonférence, mais l'implémentation de ce calcul n'est pas possible pour une forme géométrique en général → méthode abstraite

- Toute sous-classe de FormeGeometrique doit :
  - Soit implémenter cette méthode
  - Soit hériter cette méthode (et donc cette sous-classe doit être déclarée abstraite)

Plan du cours

- Retour sur l'héritage : mécanisme d'abstraction
- Méthodes abstraites
- Classes abstraites
- Concept d'interface

C'est quoi une classe abstraite ?

- Une classe qui déclare au moins une méthode abstraite doit être déclarée abstraite
- Une classe abstraite est destinée à être héritée par une autre classe (abstraite ou concrète)
- Une classe abstraite peut contenir des méthodes concrètes
- Elle peut même ne pas contenir des méthode abstraites, mais elle représente dans ce cas un concept abstrait (qu'on interdit d'instancier)
- Une classe concrète est une classe qui n'a que des méthodes concrètes : elle peut être instanciée

Syntaxe de déclaration d'une classe abstraite

- Une classe abstraite est déclarée de la façon suivante :

```
abstract class MaClasse {
    // Attributs, accesseurs et constructeur
    // ...

    public abstract function maMethodeAbstraite();

    // Méthodes abstraites ou concrètes
    // ...
}
```

Exemple d'une classe abstraite

- Une classe abstraite représentant une forme géométrique :

```
abstract class FormeGeometrique {
        private $couleur;
        public function __construct($c) {$this->couleur = $c;}
        public function getCouleur() {return $this->couleur;}
        public function setCouleur($c) {$this->couleur = $c;}
        public abstract function circonference();
        public abstract function aire();
        public abstract function dessiner($zoneDessin);
        public abstract function effacer($zoneDessin);
        public abstract function deplacer($zoneDessin,$dx,$dy);
}
```

Hériter d'une classe abstraite

- Une classe abstraite peut être héritée par d'autres classes
- Les mêmes principes du mécanisme d'héritage entre classes concrètes s'appliquent aux classes abstraites
- Seule contrainte : une classe qui hérite d'une classe abstraite doit
  - Fournir une implémentation à toutes les méthodes abstraites de sa super-classe (pour être une classe concrète)
  - Être déclarée abstraite (elle aussi), sinon (si au moins une des méthodes abstraites héritées n'est pas implémentée)

Exemple d'héritage d'une classe abstraite

- Une classe représentant un quadrilatère :

```
abstract class Quadrilatere extends FormeGeometrique {
    private $sommets; // Un tableau qui contient les 4 sommets
    public function __construct($couleur,$sommets) {
        parent::__construct($couleur);
        if(count($sommets) < 4) return; // ERREUR
        $this->sommets = $sommets;
    }
    // Accesseurs ...
    public function circonference() {
     // On sait l'implémenter ici (somme des distances entre les sommets)
    }
    // La fonction aire() est abstraite (implémentation différente
    selon le quadrilatère)}
```

Exemple d'héritage d'une classe abstraite -suite-

- Une classe représentant un carré :

```
class Carre extends Quadrilatere {
        public function __construct($couleur,$sommets) {
              parent::__construct($couleur,$sommets);
              // Vérifier l'égalité des distances entre les sommets
              // et le fait que les angles soient rectangles, sinon ERREUR
        }
        public function aire() {
          // Calculer et retourner l'aire du carré (longueur côté au carré)
        }
        // Autres méthodes concrètes
        // ...
}
```

Exemple d'héritage d'une classe abstraite -suite-

- Une classe représentant un rectangle :

```
class Rectangle extends Quadrilatere {
        public function __construct($couleur,$sommets) {
              parent::__construct($couleur,$sommets);
              // Vérifier l'égalité des côtés opposés
              // et le fait que les angles soient rectangles, sinon ERREUR
        }
        public function aire() {
         // Calculer et retourner l'aire du rectangle (longueur x largeur)
        }
         // Autres méthodes concrètes
         // ...
}
```

Plan du cours

- Retour sur l'héritage : mécanisme d'abstraction
- Méthodes abstraites
- Classes abstraites
- Concept d'interface

Spécification de méthodes abstraites

- Un regroupement de méthodes abstraites seules est appelé une interface
- Il s'agit donc d'une structure encore plus abstraite qu'une classe abstraite
- Dans une classe, on n'hérite pas d'une interface, on l'**implémente**
- Pourquoi le nom « interface » ?
  - ensemble de signatures de méthodes publiques que les classes vont implémenter. Les objets créés à partir de ces classes vont alors exposer ces méthodes, qui deviennent l'interface de l'objet

Déclarer une interface

- Une interface est déclarée de la façon suivante :

```
interface MonInterface {
    public function ma1ereMethode();
    public function ma2ndeMethode($a,$b);
    // ...
}
```

- Pas besoin de préciser que les méthodes sont abstraites (ne pas utiliser le mot « abstract »). C'est implicite

- Une interface ne peut pas être instanciée. Elle est destinée à être implémentée

Exemple d'une interface

- Une interface de comptes bancaires :

```
interface Compte {
        public function crediter($montant);
        public function debiter($montant);
        public function transferer($autre_compte,$montant);
}
```

Implémenter une interface

- Une classe peut déclarer qu'elle implémente une ou plusieurs interfaces

- Si une classe implémente une interface, elle doit fournir le corps (l'implémentation) de chaque méthode dans l'interface
  - Sinon, la classe doit être déclarée abstraite

- Exemple :

```
class CompteCourant implements Compte {
        // Fournir une implémentation des 3 méthodes
}
class CompteEpargne implements Compte {
        // Fournir une implémentation différente des mêmes méthodes
}
```

Héritage entre interfaces

- Une interface peut hériter d'une autre interface (sa super-interface)

- Dans ce cas, la classe qui implémente la sous-interface doit fournir une implémentation de toutes les méthodes (de la sous-interface et de sa super-interface)

- Exemple :

```
interface CompteAvecInteret extends Compte {
        public function appliquerInteret();
}
class CompteCourantStandard implements CompteAvecInteret {
        // Implémentation des 3 méthodes de Compte et de la méthode
        // ci-dessus
}
```

Classe abstraite vs. interface

- Une classe abstraite est une classe à part entière : elle peut déclarer des attributs, un constructeur et des méthodes concrètes
- Une interface ne peut déclarer que des méthodes abstraites
- Une classe ne peut hériter que d'une seule classe abstraite, mais peut implémenter plusieurs interfaces

Références bibliographiques

- Livres :
  - sur PHP à partir de la version 5.0
  - sur la programmation orientée (ou par) objets en général, mais souvent application à Java, C++, C# ou Objective C
- Ressources Web :
  - Le site php.net : http://fr2.php.net/manual/fr/language.oop5.php

Questions