Concepts fondamentaux de la programmation par objets

alexandre.mesle@gmail.com

19 juillet 2011

A propos de ce cours

- Synonymes : programmation orientée objet, programmation par objets, programmation objet
- Langages : Java, C++, C#, Objective Caml
- Concepts communs à ces langages

A propos de ce cours

- Synonymes : programmation orientée objet, programmation par objets, programmation objet
- Langages : Java, C++, C#, Objective Caml
- Concepts communs à ces langages

A propos de ce cours

- Synonymes : programmation orientée objet, programmation par objets, programmation objet
- Langages : Java, C++, C#, Objective Caml
- Concepts communs à ces langages

A propos de ce cours

- Synonymes : programmation orientée objet, programmation par objets, programmation objet
- Langages : Java, C++, C#, Objective Caml
- Concepts communs à ces langages

Avantages

- Programmation plus simple et plus rapide
- Abstraction des données
- Modularité/modifiabilité
- Lisibilité
- Réutilisabilité

Avantages

- Programmation plus simple et plus rapide
- Abstraction des données
- Modularité/modifiabilité
- Lisibilité
- Réutilisabilité

Avantages

- Programmation plus simple et plus rapide
- Abstraction des données
- Modularité/modifiabilité
- Lisibilité
- Réutilisabilité

Avantages

- Programmation plus simple et plus rapide
- Abstraction des données
- Modularité/modifiabilité
- Lisibilité
- Réutilisabilité

Avantages

- Programmation plus simple et plus rapide
- Abstraction des données
- Modularité/modifiabilité
- Lisibilité
- Réutilisabilité

Avantages

- Programmation plus simple et plus rapide
- Abstraction des données
- Modularité/modifiabilité
- Lisibilité
- Réutilisabilité

Inconvénients

- Plus difficile maîtriser
- Plus long à l'exécution

Inconvénients

- Plus difficile maîtriser
- Plus long à l'exécution

Inconvénients

- Plus difficile maîtriser
- Plus long à l'exécution

Types

Definition

$Type = (ensemble\ de\ valeurs) \cup (operations\ sur\ ces\ valeurs)$

- types primitifs
- types construits
  - fournis avec le langage
  - crées par le programmeur

Types

Definition

$Type = (ensemble\ de\ valeurs) \cup (operations\ sur\ ces\ valeurs)$

- types primitifs
- types construits
  - fournis avec le langage
  - crées par le programmeur

Types

Definition

$Type = (ensemble\ de\ valeurs) \cup (operations\ sur\ ces\ valeurs)$

- types primitifs
- types construits
  - fournis avec le langage
  - crées par le programmeur

Types

Definition

$Type = (ensemble\ de\ valeurs) \cup (operations\ sur\ ces\ valeurs)$

- types primitifs
- types construits
  - fournis avec le langage
  - crées par le programmeur

Types

Definition

$Type = (ensemble\ de\ valeurs) \cup (operations\ sur\ ces\ valeurs)$

- types primitifs
- types construits
  - fournis avec le langage
  - crées par le programmeur

Classes

Definition

Classe = type construit

- similaire à une structure
- contient des champs (membres, attributs)

Classes

Definition

Classe = type construit

- similaire à une structure
- contient des champs (membres, attributs)

Classes

Definition

Classe = type construit

- similaire à une structure
- contient des champs (membres, attributs)

Classes

Tout point dans $\mathbb{R}^2$ est formé de deux coordonnées

Exemple (La classe Point)

Classe *Point*
  Champs
    flottant abscisse
    flottant ordonnée
finClasse

Classes

Tout point dans $\mathbb{R}^2$ est formé de deux coordonnées

Exemple (La classe Point)

Classe *Point*
    Champs
        flottant abscisse
        flottant ordonnée
finClasse

Objets

Definition

Objet = variable d'un type construit

- une classe est un type
- un objet a un type
- ce type est défini dans une classe

Objets

Definition

Objet = variable d'un type construit

- une classe est un type
- un objet a un type
- ce type est défini dans une classe

Objets

Definition

Objet = variable d'un type construit

- une classe est un type
- un objet a un type
- ce type est défini dans une classe

Objets

Definition

Objet = variable d'un type construit

- une classe est un type
- un objet a un type
- ce type est défini dans une classe

Objets

Exemple (l'objet x de type Point)

Données *: Point x*

...
Afficher "l'abscisse du point x est ", x.abscisse
Afficher "l'ordonnée du point x est ", x.ordonnée
...

Méthodes

Definition

Méthode = sous-programme propre à chaque objet.

- définies dans les classes
- appelées depuis un objet
- les méthodes ont accès en lecture et écriture aux champs de cet objet

Méthodes

Definition

Méthode = sous-programme propre à chaque objet.

- définies dans les classes
- appelées depuis un objet
- les méthodes ont accès en lecture et écriture aux champs de cet objet

Méthodes

Definition

Méthode = sous-programme propre à chaque objet.

- définies dans les classes
- appelées depuis un objet
- les méthodes ont accès en lecture et écriture aux champs de cet objet

Méthodes

Definition

Méthode = sous-programme propre à chaque objet.

- définies dans les classes
- appelées depuis un objet
- les méthodes ont accès en lecture et écriture aux champs de cet objet

Exemple (La classe Point)

Classe *Point*

Champs

flottant abscisse

flottant ordonnee

Méthodes

Méthode *translation(entier i, entier j)*

$abscisse \longleftarrow ascisse + i$

$ordonnee \longleftarrow ordonnee + j$

finMéthode

Méthode *norme()*

retourner $\sqrt{abscisse^2 + ordonnee^2}$

finMéthode

finClasse

Objets

Exemple (l'objet x de type Point)

Données *: Point x*

...
Afficher "la norme de x est ", x.norme()
x.translation(1, 1)
Afficher "les coordonnées de x sont ", x.ordonnée, x.abscisse
...

Fonctionnement d'une Pile

Une pile permet de faire les opérations suivantes :

- Ajouter un élément
- Consulter le dernier élément ajouté
- Supprimer le dernier élément ajouté

Fonctionnement d'une Pile

Une pile permet de faire les opérations suivantes :

- Ajouter un élément
- Consulter le dernier élément ajouté
- Supprimer le dernier élément ajouté

Fonctionnement d'une Pile

Une pile permet de faire les opérations suivantes :

- Ajouter un élément
- Consulter le dernier élément ajouté
- Supprimer le dernier élément ajouté

Fonctionnement d'une Pile

Une pile permet de faire les opérations suivantes :

- Ajouter un élément
- Consulter le dernier élément ajouté
- Supprimer le dernier élément ajouté

Une implémentation d'une Pile

Exemple (La classe Pile (champs))

Classe *Pile*
    Champs
        Entier nbe
        Tableau d'Entiers t
finClasse

Exemple (La classe Pile (méthodes))

Méthodes

Méthode *ajouter(Entier e)*
$t[nbe + 1] \longleftarrow e$
$nbe \longleftarrow nbe + 1$
finMéthode
Méthode *sommet()*
retourner $t[nbe]$
finMéthode
Méthode *SupprimerSommet()*
$nbe \longleftarrow nbe - 1$
finMéthode
Méthode *estVide()*
retourner $nbe = 0$
finMéthode

Déclaration et Instanciation

Trois zones dans la mémoire

- Le Code
- La Pile (Stack)
- Le Tas (Heap)

Déclaration et Instanciation

Trois zones dans la mémoire

- Le Code
- La Pile (Stack)
- Le Tas (Heap)

Déclaration et Instanciation

Trois zones dans la mémoire

- Le Code
- La Pile (Stack)
- Le Tas (Heap)

Déclaration et Instanciation

Trois zones dans la mémoire

- Le Code
- La Pile (Stack)
- Le Tas (Heap)

Déclaration et Instanciation

Trois zones dans la mémoire

- Le Code
- La Pile (Stack)
- Le Tas (Heap)

Déclaration et Instanciation

En C,

```
// Déclaration
int* tab;
// Allocation
tab = (int*)malloc(10*sizeof(int));
```

- Déclaration dans la Pile (connu à la compilation)
- Allocation dans le Tas (connu à l'exécution)

Déclaration et Instanciation

En C,

```
// Déclaration
int* tab;
// Allocation
tab = (int*)malloc(10*sizeof(int));
```

- Déclaration dans la Pile (connu à la compilation)
- Allocation dans le Tas (connu à l'exécution)

Déclaration et Instanciation

En C,

```
// Déclaration
int* tab;
// Allocation
tab = (int*)malloc(10*sizeof(int));
```

- Déclaration dans la Pile (connu à la compilation)
- Allocation dans le Tas (connu à l'exécution)

Déclaration et Instanciation

En C,

```
// Déclaration
int* tab;
// Allocation
tab = (int*)malloc(10*sizeof(int));
```

- Déclaration dans la Pile (connu à la compilation)
- Allocation dans le Tas (connu à l'exécution)

Déclaration et Instanciation

En Java,

```
// Déclaration
Point p;
// Allocation
p = new Point();
```

- *p* contient l'identifiant d'un nouvel objet de type *Point*
- *p* référence un nouvel objet de type *Point*

Déclaration et Instanciation

En Java,

```
// Déclaration
Point p;
// Allocation
p = new Point();
```

- *p* contient l'identifiant d'un nouvel objet de type *Point*
- *p* référence un nouvel objet de type *Point*

Déclaration et Instanciation

En Java,

```
// Déclaration
Point p;
// Allocation
p = new Point();
```

- *p* contient l'identifiant d'un nouvel objet de type *Point*
- *p* référence un nouvel objet de type *Point*

Pointeurs et Références

En Java,

```
// Déclarations
Point p;
Point q;
// Allocation, p référence un nouveau Point
p = new Point();
// Affectation, q référence le même Point que p
q = p;
// Modification
p.abscisse = 2;
p.ordonnée = 3;
```

Quelles sont les abscisses et ordonnées du *Point* référencé par *q* ?

Pointeurs et Références

- Les objets ne sont pas contenus par des variables,
- Ils sont référencés (ou pointés) par des variables.

Pointeurs et Références

- Les objets ne sont pas contenus par des variables,
- Ils sont référencés (ou pointés) par des variables.

Spécification d'une Pile

Ce qui intéresse l'utilisateur de la Pile, c'est :

Exemple (La classe Pile (spécification))

Méthodes

ajouter(Entier e)
sommet() retourne un Entier
SupprimerSommet()
estVide() retourne un booléen

Indépendant de l'implémentation.

Spécification d'une Pile

Ce qui intéresse l'utilisateur de la Pile, c'est :

Exemple (La classe Pile (spécification))

Méthodes

ajouter(Entier e)
sommet() retourne un Entier
SupprimerSommet()
estVide() retourne un booléen

Indépendant de l'implémentation.

Visibilité

Definition

La portée (ou visibilité d'une variable) est l'ensemble des blocs de code dans lequels on peut l'utiliser.

- Privé : seulement visible par les méthodes
- Public : visible n'importe ou

Visibilité

Definition

La portée (ou visibilité d'une variable) est l'ensemble des blocs de code dans lequels on peut l'utiliser.

- Privé : seulement visible par les méthodes
- Public : visible n'importe ou

Visibilité

Definition

La portée (ou visibilité d'une variable) est l'ensemble des blocs de code dans lequels on peut l'utiliser.

- Privé : seulement visible par les méthodes
- Public : visible n'importe ou

Remarques

Privé implique

- impossible d'utiliser la notation pointée.
- nécessaire de passer par les méthodes.
- sécurité, bidouillage impossible.
- masquer la complexité de l'implémentation.

Remarques

Privé implique

- impossible d'utiliser la notation pointée.
- nécessaire de passer par les méthodes.
- sécurité, bidouillage impossible.
- masquer la complexité de l'implémentation.

Remarques

Privé implique

- impossible d'utiliser la notation pointée.
- nécessaire de passer par les méthodes.
- sécurité, bidouillage impossible.
- masquer la complexité de l'implémentation.

Remarques

Privé implique

- impossible d'utiliser la notation pointée.
- nécessaire de passer par les méthodes.
- sécurité, bidouillage impossible.
- masquer la complexité de l'implémentation.

Remarques

Privé implique

- impossible d'utiliser la notation pointée.
- nécessaire de passer par les méthodes.
- sécurité, bidouillage impossible.
- masquer la complexité de l'implémentation.

Un petit problème

- Comment on initialise des variables privées ?
- En utilisant une méthode appelée lors de la création de l'objet
- Cette méthode s'appelle un constructeur
- Le constructeur est appelé automatiquement au moment de l'instanciation.

Un petit problème

- Comment on initialise des variables privées ?
- En utilisant une méthode appelée lors de la création de l'objet
- Cette méthode s'appelle un constructeur
- Le constructeur est appelé automatiquement au moment de l'instanciation.

Un petit problème

- Comment on initialise des variables privées ?
- En utilisant une méthode appelée lors de la création de l'objet
- Cette méthode s'appelle un constructeur
- Le constructeur est appelé automatiquement au moment de l'instanciation.

Un petit problème

- Comment on initialise des variables privées ?
- En utilisant une méthode appelée lors de la création de l'objet
- Cette méthode s'appelle un constructeur
- Le constructeur est appelé automatiquement au moment de l'instanciation.

Exemple (La classe Point)

Classe *Point*

Champs

flottant privé abscisse

flottant privé ordonnee

Méthodes

Méthode *constructeur(flottant x, flottant y)*

abscisse $\longleftarrow$ *x*

ordonnée $\longleftarrow$ *y*

finMéthode

...

finClasse

Comment accéder aux coordonnées des points ?

Exemple (La classe Point)

Classe *Point*

Champs

flottant privé abscisse

flottant privé ordonnee

Méthodes

Méthode *constructeur(flottant x, flottant y)*

abscisse $\longleftarrow$ *x*

ordonnée $\longleftarrow$ *y*

finMéthode

...

finClasse

Comment accéder aux coordonnées des points ?

Exemple (La classe Point (méthodes))

Méthode *getX()*
    retourne abscisse
finMéthode
Méthode *getY()*
    retourne ordonnée
finMéthode
Méthode *setX(flottant x)*
    abscisse $\longleftarrow$ *x*
finMéthode
Méthode *setY(flottant y)*
    ordonnée $\longleftarrow$ *y*
finMéthode
...

Ces fonctions s'appellent des accesseurs.

Exemple (La classe Point (méthodes))

Méthode *getX()*
    retourne abscisse
finMéthode
Méthode *getY()*
    retourne ordonnée
finMéthode
Méthode *setX(flottant x)*
    abscisse $\longleftarrow$ *x*
finMéthode
Méthode *setY(flottant y)*
    ordonnée $\longleftarrow$ *y*
finMéthode
...

Ces fonctions s'appellent des accesseurs.

Javadoc

En java, documentation automatique.

```
/**
Ajoute un objet <CODE>item</CODE>
impl&eacute;mentant <CODE>Comparable</CODE>
dans le tas.
*/

public void add(Comparable item)
{
addLeaf(item);
balance(getSize()-1);
}
```

Javadoc

En java, documentation automatique.

```
/**
Ajoute un objet <CODE>item</CODE>
impl&eacute;mentant <CODE>Comparable</CODE>
dans le tas.
*/

public void add(Comparable item)
{
addLeaf(item);
balance(getSize()-1);
}
```

Exemple

Programmation de l'algorithme de Graham : besoin du sous-sommet de Pile.

Exemple

Méthodes

ajouter(Entier e)
sommet() retourne un Entier
sousSommet() retourne un Entier
SupprimerSommet()
estVide() retourne un booléen

Exemple

Programmation de l'algorithme de Graham : besoin du sous-sommet de Pile.

Exemple

Méthodes

ajouter(Entier e)
sommet() retourne un Entier
sousSommet() retourne un Entier
SupprimerSommet()
estVide() retourne un booléen

PileGraham hérite de *Pile* = récupère tous les champs et les méthodes de *Pile*.

Exemple

Classe *PileGraham* **hérite de** *Pile*
  Méthodes
    Méthode *sousSommet()*
      retourner $tab[nbe - 1]$
    finMéthode
finClasse

- *Pile* est la classe mère
- *PileGraham* est la classe fille

PileGraham hérite de *Pile* = récupère tous les champs et les méthodes de *Pile*.

Exemple

Classe *PileGraham* **hérite de** *Pile*
    Méthodes
        Méthode *sousSommet()*
            retourner $tab[nbe - 1]$
        finMéthode
finClasse

- *Pile* est la classe mère
- *PileGraham* est la classe fille

PileGraham hérite de *Pile* = récupère tous les champs et les méthodes de *Pile*.

Exemple

Classe *PileGraham* **hérite de** *Pile*
    Méthodes
        Méthode *sousSommet()*
            retourner $tab[nbe - 1]$
        finMéthode
finClasse

- *Pile* est la classe mère
- *PileGraham* est la classe fille

PileGraham hérite de *Pile* = récupère tous les champs et les méthodes de *Pile*.

Exemple

Classe *PileGraham* **hérite de** *Pile*
    Méthodes
        Méthode *sousSommet()*
            retourner $tab[nbe - 1]$
        finMéthode
finClasse

- *Pile* est la classe mère
- *PileGraham* est la classe fille

Redéfinition de Méthodes

- Contexte : circuit imprimés.
- On change de norme : $|x| + |y|$
- Comment réutiliser la classe *Point* avec une autre norme.

Redéfinition de Méthodes

- Contexte : circuit imprimés.
- On change de norme : $|x| + |y|$
- Comment réutiliser la classe *Point* avec une autre norme.

Redéfinition de Méthodes

- Contexte : circuit imprimés.
- On change de norme : $|x| + |y|$
- Comment réutiliser la classe *Point* avec une autre norme.

Redéfinition de Méthodes

- Contexte : circuit imprimés.
- On change de norme : $|x| + |y|$
- Comment réutiliser la classe *Point* avec une autre norme.

Redéfinition de Méthodes

Exemple

Classe *PointRectilinéaire* **hérite de** *Point*
  Méthodes
    Méthode *norme()*
      retourner $|abscisse| + |ordonne|$
    finMéthode
finClasse

La méthode de la classe fille masque celle de la classe mère.

Redéfinition de Méthodes

Exemple

Classe *PointRectilinéaire* **hérite de** *Point*
    Méthodes
        Méthode *norme()*
            retourner $|abscisse| + |ordonne|$
        finMéthode
finClasse

La méthode de la classe fille masque celle de la classe mère.

Définition

- Sommets : classes
- Arcs : (i, j) si i est une classe fille de j.
- Sans circuit
- Définit une relation transitive

Définition

- Sommets : classes
- Arcs : (i, j) si i est une classe fille de j.
- Sans circuit
- Définit une relation transitive

Définition

- Sommets : classes
- Arcs : (i, j) si i est une classe fille de j.
- Sans circuit
- Définit une relation transitive

Définition

- Sommets : classes
- Arcs : (i, j) si i est une classe fille de j.
- Sans circuit
- Définit une relation transitive

Exemple

- Quadrilatère
- Trapèze
- Parallèlogramme
- Rectangle
- Losange

Un algorithme de recherche d'une méthode

fonction trouveMéthode(*nomM*, *c*)
si *nomM est une méthode de la classe c* **alors**
    m ⟵ la méthode *nomM* de la classe *c*
    retourner *m*
sinon
    pour *toute classe c′ mère de nomClasse* **faire**
        e ⟵ trouveMéthode(*nomM*, *c′*)
        si $e \neq NULL$ **alors**
            retourner *e*
        fin
    fin
    retourner *NULL*
fin

Java Vs C++

- Problème : conflit de noms de méthodes dans le graphe d'héritage.
- Solutions :
  - Utiliser des règles de priorité d'exploration des classes mères
  - Ne rien faire
  - Interdire l'héritage multiple (Java)

Java Vs C++

- Problème : conflit de noms de méthodes dans le graphe d'héritage.
- Solutions :
  - Utiliser des règles de priorité d'exploration des classes mères
  - Ne rien faire
  - Interdire l'héritage multiple (Java)

Java Vs C++

- Problème : conflit de noms de méthodes dans le graphe d'héritage.
- Solutions :
  - Utiliser des règles de priorité d'exploration des classes mères
  - Ne rien faire
  - Interdire l'héritage multiple (Java)

Java Vs C++

- Problème : conflit de noms de méthodes dans le graphe d'héritage.
- Solutions :
  - Utiliser des règles de priorité d'exploration des classes mères
  - Ne rien faire
  - Interdire l'héritage multiple (Java)

Java Vs C++

- Problème : conflit de noms de méthodes dans le graphe d'héritage.
- Solutions :
  - Utiliser des règles de priorité d'exploration des classes mères
  - Ne rien faire
  - Interdire l'héritage multiple (Java)

Deux objets très spéciaux

- *Objet* : tout objet hérite d'*Objet*
- *Classe* : toute classe est une instance de *Classe*

Deux objets très spéciaux

- *Objet* : tout objet hérite d'*Objet*
- *Classe* : toute classe est une instance de *Classe*

Deux objets très spéciaux

- *Objet* : tout objet hérite d'*Objet*
- *Classe* : toute classe est une instance de *Classe*

Exemple

Exemple (l'objet x de type Point)

Données : *Pile p*
PileGraham pg
$pg \longleftarrow nouvelleInstancedePileGraham$
$p \longleftarrow pg$
p.empile(2)
Afficher *p.sommet*()

- Une variable de type *Pile* référence un objet de type *PileGraham*
- Est-ce que cela pose problème à quelqu'un ?

Exemple

Exemple (l'objet x de type Point)

Données : *Pile p*
PileGraham pg
$pg \longleftarrow nouvelleInstancedePileGraham$
$p \longleftarrow pg$
p.empile(2)
Afficher *p.sommet*()

- Une variable de type *Pile* référence un objet de type *PileGraham*
- Est-ce que cela pose problème à quelqu'un ?

Exemple

Exemple (l'objet x de type Point)

Données : *Pile p*
PileGraham pg
$pg \longleftarrow nouvelleInstancedePileGraham$
$p \longleftarrow pg$
p.empile(2)
Afficher *p.sommet*()

- Une variable de type *Pile* référence un objet de type *PileGraham*
- Est-ce que cela pose problème à quelqu'un ?

Expression arithmétique

Une expression arithmétique est soit

- une valeur numérique
- un opérateur binaire et deux sous-expressions arithmétiques.
- dessinons l'arbre d'héritage
- où placer les fonctions d'évaluation ?

Expression arithmétique

Une expression arithmétique est soit

- une valeur numérique
- un opérateur binaire et deux sous-expressions arithmétiques.
- dessinons l'arbre d'héritage
- où placer les fonctions d'évaluation ?

Expression arithmétique

Une expression arithmétique est soit

- une valeur numérique
- un opérateur binaire et deux sous-expressions arithmétiques.
- dessinons l'arbre d'héritage
- où placer les fonctions d'évaluation ?

Expression arithmétique

Une expression arithmétique est soit

- une valeur numérique
- un opérateur binaire et deux sous-expressions arithmétiques.
- dessinons l'arbre d'héritage
- où placer les fonctions d'évaluation ?