

PROGRAMMATION ORIENTEE OBJET

Par ZOUBIR Hamza

Table des matières

1. L'objet.....	4
2. Objet et classe.....	5
3. Les trois fondamentaux de la POO.....	5
3.1. Encapsulation.....	5
3.2. Héritage.....	5
3.3. Polymorphisme.....	7
II- Différents types de méthodes.....	8
1. Constructeurs et destructeurs.....	8
1.1. Constructeurs.....	8
1.2. Destructeurs.....	8
2. Pointeur interne.....	8
3. Visibilité.....	8
3.1. Champs et méthodes publics.....	9
3.2. Champs et méthodes privés.....	9
3.3. Champs et méthodes protégés.....	9
4. Surcharge et redéfinition des méthodes d'une classe.....	9
III- Adapter pattern.....	10
1. But.....	10
2. Structure.....	10
3. Participants.....	10
4. Utilisations connues.....	10
IV- Exemples.....	11
1. Class Point.....	11
2. Class Segment.....	13
3. Class pile.....	14
4. class Rationel.....	15
5. Classe matrice :.....	19
6. Classe forme :.....	20

7. Classearticle :	21
8. Gestion des Compitions	22
Documentation des classes	22
Référence de la classe Date	22
Fonctions membres publiques	22
Référence de la classe Entraîneur	22
Fonctions membres publiques	22
Référence de la classe Equipe	23
Fonctions membres publiques	23
Référence de la classe Joueur	23
Fonctions membres publiques	23
Référence de la classe Matche	23
Fonctions membres publiques	23
Fonctions membres publiques	24
Documentation des fichiers	24
Référence du fichier Date.cpp	24
Référence du fichier Date.h	24
Classes	24
Référence du fichier Entraîneur.cpp	24
Référence du fichier Entraîneur.h	24
Classes	24
Classes	25
Référence du fichier Gestioncompetition.cpp	25
Macros	25
Fonctions	25

I- Object-Oriented Programming

Ces termes sont devenus très communs, dit-il. Malheureusement peu de gens sont d'accord sur leur sens". On met souvent en parallèle les notions de "type abstrait" et d'objet (une classe définit un type pour toutes ses instances), à tel point que beaucoup confondent volontairement type et classe. Des langages comme ADA, Clu, C++ donnent la possibilité de définir des types qui se manipulent plus ou moins de la même façon que des types de base. Cela donne lieu à ce que l'on appelle souvent "types abstraits de données".

1. L'objet

Il est impossible de parler de Programmation Orientée Objet sans parler d'objet, bien entendu. Tâchons donc de donner une définition aussi complète que possible d'un objet.

Un objet est avant tout une structure de données. Autrement, il s'agit d'une entité chargée de gérer des données, de les classer, et de les stocker sous une certaine forme. En cela, rien ne distingue un objet d'une quelconque autre structure de données. La principale différence vient du fait que l'objet regroupe les données et les moyens de traitement de ces données.

Un objet rassemble de fait deux éléments de la programmation procédurale.

Les **champs** :

Les champs sont à l'objet ce que les variables sont à un programme : ce sont eux qui ont en charge les données à gérer. Tout comme n'importe quelle autre variable, un champ peut posséder un type quelconque défini au préalable : nombre, caractère... ou même un type objet.

Les **méthodes** :

Les méthodes sont les éléments d'un objet qui servent d'interface entre les données et le programme. Sous ce nom obscur se cachent simplement des procédures ou fonctions destinées à traiter les données.

Les champs et les méthodes d'un objet sont ses membres.

Si nous résumons, un objet est donc un type servant à stocker des données dans des champs et à les gérer au travers des méthodes.

Si on se rapproche du Pascal, un objet n'est donc qu'une extension évoluée des enregistrements (type record) disposant de procédures et fonctions pour gérer les champs qu'il contient.

2. Objet et classe

Avec la notion d'objet, il convient d'amener la notion de classe. Cette notion de classe n'est apparue dans le langage Pascal qu'avec l'avènement du langage Delphi et de sa nouvelle approche de la Programmation Orientée Objet. Elle est totalement absente du Pascal standard.

Ce que l'on a pu nommer jusqu'à présent objet est, pour Delphi, une classe d'objet. Il s'agit donc du type à proprement parler. L'objet en lui-même est une instance de classe, plus simplement un exemplaire d'une classe, sa représentation en mémoire.

3. Les trois fondamentaux de la POO

La Programmation Orientée Objet est dirigée par trois fondamentaux qu'il convient de toujours garder à l'esprit : encapsulation, héritage et polymorphisme. Houlà ! Inutile de fuir en voyant cela, car en fait, ils ne cachent que des choses relativement simples. Nous allons tenter de les expliquer tout de suite.

3.1. Encapsulation

L'encapsulation introduit donc une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis un ensemble de procédures et fonctions destinées à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'une seule et même entité.

Si l'encapsulation est déjà une réalité dans les langages procéduraux (comme le Pascal non objet par exemple) au travers des unités et autres librairies, il prend une toute nouvelle dimension avec l'objet.

Pour conclure, l'encapsulation permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.

3.2. Héritage

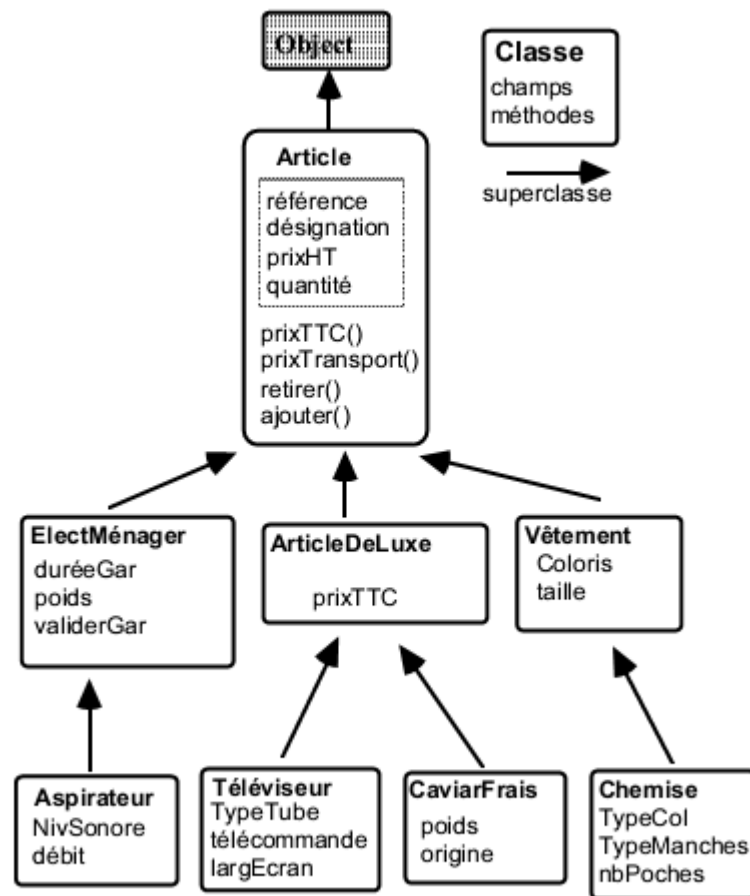
Spécialiser et factoriser : important pour la "réutilisation".

- Les connaissances les plus générales sont mises en commun dans des classes qui sont ensuite spécialisées par définitions de sous-classes (connaissances spécifiques).
- Une sous-classe est donc la spécialisation de la description d'une classe appelée sa superclasse.

- Une sous-classe hérite de sa superclasse.
- Conceptuellement tout se passe comme si les informations de la superclasse étaient recopiées dans la sous-classe.
- La spécialisation d'une classe peut être réalisée selon deux techniques :
 - l'enrichissement: la sous-classe est dotée de nouvelles variables d'instances et/ou de méthodes.
 - substitution: donner une nouvelle définition à une méthode héritée, lorsque celle-ci se révèle inadéquate pour l'ensemble des objets de la sous-classe (masquage).

Le graphe d'héritage

- La relation d'héritage lie une classe à sa superclasse.
- Lorsqu'une classe n'a qu'une seule superclasse c'est l'héritage simple, le graphe d'héritage constitue un arbre.
- La structuration en classes et sous-classes entraîne une modularité importante.
- Les modifications dans une classe n'ont d'incidence que dans le sous-arbre de la classe considérée.



Un graphe d'héritage simple

3.3. Polymorphisme

Le terme polymorphisme est certainement celui que l'on appréhende le plus. Mais il ne faut pas s'arrêter à cela. Afin de mieux le cerner, il suffit d'analyser la structure du mot : poly comme plusieurs et morphisme comme forme. Le polymorphisme traite de la capacité de l'objet à posséder plusieurs formes.

Cette capacité dérive directement du principe d'héritage vu précédemment. En effet, comme on le sait déjà, un objet va hériter des champs et méthodes de ses ancêtres. Mais un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire, ou de la compléter.

On voit donc apparaître ici ce concept de polymorphisme : choisir en fonction des besoins quelle méthode ancêtre appeler, et ce au cours même de l'exécution. Le comportement de l'objet devient donc modifiable à volonté.

Le polymorphisme, en d'autres termes, est donc la capacité du système à choisir dynamiquement la méthode qui correspond au type réel de l'objet en cours. Ainsi, si l'on considère un objet Véhicule et ses descendants Bateau, Avion, Voiture possédant tous une méthode Avancer, le système appellera la fonction Avancer spécifique suivant que le véhicule est un Bateau, un Avion ou bien une Voiture.

II- Différents types de méthodes

1. Constructeurs et destructeurs

1.1. Constructeurs

Une fonction membre portant le même nom que sa classe se nomme constructeur. Dès qu'une classe comporte un constructeur, lors de la déclaration d'un objet de cette classe, il faut fournir des valeurs pour les arguments requis par ce constructeur. Le constructeur est appelé après l'allocation de l'espace mémoire destiné à l'objet.

1.2. Destructeurs

Une fonction membre portant le même nom que sa classe, précédé du signe (~), se nomme un destructeur. Le destructeur est appelé avant la libération de l'espace mémoire associé à l'objet.

Constructeur et destructeur ne renvoie pas de valeur (aucune indication de type, même pas void !!).

Un destructeur ne peut pas comporter d'arguments .

2. Pointeur interne

Très souvent, les objets sont utilisés de manière dynamique, et ne sont donc créés que lors de l'exécution. Si les méthodes sont toujours communes aux instances d'un même type objet, il n'en est pas de même pour les données.

Il peut donc se révéler indispensable pour un objet de pouvoir se référencer lui-même. Pour cela, toute instance dispose d'un pointeur interne vers elle-même.

3. Visibilité

De par le principe de l'encapsulation, afin de pouvoir garantir la protection des données, il convient de pouvoir masquer certaines données et méthodes internes les gérant, et de pouvoir laisser visibles certaines autres devant servir à la gestion publique de l'objet. C'est le principe de la visibilité.

3.1. Champs et méthodes publics

Comme leur nom l'indique, les champs et méthodes dits publics sont accessibles depuis tous les descendants et dans tous les modules : programme, unité...

On peut considérer que les éléments publics n'ont pas de restriction particulière.

Les méthodes publiques sont communément appelées accesseurs : elles permettent d'accéder aux champs d'ordre privé.

3.2. Champs et méthodes privés

La visibilité privée restreint la portée d'un champ ou d'une méthode au module où il ou elle est déclaré(e). Ainsi, si un objet est déclaré dans une unité avec un champ privé, alors ce champ ne pourra être accédé qu'à l'intérieur même de l'unité.

Cette visibilité est à bien considérer. En effet, si un descendant doit pouvoir accéder à un champ ou une méthode privé(e), alors ce descendant doit nécessairement être déclaré dans le même module que son ancêtre.

3.3. Champs et méthodes protégés

La visibilité protégé correspond à la visibilité privé excepté que tout champ ou méthode protégé(e) est accessible dans tous les descendants, quel que soit le module où ils se situent.

4. Surcharge et redéfinition des méthodes d'une classe

Nous avons déjà parlé de la possibilité, lors de la création d'une classe héritant d'une autre classe, de surcharger certaines de ses méthodes. La surcharge, signalée par le mot-clé `override` suivant la déclaration d'une méthode déjà existante dans la classe parente, n'est possible que sous certaines conditions :

Vous devez avoir accès à la méthode surchargée. Ainsi, si vous êtes dans la même unité que la classe parente, vous pouvez ignorer cette restriction. Sinon, en dehors de la même unité, la méthode doit être au minimum protégée.

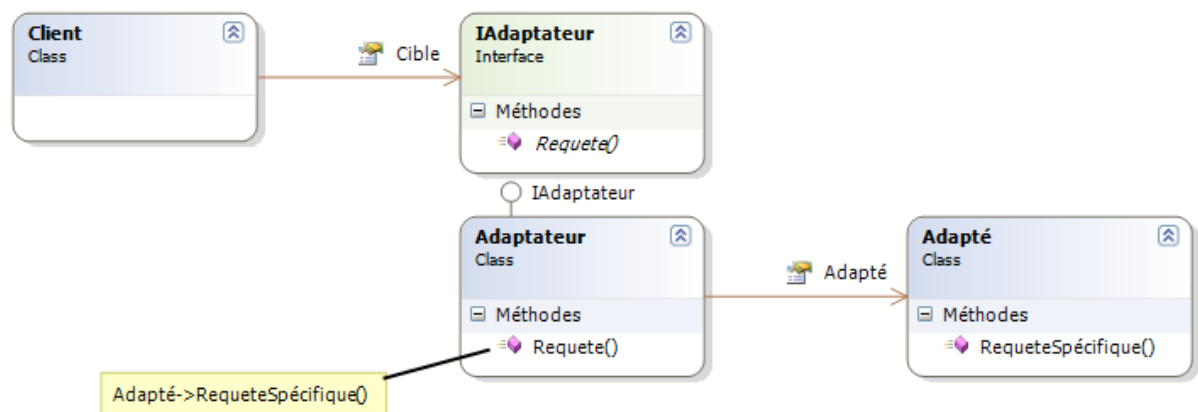
méthode de la classe parente doit être soit virtuelle (déclarée par le mot-clé `virtual`) soit déjà surchargée (`override`).

III- Adapter pattern

1. But

Il permet de convertir l'interface d'une classe en une autre interface que le client attend. L' Adaptateur fait fonctionner ensemble des classes qui n'auraient pas pu fonctionner sans lui, à cause d'une incompatibilité d'interfaces.

2. Structure



3. Participants

IAadaptateur : Définit l'interface métier utilisée par le Client.

Client : Travaille avec des objets utilisant l'interface **IAadaptateur**.

Adapté : Définit une interface existante devant être adaptée.

Adaptateur : Fait correspondre l'interface de **Adapté** à l'interface **IAadaptateur**.

4. Utilisations connues

On peut également utiliser un adaptateur lorsque l'on ne veut pas développer toutes les méthodes d'une certaine interface. Par exemple, si l'on doit développer l'interface `MouseListener` en Java, mais que l'on ne souhaite pas développer de comportement pour toutes les méthodes, on peut dériver la classe `MouseAdapter`. Celle-ci fournit en effet un comportement par défaut (vide) pour toutes les méthodes de `MouseListener`.

Exemple avec le `MouseAdapter` :

```
public class MouseBeeper extends MouseAdapter
{
    public void mouseClicked(MouseEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Exemple avec le `MouseListener` :

```
public class MouseBeeper implements MouseListener
{
    public void mouseClicked(MouseEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }

    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}
```

IV- Exemples

1. Class Point

Un point est défini par 2 valeurs abscisse et ordonné.

J'ai ajouté sur la même classe certaine Méthode, surtout la surcharge des opérateurs.

- Méthode pour déplacer un point
- Méthode pour multiplier deux points
- Méthode de surcharge de l'opérateur affectation (+=)
- Méthode de surcharge de l'opérateur de test d'égalité (==)
- J'ai ajouté aussi une méthode qui nous permet de savoir le nombre des points créer

point Class Reference

```
#include <point.h>
```

[List of all members.](#)

Public Member Functions

	point	(int x=0, int y=0)
	~point	()
point	somme	(point p2)
void	afficher	()
void	deplacer	(int dx, int dy)
point	multiplication	(point p2)
int	count	()
point &	operator+=	(int *)
point &	operator+=	(const point &)
bool	operator==	(const point &)
point &	operator=	(point &)

Constructor & Destructor Documentation

```
point::point ( int x = 0,  
              int y = 0  
              )
```

```
point::~~point ( )
```

Member Function Documentation

```
void point::afficher ( )
```

```
int point::count ( )
```

```
void point::deplacer ( int dx,  
                      int dy  
                      )
```

```
point point::multiplication ( point p2 )
```

```
point& point::operator+= ( const point & )
```

```
point& point::operator+= ( int * )
```

```
point& point::operator= ( point & )
```

```
bool point::operator== ( const point & )
```

```
point point::somme ( point p2 )
```

2. Class Segment

On utilisant la classe précédemment créer (la classe point) on a créé une autre classe segment, le segment est composée de deux points origine et extrémité.

J'ai implémenté dans cette classe les méthodes suivantes :

- Méthode qui calcul la longueur du segment
- Méthode qui renvoi l'origine du segment
- Méthode qui renvoi extrémité du segment
- Méthode qui calcul le point milieu du segment

Public Member Functions

	<code>segment (point, point)</code>
<code>double</code>	<code>longueur ()</code>
<code>point</code>	<code>originee ()</code>
<code>point</code>	<code>extremitee ()</code>
<code>point</code>	<code>point_mil ()</code>
<code>void</code>	<code>afficher ()</code>
	<code>~segment ()</code>

Constructor & Destructor Documentation

```
segment::segment ( point o,  
                  point ex  
                  )
```

```
segment::~~segment ( )
```

Member Function Documentation

```
void segment::afficher ( )
```

```
point segment::extremitee ( )
```

```
double segment::longueur ( )
```

```
point segment::originee ( )
```

```
point segment::point_mil ( )
```

3. Class pile

Dans ce programme on a créé une classe pile à l'aide d'une liste chaînée.

J'ai décidé de refaire le même projet mais cette fois à l'aide d'un tableau dynamique, en ajoutant à ça la notion Template.

Et j'ai ajouté une méthode qui nous permet de compter les éléments dans notre pile.

Public Member Functions

	<code>pile (int size)</code>
	<code>~pile ()</code>
<code>int</code>	<code>push (const T &)</code>
<code>int</code>	<code>pop (T &)</code>

```
template<class T>
class pile< T >
```

Constructor & Destructor Documentation

```
template<class T >
pile< T >::pile ( int size ) [inline]
```

```
template<class T >
int pile< T >::~~pile ( ) [inline]
```

Member Function Documentation

```
template<class T >
int pile< T >::pop ( T & resultat ) [inline]
```

```
template<class T >
int pile< T >::push ( const T & element ) [inline]
```

4. class Rationnel

Sur l'exemple suivant j'ai créé une classe rationnel qui va nous permettre de manipuler les nombre rationnel, rappelons que un nombre rationnel est constituer d'un numérateur et un dénominateur.

J'ai implémenté dans cette classe plusieurs méthodes :

- Constructeur par défaut
- Constructeur avec paramètres
- Destructeur
- Méthode qui renvoi la partie entière
- Méthode qui renvoi la partie réelle
- Méthode qui fait la somme de 2 rationnels
- Méthode d'affichage
- Méthode pour inverser un rationnel
- Méthode pour calculer le pgcd
- Et finalement j'ai surchargé les opérateurs `+`, `-`, `*`, `/` pour la classe rationnel

Public Member Functions

	rational ()
	rational (int, unsigned int)
	~rational ()
int	v_entiere ()
float	v_reel ()
rational	somme (const rational *)
void	afficher ()
rational	inverser ()
rational	pgcd ()
rational &	operator+= (const rational *)
rational &	operator-= (const rational *)
rational &	operator*= (const rational *)
rational &	operator/= (const rational *)

Constructor & Destructor Documentation

rational::rational ()

rational::rational (int **a**,
 unsigned int **b**
)

rational::~rational ()

Member Function Documentation

void `rational::afficher ( )`

`rational` **rational::inverser ()**

`rational` & **rational::operator*= (const `rational` * c)**

`rational` & **rational::operator+= (const `rational` * c)**

`rational` & **rational::operator-= (const `rational` * c)**

`rational` & **rational::operator/= (const `rational` * c)**

`rational` **rational::pgcd ()**

`rational` **rational::somme (const `rational` * a)**

int `rational::v_entiere ( )`

float `rational::v_reel ( )`

Exercice dematrice :

Il nous faut d'abord une class "Erreur" qui génère les erreurs de la classe matrice

Classe erreur :

Public Member Functions

	<code>Erreur ()</code>
	<code>Erreur (Erreur &s)</code>
	<code>Erreur (int a)</code>
int	<code>cause () const</code>
string	<code>explic () const</code>
	<code>~Erreur ()</code>

Static Public Attributes

static const int	<code>ncarre = 0</code>
static const int	<code>multiplication = 1</code>
static const int	<code>notinmat = 2</code>
static const int	<code>dbzero = 3</code>

Constructor & Destructor Documentation

`Erreur::Erreur ( )`

`Erreur::Erreur ( Erreur & s )`

`Erreur::Erreur ( int a )`

`Erreur::~~Erreur ( )`

Member Function Documentation

`int Erreur::cause ( ) const`

`string Erreur::explic ( ) const`

5. Classe matrice :

Public Member Functions

	Matrice ()
	Matrice (int, int)
	Matrice (const Matrice &)
Matrice &	operator= (const Matrice &)
	~Matrice ()
double	operator() (int, int)
Matrice &	operator* = (const Matrice &)
void	afficher ()

Friends

istream & **operator>>** (istream &, **Matrice** &)

Constructor & Destructor Documentation

Matrice::Matrice ()

Matrice::Matrice (int nx,
 int mx
)

Matrice::Matrice (const **Matrice** & a)

Matrice::~~Matrice ()

6. Classe forme :

On utilisant la classe précédemment créer "la classe point" on a créé une autre classe forme

Public Member Functions

<code>form (point)</code>
<code>void PointInForme (int x, point y)</code>
<code>point deplacer (int dx, int dy)</code>
<code>void Intersecte (int x, form y)</code>
<code>~form ()</code>

Constructor & Destructor Documentation

```
form::form ( point o )
```

```
form::~~form ( )
```

Member Function Documentation

```
point form::deplacer ( int dx,  
                      int dy  
                      )
```

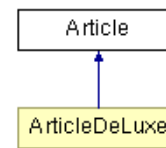
```
void form::Intersecte ( int x,  
                       form y  
                       )
```

```
void form::PointInForme ( int x,  
                         point y  
                         )
```

7. Classearticle :

```
#include <Article.h>
```

Inheritance diagram for Article:



List of all members.

Public Member Functions

double	TVA () const
	Article (std::string, double qt, double px)
	~Article (void)
void	Ajouter (double)
void	Retirer (double)
void	print () const

Detailed Description

Definition at line 3 of file **Article.h**.

Constructor & Destructor Documentation

```
Article::Article ( std::string s,  
                  double qt,  
                  double px  
                  )
```

Definition at line 6 of file **Article.cpp**.

```
Article::~Article ( void )
```

Definition at line 27 of file **Article.cpp**.

Member Function Documentation

```
void Article::Ajouter ( double qt )
```

Definition at line 13 of file **Article.cpp**.

```
void Article::print ( ) const
```

Definition at line 37 of file **Article.cpp**.

```
void Article::Retirer ( double qt )
```

Definition at line 18 of file **Article.cpp**.

```
double Article::TVA ( ) const
```

Reimplemented in **ArticleDeLuxe**.

Definition at line 32 of file **Article.cpp**.

8. Gestion des Compitions

Documentation des classes

Référence de la classe Date

```
#include <Date.h>
```

Fonctions membres publiques

- **Date** (int, int, int)
- **~Date** ()
- void **print** ()

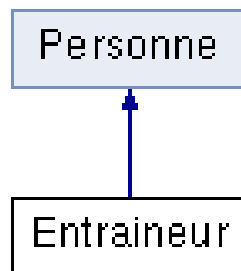
La documentation de cette classe a été générée à partir des fichiers suivants :

- **Date.h**
- **Date.cpp**

Référence de la classe Entraîneur

```
#include <Entraîneur.h>
```

Graphe d'héritage de Entraîneur:



Fonctions membres publiques

- **Entraîneur** (string, string, string, int)
- int **trainthiseqp** (int)
- **~Entraîneur** ()

La documentation de cette classe a été générée à partir des fichiers suivants :

- **Entraîneur.h**
- **Entraîneur.cpp**

Référence de la classe Equipe

```
#include <Equipe.h>
```

Fonctions membres publiques

- **Equipe** (int, string)
- void **print** ()
- int **recherche** (int)
- ~**Equipe** ()

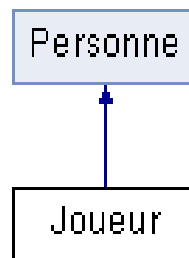
La documentation de cette classe a été générée à partir des fichiers suivants :

- **Equipe.h**
- **Equipe.cpp**

Référence de la classe Joueur

```
#include <Joueur.h>
```

Graphe d'héritage de Joueur:



Fonctions membres publiques

- **Joueur** (string, string, string, int, int)
- int **InEquip** (int)
- ~**Joueur** ()

La documentation de cette classe a été générée à partir des fichiers suivants :

- **Joueur.h**
- **Joueur.cpp**

Référence de la classe Matche

```
#include <Matche.h>
```

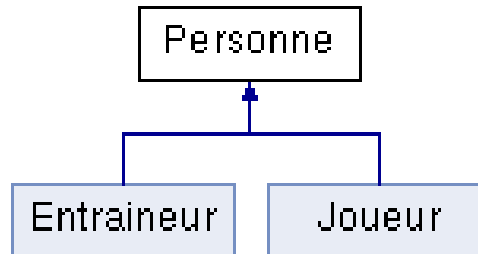
Fonctions membres publiques

- **Matche** (int e1, int e2, int b1, int b2, int jo, int mo, int an)
- void **print** ()
- int **equip1** ()

- `int equip2 ()`
- `~Matche ()`

```
#include <Personne.h>
```

Graphe d'héritage de Personne:



Fonctions membres publiques

- `Personne (string, string, string)`
- `~Personne ()`
- `void print ()`

La documentation de cette classe a été générée à partir des fichiers suivants :

- **Personne.h**
- **Personne.cpp**

Documentation des fichiers

Référence du fichier Date.cpp

```
#include "stdafx.h"
#include "Date.h"
```

Référence du fichier Date.h

```
#include <iostream>
```

Classes

- class **Date**

Référence du fichier Entraîneur.cpp

```
#include "stdafx.h"
#include "Entraîneur.h"
```

Référence du fichier Entraîneur.h

```
#include "Personne.h"
```

Classes

- class **Entraîneur**

Classes

- class **Equipe**

Référence du fichier Gestioncompetition.cpp

```
#include "stdafx.h"
#include <conio.h>
#include "Matche.h"
#include "Joueur.h"
#include "Entraîneur.h"
```

Macros

- #define **nbmx** 30
- #define **nbj** 100

Fonctions

- void **AddTeam** (**Equipe** **eq, int *i)
- void **AddPlayer** (**Joueur** **js, int *i)
- void **AddTrainer** (**Entraîneur** **ens, int *i)
- void **OragMatch** (**Matche** **mts, int *im)
- void **show** (**Matche** **m, int im, **Equipe** **eq, int ieq, **Joueur** **js, int ijo, **Entraîneur** **ens, int ient)
- void **fre** (**Equipe** **eqp, **Entraîneur** **ens, **Joueur** **js, **Matche** **mts, int ie, int im, int ij, int ient)
- void **menu** (**Equipe** **eqp, **Entraîneur** **ens, **Joueur** **js, **Matche** **mts, int *ie, int *im, int *ij, int *ient)
- int **_tmain** (int argc, _TCHAR *argv[])