# Algorithme et Programmation

Romain Bourqui
Maitre de Conférences



*IUT Informatique Semestre 2*

1er février 2010

# Algorithme et Programmation 2

## *Objectifs*

Initiation à la
**Programmation Orientée Objet**
(héritage, polymorphisme,
agrégation, classe abstraite)

Algorithmique : manipulation de
**Types Abstraits de Données**

# Organisation du Cours

# Algorithme et Programmation 2

## *Organisation du cours*

- **Programmation (C++) :**
  **STL, classe, héritage, agrégation, polymorphisme, ...**
- **Algorithmique :**
  **Type Abstrait de Données (TAD)**

# Algorithme et Programmation 2

## *Organisation du cours : TDs*

- **Semaines 1 – 4 :**
  Introduction des classes

# Algorithme et Programmation 2

## *Organisation du cours : TDs*

- **Semaines 1 – 4 :**
  Introduction des classes
- **Semaines 5 – 7 :**
  Héritage et polymorphisme

# Algorithme et Programmation 2

## *Organisation du cours : TDs*

- **Semaines 1 – 4 :**
  Introduction des classes
- **Semaines 5 – 7 :**
  Héritage et polymorphisme
- **Semaines 8 – 14 :**
  Type Abstrait de Données

# Algorithme et Programmation 2

## *Organisation du cours : TDs*

- **Semaines 1 – 4 :**
  Introduction des classes
- **Semaines 5 – 7 :**
  Héritage et polymorphisme
- **Semaines 8 – 14 :**
  Type Abstrait de Données
- **Semaines 15 – 16 :**
  Implémentation de TAD

# Algorithme et Programmation 2

## *Organisation du cours : TDs*

- **Semaines 1 – 4 :**
  Introduction des classes
- **Semaines 5 – 7 :**
  Héritage et polymorphisme

} C++

- **Semaines 8 – 14 :**
  Type Abstrait de Données
- **Semaines 15 – 16 :**
  Implémentation de TAD

# Algorithme et Programmation 2

## *Organisation du cours : TDs*

- **Semaines 1 – 4 :**
  Introduction des classes
- **Semaines 5 – 7 :**
  Héritage et polymorphisme

} C++

- **Semaines 8 – 14 :**
  Type Abstrait de Données
- **Semaines 15 – 16 :**
  Implémentation de TAD

} Algorithmique

# Algorithme et Programmation 2

## *Organisation du cours : TPs (TD machine double)*

- **Semaines 1 – 4 :**
  Introduction des classes
- **Semaines 5 – 7 :**
  Héritage et polymorphisme
- **Semaines 8 – 14 :**
  Type Abstrait de Données
- **Semaines 15 – 16 :**
  Implémentation de TAD

# Algorithme et Programmation 2

*Organisation du cours : TPs (TD machine double)*

- **Semaines 1 – 4 :**
  Introduction des classes
- **Semaines 5 – 7 :**
  Héritage et polymorphisme
- **Semaines 8 – 14 :**
  Type Abstrait de Données
- **Semaines 15 – 16 :**
  Implémentation de TAD

C++

# Algorithme et Programmation 2

## *Organisation du cours : Evaluation*

- **Semaine 4 :** Interrogation en TD
- **Semaines 9 ou 10 :** TP Noté
- **Semaine ~11 :** Interrogation en TD
- **Fin du semestre :** DS final

# Algorithme et Programmation 2

## *Organisation du cours : Evaluation*

- **Semaine 4 :** Interrogation en TD
- **Semaines 9 ou 10 :** TP Noté
- **Semaine ~11 :** Interrogation en TD
- **Fin du semestre :** DS final
- **... Interrogations surprises ...**

# Algorithme et Programmation 2

## *Organisation du cours : Evaluation*

- **Semaine 4 :** Interrogation en TD
- **Semaines 9 ou 10 :** TP Noté
- **Semaine ~11 :** Interrogation en TD
- **Fin du semestre :** DS final
- **... Interrogations surprises ...**

**Projet de Programmation**

# Programmation Orientée Objet : un aperçu

# Algorithme et Programmation 2

## *Programmation impérative*

Séparation des **données** et des **traitements** sur ces données (Fortran, Cobol, Pascal, Basic, C)

**Type** de donnée : définition des seules valeurs possibles

**Opérations** sur le type : définies **séparément** sous forme de fonctions

# Algorithme et Programmation 2

## *Programmation impérative : exemple*

```
struct Personne {
  string prenom;
  string nom;
  int age;
} ;
```

Structure Personne

```
void initialise ( Personne & p, string pr, string n, int a );
void quiSuisJe ( Personne p );
void anniversaire ( Personne & p );
```

Fichier.h

```
#include "Personne.h"

void initialise( Personne & p, string pr, string n, int a ) {
  p.prenom = pr;
  p.nom = n;
  p.age = a;
}

void quiSuisJe( Personne p ) {
  cout << "je suis " << p.prenom << " " << p.nom << endl;
  cout << "j'ai " << p.age << " ans" << endl;
}

void anniversaire( Personne & p ) {
  p.age++;
}
```

Fichier.cc

# Algorithme et Programmation 2

## *Programmation impérative : exemple*

```
struct Personne {
  string prenom;
  string nom;
  int age;
} ;
```

Structure Personne

```
void main() {
  Personne jpp;
  initialise( jpp, "Jean-Pierre","Papin", 50 );
  quiSuisJe( jpp );
  anniversaire( jpp );

  // ...

  Personne zz;
  zz.prenom = "Zinedine";
  zz.nom = "Zidane";
  zz.age = 40;
  zz.age++;
}
```

progPrincipal.cc

## *Programmation impérative : Problèmes*

**Variables non encapsulées** : on peut lire ou modifier ces variables partout
**Variables mal initialisées :** inconsistantes de certaines valeurs
**Traitements séparés :** connaissance imprécise de toutes les opérations possibles sur le type ; réécriture de fonctions existantes ; définition de traitements incohérents avec la sémantique du type

# Algorithme et Programmation 2

## *POO*

***P**rogrammation **O**rientée **O**bjet **(POO)***

- **Offre** un mécanisme de classe rassemblant données et traitements
- **C++ :** mélange programmations objet et impérative (Programmation **« Orientée »** Objet)

## POO

**Une classe** : type de données rassemblant (sous un même nom) **données** et **traitements**

- Données attachées à une classe : **attributs (ou données membres)**
- **T**raitements attachés à une classe : **méthodes** (ou **fonctions membres**, **opérations**)

# Algorithme et Programmation 2

## *POO : exemple*

**Interface : .h**

attributs / prototypes des méthodes

```
// Personne.h
class Personne {
private:
  // attributs ou données membres
  string my_prenom;
  string my_nom;
  int my_age;
public:
  // méthodes ou fonctions membres
  void initialise( string pr, string n, int a );
  void quiSuisJe( );
  void anniversaire( );
} ;
```

**Corps** :

fichier source **.cc** (ou .cpp, .C, .cxx)

```
// Personne.cc
#include "Personne.h"

void Personne::initialise( string pr , string n, int a ) {
  my_prenom = pr;
  my_nom = n;
  my_age = a;
}
void Personne::quiSuisJe( ) {
  cout << "je suis " << my_prenom << " " << my_nom << endl;
  cout << "j'ai " << my_age << " ans" << endl;
}
void Personne::anniversaire( ) {
  my_age++;
}
```

# Algorithme et Programmation 2

## *POO: Conventions*

- **Classes** ou structures **:** commencent par des majuscules (class Point, class Vecteur, class TriangleIsocele, ...)
- **Attributs** ou données membres d'une classe : en minuscules et commencent par my_ (ou mon_, ou m_, ou _)
- **Méthodes** ou fonctions membres : commencent par une minuscule (void initialise(), float vitesseAngulaire(), ...)
- **Paramètres** d'une fonction et variables locales : en minuscules avec séparation des mots
- **Constantes** : en majuscules (const float PI = 3.14)

## *POO: Instance d'une classe/Objet*

**Objet = instance d'une classe**

- variable dont le type est une classe
- un objet est la réalisation effective d'une classe

## POO: Instance d'une classe/Objet

```
#include "Personne.h"

// ...

Personne jpp;
Personne zz;

// ...
```

progPrincipal.cc

# Algorithme et Programmation 2

## *POO : Visibilité des membres*

- **public :** accès pour chaque instance
  - *Exemple* : initialise, quiSuisJe et anniversaire utilisables pour chaque instance de Personne
- **private :** accès restreint aux seuls corps des méthodes de la class
  - *Exemple* : my_prenom, my_nom et my_age accessibles seulement dans les corps des méthodes de la classe Personne.
- **protected :** private + accès autorisé aux corps des méthodes des classes qui héritent de cette classe

# Algorithme et Programmation 2

## *POO : exemple*

**Interface : .h**

attributs / prototypes des méthodes

```
// Personne.h
class Personne {
private:
  // attributs ou données membres
  string my_prenom;
  string my_nom;
  int my_age;
public:
  // méthodes ou fonctions membres
  void initialise( string pr, string n, int a );
  void quiSuisJe( );
  void anniversaire( );
} ;
```

**Corps** :

fichier source **.cc** (ou .cpp, .C, .cxx)

```
// Personne.cc
#include "Personne.h"

void Personne::initialise( string pr , string n, int a ) {
  my_prenom = pr;
  my_nom = n;
  my_age = a;
}
void Personne::quiSuisJe( ) {
  cout << "je suis " << my_prenom << " " << my_nom << endl;
  cout << "j'ai " << my_age << " ans" << endl;
}
void Personne::anniversaire( ) {
  my_age++;
}
```

## *POO : Accès aux attributs/méthodes*

**Accès aux attributs et aux méthodes d'un objet :** notation pointée " . "

- Exemple :

```
#include "Personne.h"

// ...

Personne jpp;
jpp.initialise ( "Jean-Pierre", "Papin", 50 );
jpp.quiSuisJe();

// ...
```

## *POO : Accès aux attributs/méthodes*

**Attributs** de la classe Personne sont **« private »**

- utiliser les **fonctions membres** pour les manipuler
- la méthode **initialise(...)** est nécessaire pour leur donner des valeurs

## *POO : Constructeurs*

**Constructeurs** : définition de comportements particuliers lors de l'instanciation d'une classe (notamment pour initialiser un nouvel objet)

## *POO : Constructeurs*

**Constructeurs** : définition de comportements particuliers lors de l'instanciation d'une classe (notamment pour initialiser un nouvel objet)

- Les **constructeurs** portent tous le **nom de la classe** :
  - méthodes **Personne** ( … ) pour la classe **Personne**

## *POO : Constructeurs*

**Constructeurs** : définition de comportements particuliers lors de l'instanciation d'une classe (notamment pour initialiser un nouvel objet)

- Les **constructeurs** portent tous le **nom de la classe** :
  - méthodes **Personne** ( … ) pour la classe **Personne**
- **Plusieurs constructeurs** différenciés par la liste et les types des paramètres

## *POO : Constructeurs*

- Exemple remplaçant **initialise(...)**

# Algorithme et Programmation 2

## *POO : Constructeurs*

Personne.h

```
class Personne {
private:
   // attributs ou données membres
   string my_prenom;
   string my_nom;
   int my_age;
public:
   ...
   // Constructeur
   Personne( string pr, string n, int a );
   ...
} ;
```

Personne.cc

```
// Personne.cc
#include "Personne.h"

Personne::Personne( string pr , string n, int a ) {
   my_prenom = pr;
   my_nom = n;
   my_age = a;
}
```

## *POO : Constructeurs*

- Exemple remplaçant **initialise(...)**
- **Constructeur par défaut** ou constructeur sans paramètre :
  - appelé lors de l'instanciation d'un objet sans arguments d'appel
  - appelé lors de l'instanciation d'un tableau d'objets sur chacune des positions du tableau.

# Algorithme et Programmation 2

## *POO : Constructeurs*

Personne.h

```
class Personne {
private:
  // attributs ou données membres
  string my_prenom;
  string my_nom;
  int my_age;
public:
  // ...
  //constructeurs
  Personne();
  Personne( string pr , string n , int a );
  // ...
} ;
```

Personne.cc

```
//Personne.cc
#include "Personne.h"

Personne::Personne(){
  // ...
}
```

## POO : Constructeurs

```
_


//  appel du constructeur par défaut de Personne
Personne inconnu;

// pour chaque Personne du tableau groupe, appel du constructeur par défaut
Personne groupe[5];
```

progPrincipal.cc

## *POO : Constructeurs*

- Exemple remplaçant **initialise(...)**
- **Constructeur par défaut** ou constructeur sans paramètre :
  - appelé lors de l'instanciation d'un objet sans arguments d'appel
  - appelé lors de l'instanciation d'un tableau d'objets sur chacune des positions du tableau.
- Autre constructeur : **le constructeur par copie...** (cf. TD)

## *POO : Destructeur*

**Le destructeur** : méthode particulière définie implicitement pour toutes les classes

- Son nom : **~nom_classe**
- un destructeur unique par classe
  - appelé lors de la destruction/désallocation de l'objet
  - nécessaire quand on fait appel à l'allocation dynamique…

# Algorithme et Programmation 2

## POO : Accesseurs

```
void affTab (Personne tab[], int n) {
  int i;
  for (i=0 ; i<n ; i++)
    tab[i].quiSuisJe();
}
```

## POO : Accesseurs

```
void affTab (Personne tab[], int n) {
  int i;
  for (i=0 ; i<n ; i++)
    tab[i].quiSuisJe();
}
```

**Comment n'afficher que les personnes ayant plus de 25 ans ?**

## *POO : Accesseurs*

```
void affTab (Personne tab[], int n) {
  int i;
  for (i=0 ; i<n ; i++)
    tab[i].quiSuisJe();
}
```

**Comment n'afficher que les personnes ayant plus de 25 ans ?**

- Impossible : l'accès aux attributs **my_age** et **my_prenom** est interdit (**private**)

# Algorithme et Programmation 2

## *POO : Accesseurs*

```
void affTab (Personne tab[], int n) {
  int i;
  for (i=0 ; i<n ; i++)
    tab[i].quiSuisJe();
}
```

**Comment n'afficher que les personnes ayant plus de 25 ans ?**

- Impossible : l'accès aux attributs **my_age** et **my_prenom** est interdit (**private**)

=> Les **accesseurs** permettent de retourner les attributs privés nécessaires à une fonction **non membre** de la classe

# Algorithme et Programmation 2

## *POO : Accesseurs*

Personne.h

```
class Personne {
private:
  // attributs ou données membres
  string my_prenom;
  string my_nom;
  int my_age;
public:
  // ...
  // accesseurs
  string getPrenom() const;
  string getNom() const;
  int getAge() const;
  // ...
} ;
```

Personne.cc

```
//Personne.cc
#include "Personne.h"

string Personne::getPrenom() const
{ return my_prenom; }
string Personne::getNom() const
{ return my_nom; }
int Personne::getAge() const
{ return my_age; }
```

## *POO : Encapsulation des attributs*

**Encapsuler les attributs et définir des accesseurs**

- utilisation d'une classe est rendue totalement indépendante de son implémentation

# Algorithme et Programmation 2

## *POO : Accesseurs*

Personne.h

```
class Personne {
private:
  string my_prenom_nom[2];
  int my_age;
public:
  // ...
  // accesseurs
  string getPrenom() const;
  string getNom() const;
  int getAge() const;
  // ...
};
```

Personne.cc

```
//Personne.cc
#include "Personne.h"

string Personne::getPrenom() const
{ return my_prenom_nom[0]; }
string Personne::getNom() const
{ return my_prenom_nom[1]; }
int Personne::getAge() const
{ return my_age; }
```

# Algorithme et Programmation 2

## *POO : Accesseurs en **écriture***

Personne.h

```
class Personne {
private:
  string my_prenom_nom[2];
  int my_age;
public:
  // ...
  // accesseurs
  void setPrenom( string pr );
  void setNom( string n );
  void setAge( int a );
  // ...
};
```

Personne.cc

```
// Personne.cc
#include "Personne.h"

void Personne::setPrenom( string pr )
{ my_prenom_nom[0]=pr; }
void Personne::setNom( string n )
{ my_prenom_nom[1]=n; }
void Personne::setAge( int a )
{ my_age=a; }
```

# Algorithme et Programmation 2

## POO : Classe Personne

```
class Personne {
private:
  string my_prenom_nom[2];
  int my_age;
public:
  // constructeurs
  Personne();
  Personne( const Personne &p);
  Personne( string pr, string n , int a );

  // accesseurs en lecture
  string getPrenom() const;
  string getNom() const;
  int getAge() const;

  // accesseurs en écriture
  void setPrenom( string pr );
  void setNom( string n );
  void setAge( int a );

  void quiSuisJe();
  void anniversaire();
};
```

```
#include "Personne.h"

Personne::Personne(){
  my_prenom_nom[0] = "";
  my_prenom_nom[1] = "";
  my_age = -1;
}
Personne::Personne( const Personne &p){
  my_prenom_nom[0] = p.my_prenom_nom[0];
  my_prenom_nom[1] = p.my_prenom_nom[1];
  my_age = p.my_age;
}
Personne::Personne( string pr, string n , int a ){
  my_prenom_nom[0] = pr;
  my_prenom_nom[1] = n;
  my_age = a;
}

string Personne::getPrenom() const
{ return my_prenom_nom[0]; }
string Personne::getNom() const
{ return my_prenom_nom[1]; }
int Personne::getAge() const
{ return my_age; }

void Personne::setPrenom( string pr )
{ my_prenom_nom[0]=pr; }
void Personne::setNom( string n )
{ my_prenom_nom[1]=n; }
void Personne::setAge( int a )
{ my_age=a; }

void Personne::quiSuisJe(){
  cout << "je suis " << my_prenom_nom[0] << " " << my_prenom_nom[1] << endl;
  cout << "j'ai " << my_age << " ans" << endl;
}

void Personne::anniversaire(){
  my_age++;
}
```

## *POO : Agrégation*

L'**agrégation** permet d'assembler des objets de base afin de définir des objets plus complexes

==> définir des objets composés d'autres objets

- Exemple :

On peut imaginer une classe Groupe comme étant l'agrégation d'une classe Etudiant

## *POO : Héritage (simple)*

L'**héritage** est un mécanisme de transmission des caractéristiques d'une classe (attributs et méthodes) vers une sous-classe.

=> permet d'exprimer entre les classes une relation de type « **est une sorte de** ».

- Exemple :

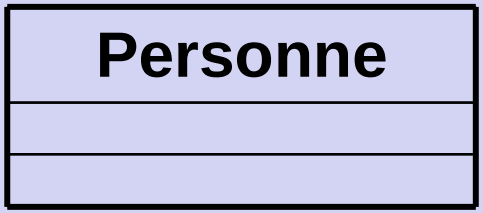
On peut définir une classe Etudiant comme une spécialisation de la classe Personne

## *POO : Héritage (simple)*

- **L'héritage** permet ainsi de déclarer des **classes très générales**, puis progressivement de les « **spécialiser** ».
- Cette spécialisation est aussi une « **dérivation** » de la classe de base.
- On parle alors de **hiérarchie d'héritage** et donc de **graphe d'héritage**.

# Algorithme et Programmation 2

## *POO : Héritage (simple)*

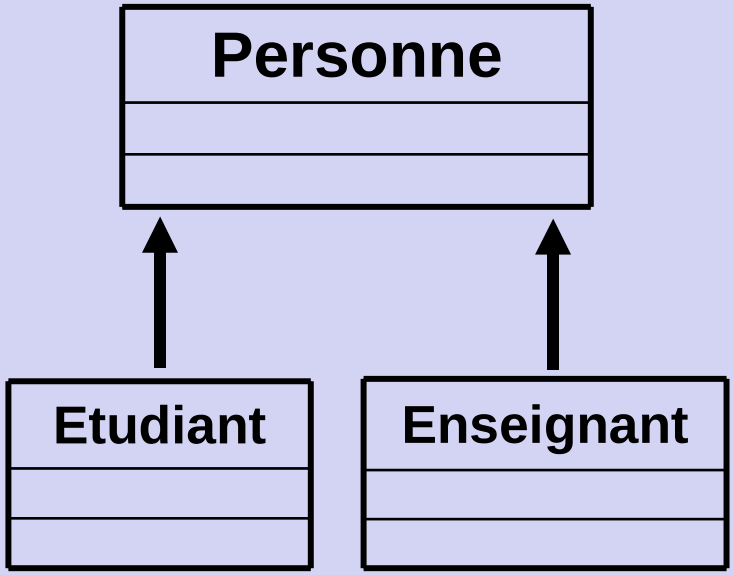



Exemple

# Algorithme et Programmation 2

## *POO : Héritage (simple)*

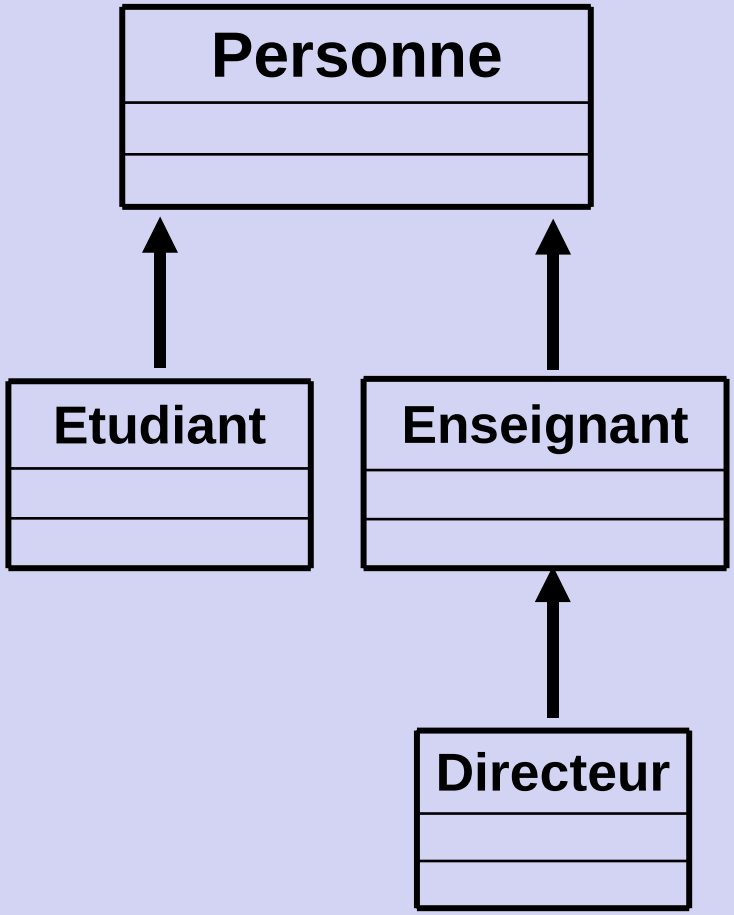
Exemple



# Algorithme et Programmation 2

## *POO : Héritage (simple)*

Exemple



# Algorithme et Programmation 2

## *POO : Héritage (simple)*

- La flèche signifie **« hérite de »** ou **« spécialise ».**

  Etudiant **hérite** de Personne

  Etudiant est une **sous-classe** de Personne

- L'héritage est un mécanisme qui **transmet les caractéristiques** d'une classe (attributs et méthodes) vers ses sous-classes :

  les classes qui héritent (Etudiant, Enseignant) intègrent les attributs et méthodes de la classe qu'elles spécialisent (Personne).

# Algorithme et Programmation 2

## *POO : Héritage (simple) en c++*

- Les classes Etudiant et Enseignant « spécialisent » certaines méthodes

- Exemple :

la méthode quiSuisJe() va pouvoir spécifier que je suis enseignant ou étudiant qu'elles spécialisent (Personne).

# Algorithme et Programmation 2

## *POO : Héritage (simple) en c++*

```
class Personne {
private:
  string my_prenom_nom[2];
  int my_age;
public:
  // constructeurs
  Personne();
  Personne( const Personne &p);
  Personne( string pr, string n , int a );

  //destructeur
  virtual ~Personne();

  // accesseurs en lecture
  string getPrenom() const;
  string getNom() const;
  int getAge() const;

  // accesseurs en écriture
  void setPrenom( string pr );
  void setNom( string n );
  void setAge( int a );

  virtual void quiSuisJe();
  void anniversaire();
};
```

```
class Enseignant: public Personne {
public:
  // constructeurs
  Enseignant();
  Enseignant( const Enseignant &p);
  Enseignant( string pr, string n , int a );

  // destructeur
  virtual ~Enseignant();

  virtual void quiSuisJe();
};
```

# Algorithme et Programmation 2

## *POO : Héritage (simple) en c++*

```
class Enseignant: public Personne {
public:
  // constructeurs
  Enseignant();
  Enseignant( const Enseignant &p);
  Enseignant( string pr, string n , int a );

  // destructeur
  virtual ~Enseignant();

  virtual void quiSuisJe();
};
```

```
void Enseignant::quiSuisJe(){
  cout << "je suis " << my_prenom_nom[0] << " " << my_prenom_nom[1] << endl;
  cout << "je suis un enseignat de " << my_age << " ans" << endl;
}
```

## *POO : Héritage VS Agrégation*

- L'**héritage** indique une relation **« est un »** :

  Ex. : un Etudiant est une Personne

- L'**agrégation** indique une relation du type **« fait partie de » :**

  Ex. : un Etudiant fait partie d'un Groupe

# Algorithme et Programmation 2

## *POO : Polymorphisme*

**Polymorphisme** = utilisation homogène d'objets distincts d'une une même hiérarchie d'héritage

- le comportement de ces objets reste **spécifique** (au type d'instanciation).
- Puissant mécanisme **d'abstraction** : possibilité de traiter plusieurs formes d'une classe comme si elles ne faisaient qu'une une…

## POO : Polymorphisme

```
int main(){
  Etudiant etd("Jonathan", "Dubois", 22);
  Enseignant ens("Romain", "Bourqui", 28);

  Personne * pers;

  pers = &etd;
  pers->quiSuisJe(); // utilise la méthode quiSuisJe() de la classe Etudiant

  pers = &ens;
  pers->quiSuisJe(); // utilise la méthode quiSuisJe() de la classe Enseignant

  return EXIT_SUCCESS;
}
```

Exemple de polymorphisme

## POO : Polymorphisme

```
int main(){
  Personne etd("Jonathan", "Dubois", 22);
  Enseignant ens("Romain", "Bourqui", 28);

  Personne pers;

  pers = etd;
  pers.quiSuisJe(); // utilise la méthode quiSuisJe() de la classe Etudiant

  pers = ens;
  pers.quiSuisJe(); // utilise la méthode quiSuisJe() de la classe Etudiant

  return EXIT_SUCCESS;
}
```

Exemple de polymorphisme

# Type Abstrait de Données : un aperçu

## *Type Abstrait de Données : Définition*

Pour définir un **TAD** on se donne

- une **notation** qui décrit les données,
- des **opérations** applicables à ces données (ou primitives), et
- les **propriétés** de ces opérations (ou sémantique)

## *Type Abstrait de Données : Exemple*

Le type **entier** :

- Chiffres décimaux, précédés de + ou -
- Opérations arithmétiques : +, -, *, /
- Sens usuel de ces opérations

## *Type Abstrait de Données : Exemple*

Le type **entier** :

- Chiffres décimaux, précédés de + ou -
- Opérations arithmétiques : +, -, *, /
- Sens usuel de ces opérations

On se soucie peu de la représentation **concrète**

## *Type Abstrait de Données : Quand?*

Généralement lorsqu'on éprouve le besoin de désigner des types de données non « ***primitifs*** », i.e. non nécessairement disponibles dans les langages de programmation courants

# Algorithme et Programmation 2

## *Types Abstraits de Données*

**4 TADs** ce semestre :

# Algorithme et Programmation 2

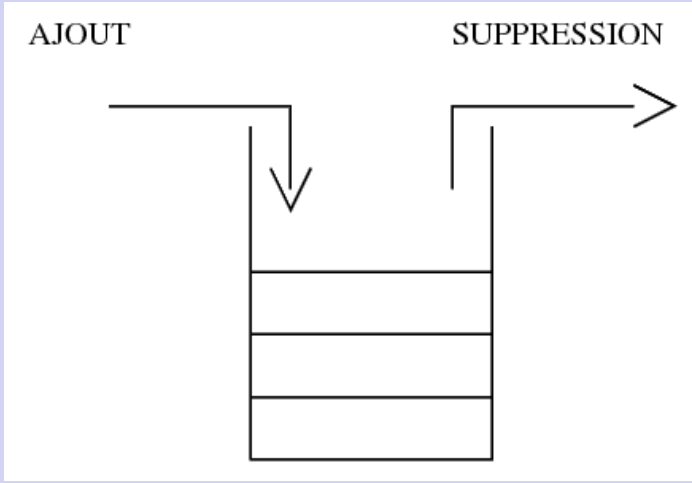
## *Types Abstraits de Données*

**4 TADs** ce semestre :

- **Pile**

## *Types Abstraits de Données : Pile*

**Description** : liste dans laquelle les ajouts/suppressions n'ont lieu qu'à une extrémité, appelée *sommet de la pile.*




Structure **LIFO** (*Last In Fisrt Out*)

# Algorithme et Programmation 2

## *Types Abstraits de Données : Pile*

Soit un type **TInfo**, un pile de ce type :

Type TPile = Pile de Tinfo

var P : TPile

ou encore var P : Pile de TInfo

# Algorithme et Programmation 2

## *Types Abstraits de Données : Pile*

Soit un type **TInfo**, un pile de ce type :

Type TPile = Pile de Tinfo

var P : TPile

ou encore var P : Pile de TInfo

**Primitives :**

action créerPile(S P : Tpile)

fonction pileVide(P : TPile) : Booléen

fonction valeurSommet(P : TPile) : Tinfo

action empiler(ES P : TPile, E Elem : Tinfo)

action empiler(S P : TPile)

# Algorithme et Programmation 2

## *Types Abstraits de Données*

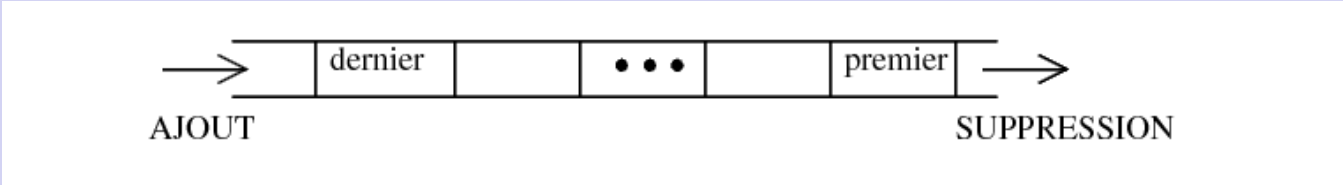
**4 TADs** ce semestre :

- **Pile**
- **File**

# Algorithme et Programmation 2

## *Types Abstraits de Données : File*

**Description** : liste dans laquelle les ajouts se font à une extrémité (*fin de liste*) et les suppressions à une autre (*tête de liste*)




Structure **FIFO** (*First In Fisrt Out*)

# Algorithme et Programmation 2

## *Types Abstraits de Données : File*

Soit un type **TInfo**, un file de ce type :

Type TFile = File de Tinfo

var F : TFile

ou encore var F : File de TInfo

**Primitives :**

action créerFile(S F : TFile)

fonction fileVide(F : TFile) : Booléen

fonction valeurPremier(F : TFile) : Tinfo

action enfiler(ES F : TFile, E Elem : Tinfo)

action défiler(S F : TFile)

# Algorithme et Programmation 2

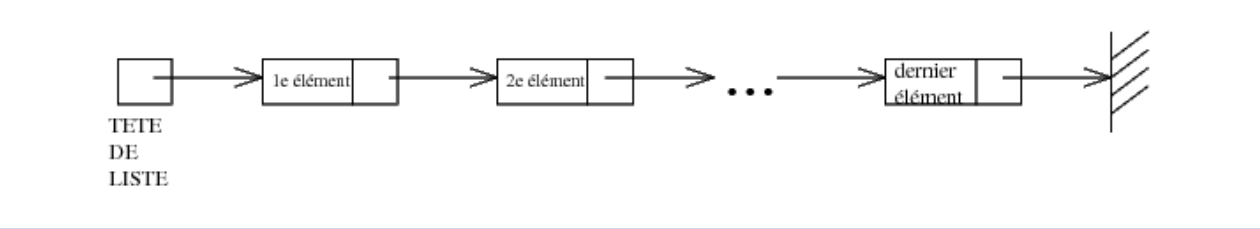
## *Types Abstraits de Données*

**4 TADs** ce semestre :

- **Pile**
- **File**
- **Liste**

## *Types Abstraits de Données : Liste*

**Description** : liste dans laquelle les éléments sont ordonnés. On peut ajouter/supprimer à n'importe quelle position.




Notions importantes :

- *Tête de liste*
- *Flèche* matérialisée par une TAdresse

# Algorithme et Programmation 2

## *Types Abstraits de Données : Liste*

Soit un type **TInfo**, un liste de ce type :

Type TListe = Liste de Tinfo

var L : TListe

**Primitives :**

```
Action créerListe( S L : TListe );
Fonction adressePremier( E L : TListe ) : TAdresse;
// Retourne NULL si la liste est vide.
Fonction adresseSuivant( E L : TListe, E Adr : TAdresse ) : TAdresse;
Fonction valeurElement( E L : TListe, E Adr : TAdresse ) : TInfo;
Action modifieValeurElement( E L : TListe, E Adr : TAdresse, E Elem : TInfo );
Action insérerEnTête( ES L : TListe, E Elem : TInfo );
Action insérerApres( ES L : TListe, E Elem : TInfo, E Adr : TAdresse );
Action supprimerEnTête( ES L : TListe );
Action supprimerAprès( ES L : TListe, E Adr : TAdresse );
```

# Algorithme et Programmation 2

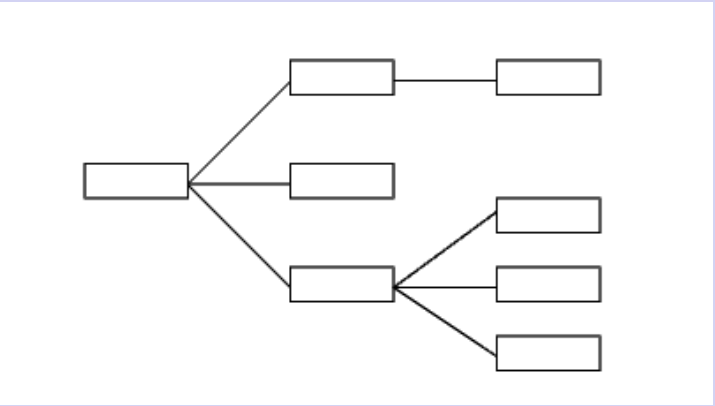
## *Types Abstraits de Données*

**4 TADs** ce semestre :

- **Pile**
- **File**
- **Liste**
- **Arbre (Binaire)**

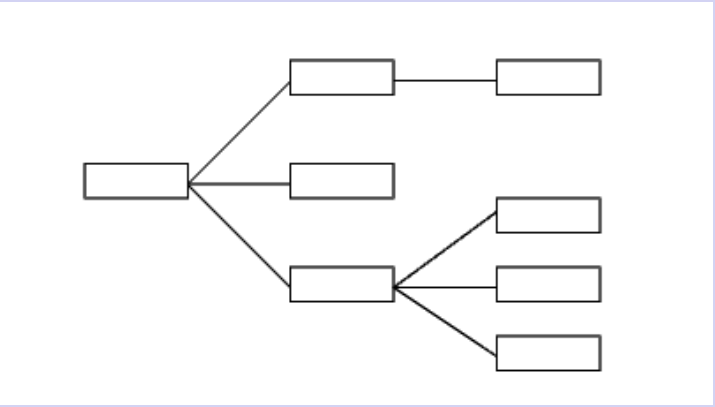
## *Types Abstraits de Données : Arbre*

**Description** : généralisation de la structure Liste : un élément peut avoir plusieurs successeurs.



# Algorithme et Programmation 2

## *Types Abstraits de Données : Arbre*



**sommet** : élément (TInfo) de la structure

**racine** : « premier » élément

**feuille** : élément sans successeur

**fils** : successeur d'une sommet

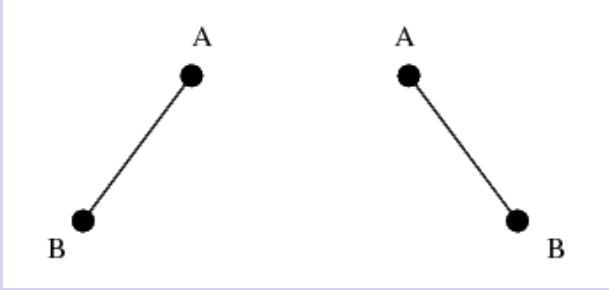
**père** : prédécesseur d'un sommet

**frère** : sommets fils d'un même père

**sous-arbre** : les sous arbres d'un sommets sont les arbres ayant pour racine les fils de ce sommet

## *Types Abstraits de Données : Arbre binaire*

**Description** : arbre dont tout sommet à 0, 1 fils (gauche ou droit) ou 2 fils (gauche et droit).



# Algorithme et Programmation 2

## *Types Abstraits de Données : Arbre Binaire*

Soit un type **TInfo**, un file de ce type :

Type TArbBin = Arbre Binaire de Tinfo

**Primitives :**

```
Création :
  Action créerArbre (S T: TArbBin, E val: TInfo)
       // Crée un arbre dont la racine est val (pas d'arbre vide)
Consultation :
  Fonction adresseRacine (E T: TArbBin): TAdresse
  Fonction adresseFilsGauche (E T: TArbBin, adr: TAdresse): TAdresse
       // Retourne l'adresse du fils gauche de l'élément adr, NULL sinon
  Fonction adresseFilsDroit (E T: TArbBin, adr: TAdresse): TAdresse
  Fonction valeurSommet (E T: TArbBin, adr: TAdresse): TInfo
Edition (minimum...) :
  Action modifierValeurSommet (ES T: TArbBin, E adr: TAdresse, val: TInfo)
  Action insérerFilsGauche (ES T: TArbBin, E adr: TAdresse, val: TInfo )
       // Ajoute un fils gauche, s'il n'existe pas déjà, à l'élément adr
  Action insérerFilsDroit (ES T: TArbBin, E adr: TAdresse, val: TInfo)
```

# Au boulot !