

Classes

Objets

Portée

Accessibilité

Approches Objet de la Programmation

～ Classes et Objets ～

Didier Verna

didier@didierverna.net



didierverna.net



[@didierverna](https://twitter.com/didierverna)



[didier.verna](https://facebook.com/didier.verna)



[in/didierverna](https://in.linkedin.com/didierverna)

Plan

Notion de Classe

- Origine

- Cycle de Vie

Notion d'Objet

- Instanciation

- Cycle de Vie

Portée de l'Information

- Niveau Instance

- Niveau Classe

Accessibilité de l'Information

- Niveaux de Protection

- Notion d'Accesseur

- Amitié

Plan

Notion de Classe

- Origine

- Cycle de Vie

Notion d'Objet

- Instanciation

- Cycle de Vie

Portée de l'Information

- Niveau Instance

- Niveau Classe

Accessibilité de l'Information

- Niveaux de Protection

- Notion d'Accesseur

- Amitié

Plan

Notion de Classe

- Origine

- Cycle de Vie

Notion d'Objet

- Instanciation

- Cycle de Vie

Portée de l'Information

- Niveau Instance

- Niveau Classe

Accessibilité de l'Information

- Niveaux de Protection

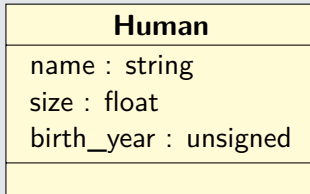
- Notion d'Accesseur

- Amitié

Origine

- ▶ But initial : représenter une famille d'objets similaires dotés de propriétés communes
- ▶ « Records and record classes » [Hoare, 1965]
- ▶ Mention des « union classes » de John McCarthy

UML



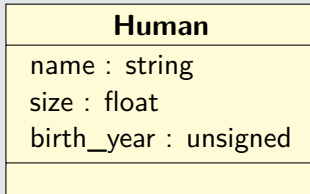
C++

```
class Human
{
    std::string name_;
    float size_;
    unsigned birth_year_;
};
```

Origine

- ▶ But initial : représenter une famille d'objets similaires dotés de propriétés communes
- ▶ « Records and record classes » [Hoare, 1965]
- ▶ Mention des « union classes » de John McCarthy

UML



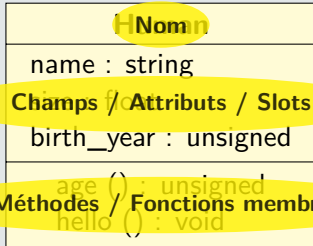
Java

```
class Human
{
    String name;
    float size;
    int birthYear;
}
```

Extension du Modèle de Hoare

- ▶ Les propriétés peuvent inclure du comportement
- ▶ « We needed subclasses of processes with own actions and local data stacks, not only of pure data records. » [Dahl, 1978]

UML



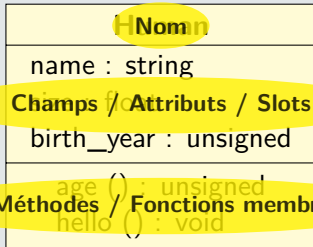
C++

```
class Human
{
    std::string name_;
    float size_;
    unsigned birth_year_;
    unsigned age ();
    void hello ();
};
```

Extension du Modèle de Hoare

- ▶ Les propriétés peuvent inclure du comportement
- ▶ « We needed subclasses of processes with own actions and local data stacks, not only of pure data records. » [Dahl, 1978]

UML



Java

```
class Human
{
    String name;
    float size;
    int birthYear;
    int age () { ... }
    void hello () { ... }
}
```

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

Niveaux de Protection

Notion d'Accesseur

Amitié

Cycle de Vie d'une Classe

- ▶ Statique comme tout autre type de données
fixé et connu à la compilation
- ▶ API d'introspection dans certains langages
p.ex. Java
- ▶ Pas d'intercession

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

Niveaux de Protection

Notion d'Accesseur

Amitié

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

Niveaux de Protection

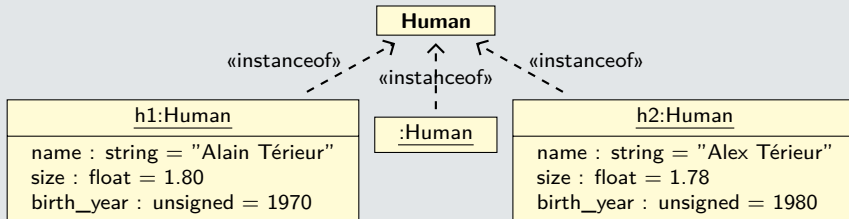
Notion d'Accesseur

Amitié

Instantiation

- ▶ Processus de création d'un objet à partir d'une classe
- ▶ Une classe permet d'instancier plusieurs objets similaires
- ▶ Un objet n'est l'instance que d'une seule classe

UML



Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

Niveaux de Protection

Notion d'Accesseur

Amitié

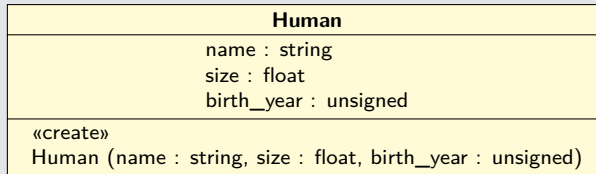
Cycle de Vie d'un Objet

- ▶ Dynamique comme toute autre valeur
créé, utilisé, puis détruit pendant l'exécution
- ▶ *Construction et Destruction*
deux phases très importantes (et formalisées) dans la vie d'un objet
- ▶ Remarque : problème non spécifique à l'orienté-objet
tout type agrégatif est potentiellement concerné

Construction

- ▶ Vision *atomique* de la phase de création d'un objet
 - ▶ Allocation
 - ▶ Initialisation & cohérence interne (*p.ex.* « /attributs » dérivés)
 - ▶ Logique applicative
- ▶ **Constructeur** : pseudo-procédure (pas de valeur de retour)
 - ▶ fourni par défaut mais réimplémentable
 - ▶ nom prédéfini

UML



Construction

- ▶ Vision *atomique* de la phase de création d'un objet
 - ▶ Allocation
 - ▶ Initialisation & cohérence interne (p.ex. « /attributs » dérivés)
 - ▶ Logique applicative

▶ Cor C++

```
class Human
{
    std::string name_;
    float size_;
    unsigned birth_year_;
    Human (const std::string& name, float size, unsigned birth_year);
};
```

name : string
size : float
birth_year : unsigned

«create»

Human (name : string, size : float, birth_year : unsigned)

Construction

- ▶ Vision *atomique* de la phase de création d'un objet
 - ▶ Allocation
 - ▶ Initialisation & cohérence interne (p.ex. « /attributs » dérivés)
 - ▶ Logique applicative
- ▶ **Constructeur** : pseudo-procédure (pas de valeur de retour)
 - ▶ fourni par défaut mais réimplémentable
 - ▶ **C++**

```
Human::Human (const std::string& name, float size, unsigned birth_year)  
: name_ (name), size_ (size), birth_year_ (birth_year)  
{}
```

Human
name : string size : float birth_year : unsigned
«create» Human (name : string, size : float, birth_year : unsigned)

Construction

- ▶ Vision *atomique* de la phase de création d'un objet
 - ▶ Allocation
 - ▶ Initialisation & cohérence interne (p.ex. « /attributs » dérivés)
 - ▶ Logique applicative
- ▶ Constructeurs : pseudo-procédure (pas de valeur de retour)

C++

```
Human h1 = Human ("Alain Térieur", 1.80, 1970);  
Human h2 ("Alex Térieur", 1.78, 1975);  
Human h3 { "Vladimir Guez", 1.83, 1980 };  
auto h4 = Human { "Anne Titgoutte", 1.85, 1985 };  
Human* h5 = new Human ("Corinne Titgoutte", 1.68, 1990);  
auto h6 = std::make_unique<Human> ("Justine Titgoutte", 1.70, 1995);
```

```
name : string  
size : float  
birth_year : unsigned
```

«create»

```
Human (name : string, size : float, birth_year : unsigned)
```

Construction

► Vision *atomique* de la phase de création d'un objet

► Allocation

► Initialisation des chaînes internes (mémoire / attributs dérivés)

► Java

► Cor

```
class Human
{
    String name;
    float size;
    int birthYear;
    Human (String _name, float _size, int _birthYear)
    {
        name = _name;
        size = _size;
        birthYear = _birthYear;
    }
}
```

«create»

Human (name : string, size : float, birth_year : unsigned)

Construction

- ▶ Vision *atomique* de la phase de création d'un objet
 - ▶ Allocation
 - ▶ Initialisation & cohérence interne (*p.ex.* « /attributs » dérivés)
 - ▶ Logique applicative
- ▶ **Constructeur** : pseudo-procédure (pas de valeur de retour)
 - ▶ fourni par défaut mais réimplémentable

Java

```
Human h1 = new Human ("Alain Térieur", 1.80f, 1970);
```

Human

name : string
size : float
birth_year : unsigned

«create»

Human (name : string, size : float, birth_year : unsigned)

Construction

- ▶ Vision *atomique* de la phase de création d'un objet
 - ▶ Allocation
 - ▶ Initialisation & cohérence interne (*p.ex.* « /attributs » dérivés)
 - ▶ Logique applicative
- ▶ **Constructeur** : pseudo-procédure (pas de valeur de retour)
 - ▶ fourni par défaut mais réimplémentable

Remarque : Objective-C

```
Class *object = [[Class alloc] init];
```

```
Class *object = [Class new];
```

Human

name : string
size : float
birth_year : unsigned

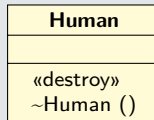
«create»

Human (name : string, size : float, birth_year : unsigned)

Destruction

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Fonctionnalité inverse du constructeur
- ▶ **Destructeur** : langages à gestion manuelle de la mémoire
p.ex. C++ (mais cf. les « smart pointers »)
 - ▶ pseudo-procédure (pas de valeur de retour, pas d'arguments)
 - ▶ fourni par défaut mais réimplémentable
 - ▶ nom prédéfini

UML



Destruction

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Fonctionnalité inverse du constructeur
- ▶ **Destructeur** : langages à gestion manuelle de la mémoire
p.ex. C++ (mais cf. les « smart pointers »)
 - ▶ pseudo-procédure (pas de valeur de retour, pas d'arguments)
 - ▶ fourni par défaut mais réimplémentable
 - ▶ nom prédéfini

C++

```
class Human
{
    ~Human ();
};
```

Destruction

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Fonctionnalité inverse du constructeur
- ▶ **Destructeur** : langages à gestion manuelle de la mémoire
p.ex. C++ (mais cf. les « smart pointers »)
 - ▶ pseudo-procédure (pas de valeur de retour, pas d'arguments)
 - ▶ fourni par défaut mais réimplémentable
 - ▶ nom prédéfini

C++

```
Human : ~Human ()  
{}
```

Destruction

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Fonctionnalité inverse du constructeur
- ▶ **Destructeur** : langages à gestion manuelle de la mémoire
p.ex. C++ (mais cf. les « smart pointers »)
 - ▶ pseudo-procédure (pas de valeur de retour, pas d'arguments)
 - ▶ fourni par défaut mais réimplémentable
 - ▶ nom prédéfini

C++

```
// Called automatically for stack-allocated objects  
// Called by explicit pointer deletion:  
delete h5;  
// Called automatically for smart pointers
```

Destruction (Finalisation)

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Fonctionnalité inverse du constructeur
- ▶ **Finaliseur** : langages à ramasse-miettes
p.ex. Java (`finalize()` ≤ 8 , `cleaners` ≥ 9), peu fiable
 - ▶ Convention : *p.ex. `die()`, `close()`, `dispose()`, `release()` etc.*

Java (don't try this at home !)

```
class Human
{
    void finalize () { ... }
}
```

Destruction (Finalisation)

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Fonctionnalité inverse du constructeur
- ▶ **Finaliseur** : langages à ramasse-miettes
p.ex. Java (`finalize()` ≤ 8 , `cleaners` ≥ 9), peu fiable
 - ▶ Convention : *p.ex. `die()`, `close()`, `dispose()`, `release()` etc.*

Java (don't try this at home !)

```
// Called automatically by the garbage collector  
h1 = null;  
System.gc ();
```

Destruction (La Totale !)

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Fonctionnalité inverse du constructeur
- ▶ **Objective-C**
 - ▶ MRC (Manual Reference Counting) : historique
 - ▶ ARC (Automatic Reference Counting ; proche des smart pointers) : Xcode 4.2
 - ▶ GC : Objective-C 2.0

Objective-C MRC

```
[object retain]; // refcnt += 1  
[object release]; // refcnt -= 1  
[object autorelease]; // move to the autorelease pool
```

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

Niveaux de Protection

Notion d'Accesseur

Amitié

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

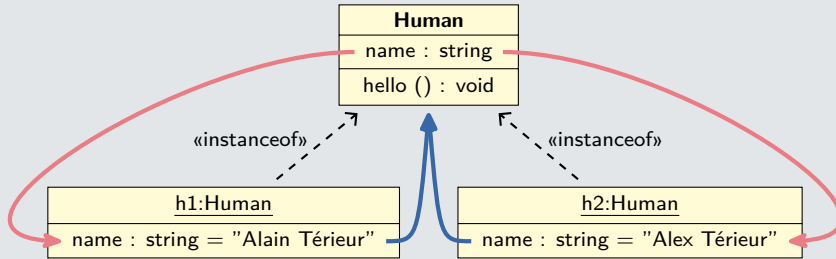
Niveaux de Protection

Notion d'Accesseur

Amitié

Cas Général : Attributs d'Instance

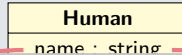
UML



- ▶ Attributs : une déclaration par classe, une valeur par objet
- ▶ Méthodes : une définition par classe (« single dispatch »)
mais accès au contenu spécifique à chaque objet

Cas Général : Attributs d'Instance

UML



C++

```
void Human::hello ()
{
    std::cout << "Hello! I'm " << name_ << ", "
              << size_ << "m, "
              << age () << "yo.\n";
}
```

- ▶ Attributs : un `h1.hello ()`;
`h5->hello ()`;
- ▶ Méthodes : une définition par classe (« single dispatch »)
mais accès au contenu spécifique à chaque objet

Cas Général : Attributs d'Instance

UML

Human

Java

```
class Human
{
    void hello ()
    {
        System.out.println ("Hello! I'm " + name + ", "
                             + size + "m, "
                             + age () + "yo.");
    }
}
```

► Attributs : un }

► Méthodes : u h1.hello ();

mais accès au contenu spécifique à chaque objet

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

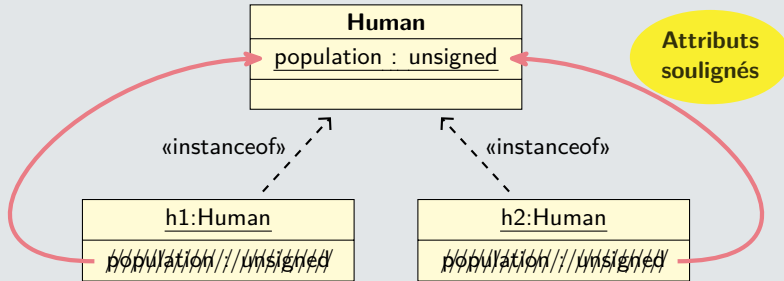
Niveaux de Protection

Notion d'Accesseur

Amitié

Attributs de Classe

UML



- ▶ Attributs définis dans la classe et communs à tous les objets
- ▶ Accès ne nécessitant donc pas l'existence d'une instance
- ▶ `static` en C++ ou Java

Attributs de Classe

UN C++

```
class Human
{
    static unsigned population_;

    // Access via objects: h1.population_ / h5->population_
    unsigned Human::population_ = 0;

    Human::Human (const std::string& name, float size, unsigned birth_year)
    {
        population_ += 1;
    }

    Human::~Human ()
    {
        population_ -= 1;
    }
}
```

▶ Atti

▶ Acc

▶ sta

Attributs de Classe

UN Java

```
class Human
{
    // Access via the class: Human.population
    // Access via objects: h1.population
    static int population = 0;

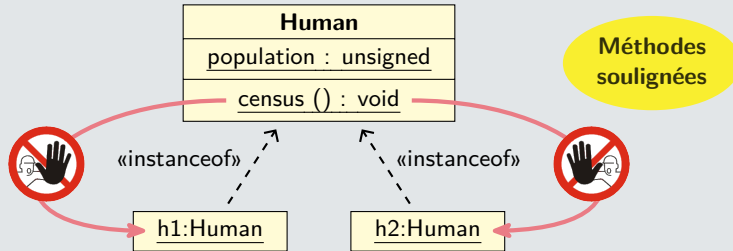
    Human (String _name, float _size, int _birthYear)
    {
        population += 1;
    }

    // Remember: don't try this at home!
    void finalize ()
    {
        population -= 1;
    }
}
```

- ▶ Atti
- ▶ Acc
- ▶ sta

Méthodes de Classe

UML



- ▶ Méthodes n'accédant qu'à des attributs de classe
- ▶ Accès ne nécessitant donc pas l'existence d'une instance
- ▶ `static` en C++ ou Java

Méthodes de Classe

UM C++

```
class Human
{
    static void census ();
};

void Human::census ()
{
    if (population_)
        std::cout << population_ << " human"
                    << (population_ > 1 ? "s " : " ")
                    << "currently alive.\n";
    else
        std::cout << "No human currently alive.\n";
}

static Human::census ();
h1.census ();
h5->census ();
```

- ▶ Méthode
- ▶ Accès ne
- ▶ static

Méthodes de Classe

UML

Java

```
class Human
{
    static void census ()
    {
        if (population != 0)
            System.out.println (population
                + " human" + ((population > 1) ? "s " : " ")
                + "currently alive.");
        else
            System.out.println ("No human currently alive.");
    }
}

Human.census ();
h1.census ();
```

► Méthode

► Accès ne

► static

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

Niveaux de Protection

Notion d'Accesseur

Amitié

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

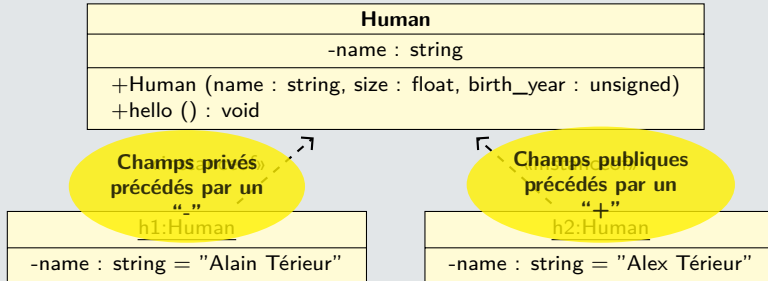
Niveaux de Protection

Notion d'Accesseur

Amitié

Niveaux de Protection

UML



- ▶ **Attribut / Méthode Privé(e)** : accès restreint à la classe
- ▶ **Attribut / Méthode Publique** : accès libre

Niveaux de Protection

UN C++

```
class Human
{
public:
    static void census ();

    Human (const std::string& name, float size, unsigned birth_year);
    ~Human ();

    unsigned age ();
    void hello ();

private:
    static unsigned population_;

    std::string name_;
    float size_;
    unsigned birth_year_;
};
```

► Attribut

► Attribut

Niveaux de Protection

UM' Java

```
public class Human
{
    public static void census () { ... }

    public Human (String _name, float _size, int _birthYear) { ... }
    public void finalize () { ... }

    public int age () { ... }
    public void hello () { ... }

    private static int population = 0;

    private String name;
    private float size;
    private int birthYear;
}
```

► Attribut

► Attribut

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

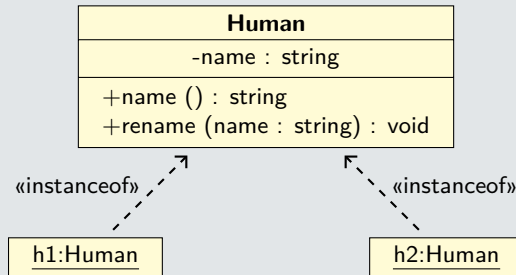
Niveaux de Protection

Notion d'Accesseur

Amitié

Notion d'Accesseur

UML



- ▶ **Accesseurs** : getter / setter
 - ▶ consultation / modification des attributs privés
- ▶ **Interface** : ensemble de l'information publique

Notion d'Accesneur

UML

Human

C++

```
class Human
{
public:
    const std::string& name () const;
    void rename (const std::string& name);

private:
    std::string name_;
    const float size_;
    const unsigned birth_year_;
};
```

► Accesseurs :

► consultat

► Interface : ensemble de l'information publique

Notion d'Accesseur

UML

Human

C++

```
const std::string& Human::name () const
{
    return name_;
}

void Human::rename (const std::string& name)
{
    // Maybe check with the administration first ;-)
    name_ = name;
}
```

- ▶ **Accesseurs :**
 - ▶ consultation / modification des attributs privés
- ▶ **Interface :** ensemble de l'information publique

Notion d'Accesseur

UML

Java

```
public class Human
{
    public String name ()
    {
        return name;
    }
    public void rename (String _name)
    {
        name = _name;
    }
}
```

► Accesseurs :

► consultat

► Interface : e }

```
private String name;
private final float size;
private final int birthYear;
```

Plan

Notion de Classe

Origine

Cycle de Vie

Notion d'Objet

Instanciation

Cycle de Vie

Portée de l'Information

Niveau Instance

Niveau Classe

Accessibilité de l'Information

Niveaux de Protection

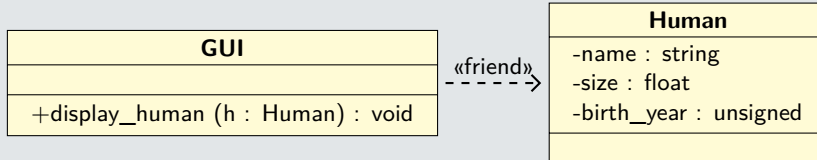
Notion d'Accesseur

Amitié

Amitié

- ▶ Principe d'exception au régime de privatisation
- ▶ Accès privilégié depuis l'extérieur autorisé au cas par cas

Exemple UML



- ▶ Concept d'amitié variable selon les langages
- ▶ Politiques de protection par défaut également

Amitié

- ▶ Principe d'exception au régime de privatisation
- ▶ Accès privilégié depuis l'extérieur autorisé au cas par cas

C++

Exe

```
class Human
{
    friend void birth_control (const Human& human);
};

void birth_control (const Human& human)
{
    std::cout << "The NSA knows about " << human.name_ << "...\\n";
}
```

- ▶ Concept `birth_control (h1);`
- ▶ Politiques de protection par défaut également `birth_control (*h5);`



Bibliographie

Plan

Bibliographie

Bibliographie

-  C.A.R. (Tony) Hoare.
Record Handling.
Algol Bulletin n° 21, 1965.
-  Ole-Johan Dahl and Kristen Nygaard.
The Development of the SIMULA Languages.
History of Programming Languages Conference, 1978.