



Classes



Objets



Portée



Accessibilité

# Approches Objet de la Programmation

## ～ Classes et Objets ～

Didier Verna  
EPITA / LRDE

[didier@lrde.epita.fr](mailto:didier@lrde.epita.fr)



[lrde/~didier](#)



[@didierverna](#)



[didier.verna](#)



[google+](#)



[in/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*

*Portée de l'Information*

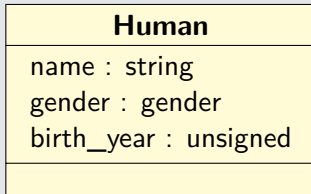
*Accessibilité de l'Information*



# Origine

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

## UML



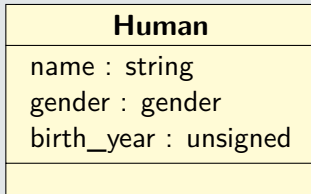
## C++

```
class Human
{
    std::string name_;
    gender gender_;
    unsigned birth_year_;
};
```

# Origine

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

## UML



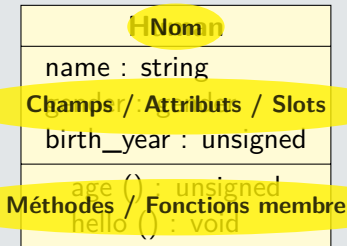
## Java

```
class Human
{
    String name;
    Gender gender;
    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



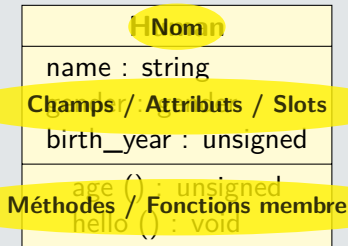
## C++

```
class Human
{
    std::string name_;
    gender gender_;
    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;
    Gender gender;
    int birthYear;
    int age () { ... }
    void hello () { ... }
}
```

# 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

Notion d'Objet

  Instanciation

  Cycle de Vie

Portée de l'Information

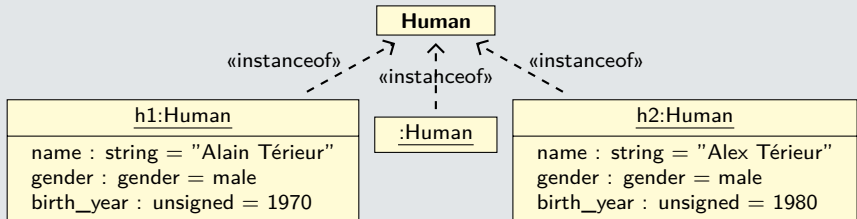
Accessibilité de l'Information



# 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



# Cycle de Vie d'un Objet

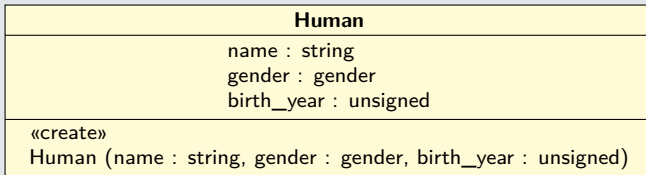
- ▶ Dynamique comme toute autre valeur  
*créé, utilisé, puis détruit pendant l'exécution*
- ▶ *Construction et Destruction*  
*deux phases très importantes (et formalisées) dans la vie d'un objet*
- ▶ Remarque : problème non spécifique à l'orienté-objet  
*tout type agrégatif est potentiellement concerné*



# Construction

- ▶ Vision *atomique* de la phase de création d'un objet
- ▶ Assure sa cohérence interne (aspects agrégatifs)
- ▶ **Constructeur** : procédure spéciale (pas de type de retour)
  - ▶ fourni par défaut (mais modifiable)
  - ▶ nom prédéfini
  - ▶ allocation / initialisation (séparation parfois explicite)

## UML



# Construction

- ▶ Vision *atomique* de la phase de création d'un objet
- ▶ Assure sa cohérence interne (aspects agrégatifs)
- ▶ **Constructeur** : procédure spéciale (pas de type de retour)
  - ▶ fourni par défaut (mais modifiable)
  - ▶ nom prédéfini
  - ▶ allocation / initialisation (séparation parfois explicite)

## C++

```
class Human
{
    std::string name_;
    gender gender_;
    unsigned birth_year_;
    Human (const std::string& name, gender g, unsigned birth_year);
};
```



# Construction

- ▶ Vision *atomique* de la phase de création d'un objet
- ▶ Assure sa cohérence interne (aspects agrégatifs)
- ▶ **Constructeur** : procédure spéciale (pas de type de retour)
  - ▶ fourni par défaut (mais modifiable)
  - ▶ nom prédéfini
  - ▶ allocation / initialisation (séparation parfois explicite)

## C++

```
Human::Human (const std::string& name, gender g, unsigned birth_year)
: name_ (name), gender_ (g), birth_year_ (birth_year)
{}
```



# Construction

- ▶ Vision *atomique* de la phase de création d'un objet
- ▶ Assure sa cohérence interne (aspects agrégatifs)
- ▶ **Constructeur** : procédure spéciale (pas de type de retour)
  - ▶ fourni par défaut (mais modifiable)
  - ▶ nom prédéfini
  - ▶ allocation / initialisation (séparation parfois explicite)

## C++

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



# Construction

- ▶ Vision *atomique* de la phase de création d'un objet
- ▶ Assure sa cohérence interne (aspects agrégatifs)

## ▶ Con: Java

```
class Human
{
    String name;
    Gender gender;
    int birthYear;
    Human (String _name, Gender _gender, int _birthYear)
    {
        name = _name;
        gender = _gender;
        birthYear = _birthYear;
    }
}
```





# Construction

- ▶ Vision *atomique* de la phase de création d'un objet
- ▶ Assure sa cohérence interne (aspects agrégatifs)
- ▶ **Constructeur** : procédure spéciale (pas de type de retour)
  - ▶ fourni par défaut (mais modifiable)
  - ▶ nom prédéfini
  - ▶ allocation / initialisation (séparation parfois explicite)

## Java

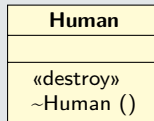
```
Human h1 = new Human ("Alain Térieur", Gender.male, 1970);
```



# Destruction

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Assure le "nettoyage"
- ▶ **Destructeur** : langages à gestion manuelle de la mémoire  
*p.ex. C++ (mais cf. les « smart pointers »)*
  - ▶ procédure spéciale (pas de type de retour ni d'argument)
  - ▶ fourni par défaut (mais modifiable)
  - ▶ nom prédéfini

## UML



# Destruction

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Assure le "nettoyage"
- ▶ **Destructeur** : langages à gestion manuelle de la mémoire  
*p.ex. C++ (mais cf. les « smart pointers »)*
  - ▶ procédure spéciale (pas de type de retour ni d'argument)
  - ▶ fourni par défaut (mais modifiable)
  - ▶ nom prédéfini

## C++

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



# Destruction

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Assure le "nettoyage"
- ▶ **Destructeur** : langages à gestion manuelle de la mémoire  
*p.ex. C++ (mais cf. les « smart pointers »)*
  - ▶ procédure spéciale (pas de type de retour ni d'argument)
  - ▶ fourni par défaut (mais modifiable)
  - ▶ nom prédéfini

## C++

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



# Destruction

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Assure le "nettoyage"
- ▶ **Destructeur** : langages à gestion manuelle de la mémoire  
*p.ex. C++ (mais cf. les « smart pointers »)*
  - ▶ procédure spéciale (pas de type de retour ni d'argument)
  - ▶ fourni par défaut (mais modifiable)
  - ▶ 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
- ▶ Assure le "nettoyage"
- ▶ **Finaliseur** : langages à ramasse-miettes  
*p.ex. Java avec `finalize()`, mais peu fiable*
  - ▶ Convention : *p.ex. `die()`, `close()`, `dispose()`, `release()` etc.*

## Java

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



# Destruction (Finalisation)

- ▶ Vision *atomique* de la phase de disparition d'un objet
- ▶ Assure le “nettoyage”
- ▶ **Finaliseur** : langages à ramasse-miettes  
*p.ex. Java avec `finalize()`, mais peu fiable*
  - ▶ Convention : *p.ex. `die()`, `close()`, `dispose()`, `release()` etc.*

## Java

```
// Called automatically by the garbage collector  
// Don't do this for real!  
h1 = null;  
System.gc ();
```



# Plan

Notion de Classe

Notion d'Objet

Portée de l'Information

- Niveau Instance

- Niveau Classe

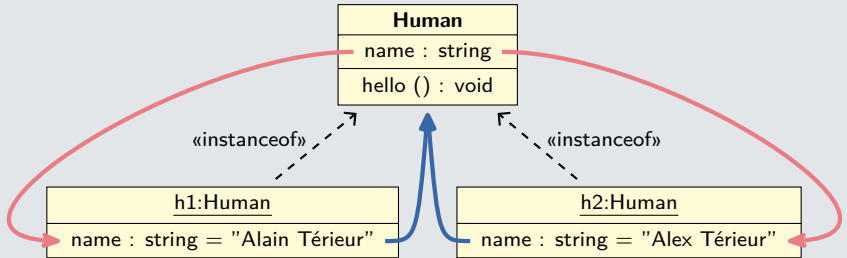
Accessibilité de l'Information





# Cas Général : Attributs d'Instance

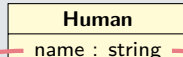
## UML



- ▶ 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_ << ", "
               << gender_ << ", "
               << age () << ".\n";
}
```

A red arrow points from the '}' character to the object creation code below.

```
h1.hello ();
```

► A h5->hello ();

- Méthodes : une définition par classe (« single dispatch »)  
*mais accès au contenu spécifique à chaque objet*

# Cas Général : Attributs d'Instance

## UML

Human

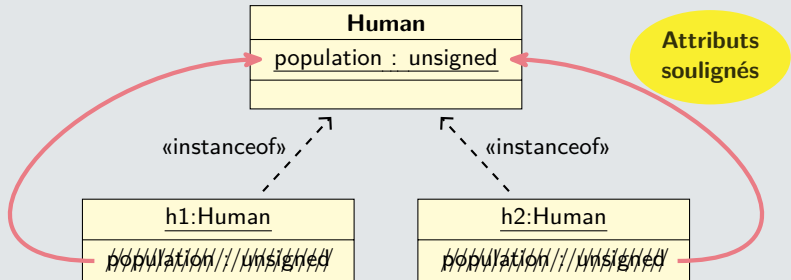
### Java

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

- ▶ A
- ▶ h1.hello ();
- ▶ Méthodes : une définition par classe (« single dispatch »)  
*mais accès au contenu spécifique à chaque objet*

# 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

## U! C++

```
class Human
{
    static unsigned population_;
};

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

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

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



# Attributs de Classe

## UMI Java

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

    Human (String _name, Gender _gender, int _birthYear)
    {
        population += 1;
    }

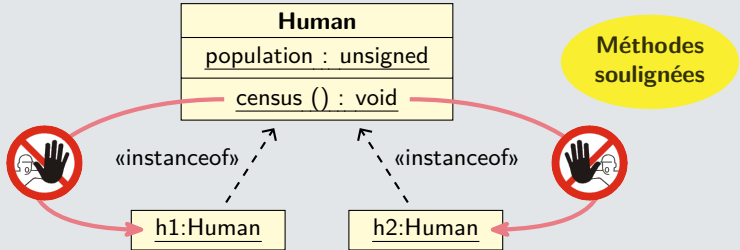
    void finalize ()
    {
        population -= 1;
    }
}
```

► static en C++ ou Java



# 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

UML

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";
}

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

- Méth
- Accès
- stat

Méthodes  
signées



# 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.");
    }
}
```

► Méthode

► Accès

► stat.

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

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

Méthodes  
signées



# Plan

Notion de Classe

Notion d'Objet

Portée de l'Information

Accessibilité de l'Information

- Niveaux de Protection

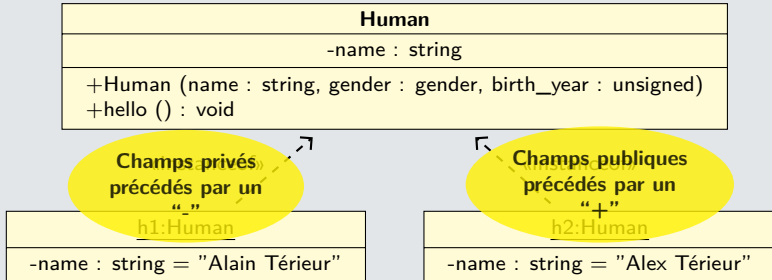
- Notion d'Accesseur

- Amitié



# Niveaux de Protection

## UML



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

# Niveaux de Protection

## UI C++

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

    Human (const std::string& name, gender g, unsigned birth_year);
    ~Human ();

    unsigned age ();
    void hello ();

private:
    static unsigned population_;

    std::string name_;
    gender gender_;
    unsigned birth_year_;
};
```



# Niveaux de Protection

UM  
Java

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

    public Human (String _name, Gender _gender, int _birthYear) { ... }
    public void finalize () { ... }

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

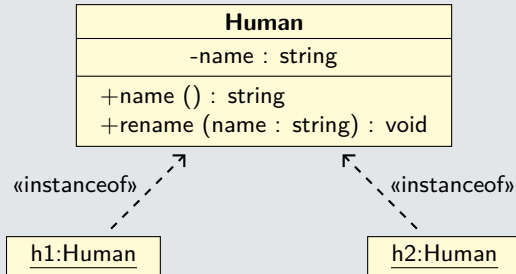
    private static int population = 0;

    private String name;
    private Gender gender;
    private int birthYear;
}
```



# Notion d'Accesseur

## UML



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

# Notion d'Accesseur

## UML

Human

C++

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

private:
    std::string name_;
    const gender gender_;
    const unsigned birth_year_;
};
```

- ▶ **Accesseur**
  - ▶ consultation / modification des attributs privés
- ▶ **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;
}
```

### ► Accesseur

- 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;
    }
}
```

#### ► Accesseur

##### ► (

```
private String name;
private final Gender gender;
private final int birthYear;
```

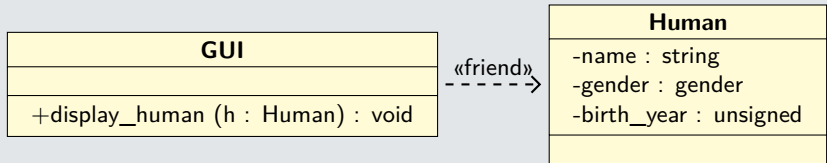
#### ► Interface : ensemble de l'information publique



# Amitié

- ▶ Principe d'exception au régime de privatisation
- ▶ Accès privilégié depuis l'extérieur autorisé au cas pas 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";
}
```

- ▶ birth\_control (h1);
- ▶ birth\_control (\*h5);
- ▶ Politiques de protection par défaut également

# 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.*