
# Object-Oriented Approaches to Programming

## Classes and Objects

Didier Verna

EPITA / LRDE

didier@lrde.epita.fr

lrde/~didier


@didierverna


didier.verna


in/didierverna

# Outline



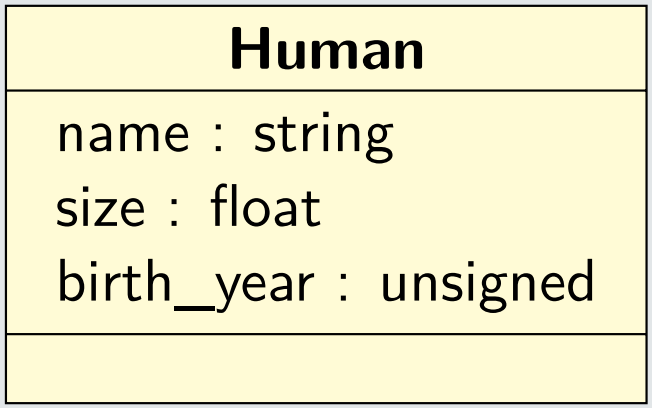
# Plan



# Origin

- Initial goal: represent a family of similar objects, having common properties
- “Records and record classes” [Hoare, 1965]
- Mention of John McCarthy’s “union classes”

## UML



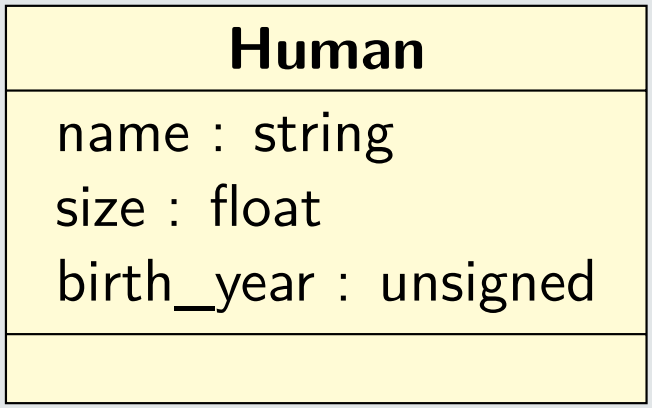

## C++

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

# Origin

- Initial goal: represent a family of similar objects, having common properties
- “Records and record classes” [Hoare, 1965]
- Mention of John McCarthy’s “union classes”

## UML



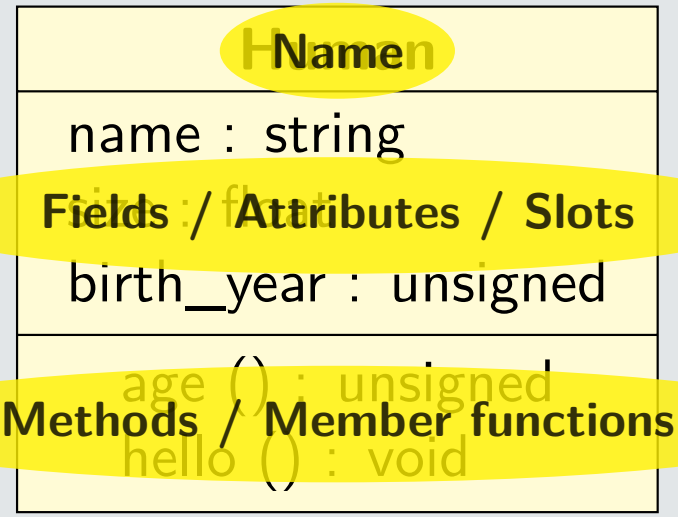

## Java

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

# Extension to the Hoare Model

- Properties may include behavior
- "We needed subclasses of processes with own actions and local data stacks, not only of pure data records." [Dahl, 1978]

## UML



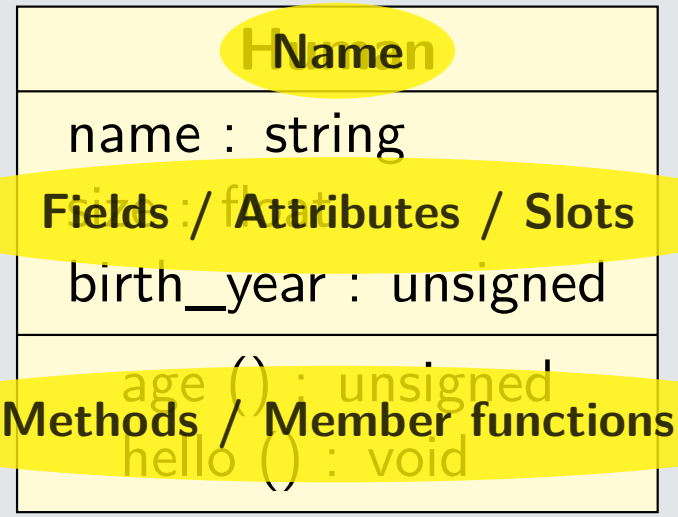

## C++

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

# Extension to the Hoare Model

- Properties may include behavior
- "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 () { ... }
}
```

# Class Life Cycle

- Static, like any other data type
  *fixed and known at compile-time*
- Introspection API in some languages
  *e.g. Java*
- No intercession

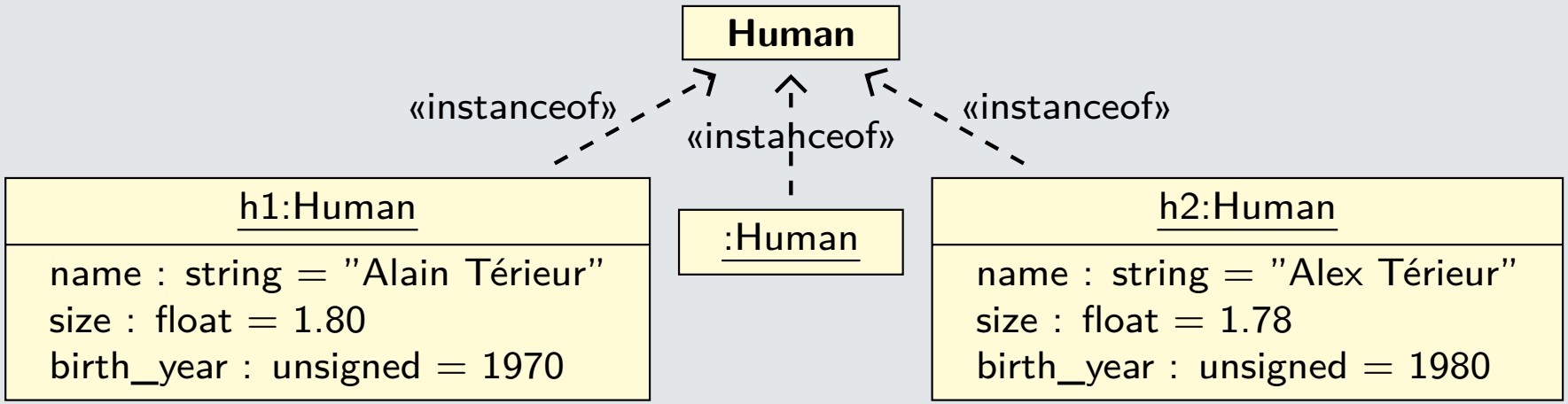
# Plan



# Instantiation

- Creation of an object based on a class
- Many objects may be instantiated from a class
- An object is an instance of a single class

## UML



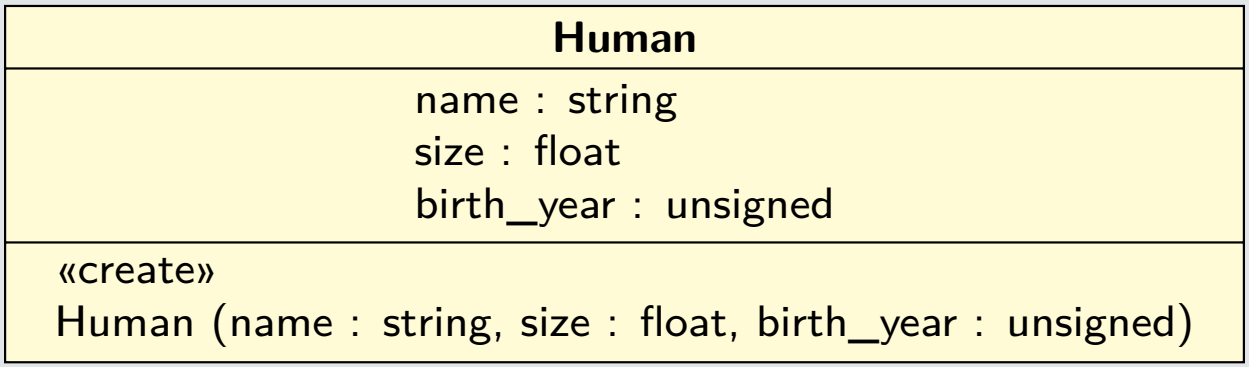
# Object Life Cycle

- Dynamic, like any other value
  *Created, used, and destroyed at run-time*
- *Construction* and *Destruction*
  *Two very important (and formalized) phases of an object's life cycle*
- Remark: problem orthogonal to object-orientation
  *Any aggregative type is potentially concerned*

# Construction

- *Atomic* view on an object's creation phase
  - Allocation
  - Initialization & internal coherence (*e.g.* derived "/attributes")
  - Application logic
- **Constructor:** pseudo-procedure (no return value)
  - provided by default but reimplementable
  - predefined name

## UML



# Construction

- *Atomic* view on an object's creation phase
  - Allocation
  - Initialization & internal coherence (*e.g.* derived "/attributes")
  - Application logic
- **Constructor:** pseudo-procedure (no return value)
  - provided by default but reimplementable
  - predefined name

## C++

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

# Construction

- *Atomic* view on an object's creation phase
  - Allocation
  - Initialization & internal coherence (*e.g.* derived "/attributes")
  - Application logic
- **Constructor:** pseudo-procedure (no return value)
  - provided by default but reimplementable
  - predefined name

## C++

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

# Construction

- *Atomic* view on an object's creation phase
  - Allocation
  - Initialization & internal coherence (*e.g.* derived "/attributes")
  - Application logic
- **Constructor:** pseudo-procedure (no return value)
  - provided by default but reimplementable
  - predefined name

## 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);
```

# Construction

- *Atomic* view on an object's creation phase
  - Allocation
  - [illegible]s")
  - [illegible]
- Con[illegible]
  - [illegible]
  - [illegible]

**Java**

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

# Construction

- *Atomic* view on an object's creation phase
  - Allocation
  - Initialization & internal coherence (*e.g.* derived "/attributes")
  - Application logic
- **Constructor:** pseudo-procedure (no return value)
  - provided by default but reimplementable
  - predefined name

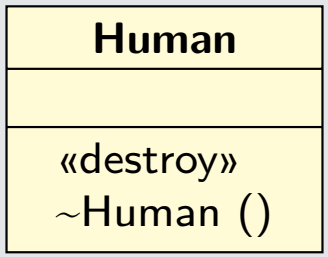
**Java**

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

# Destruction

- *Atomic* view on an object's removal phase
- Constructor's inverse functionality
- **Destructor:** manual memory management languages
  *e.g. C++ (but see "smart pointers")*
  - pseudo-procedure (no return value, no arguments)
  - provided by default but reimplementable
  - predefined name

## UML



# Destruction

- *Atomic* view on an object's removal phase
- Constructor's inverse functionality
- **Destructor:** manual memory management languages
  *e.g. C++ (but see "smart pointers")*
  - pseudo-procedure (no return value, no arguments)
  - provided by default but reimplementable
  - predefined name

## C++

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

# Destruction

- *Atomic* view on an object's removal phase
- Constructor's inverse functionality
- **Destructor:** manual memory management languages
  *e.g. C++ (but see "smart pointers")*
  - pseudo-procedure (no return value, no arguments)
  - provided by default but reimplementable
  - predefined name

**C++**

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

# Destruction

- *Atomic* view on an object's removal phase
- Constructor's inverse functionality
- **Destructor:** manual memory management languages
  *e.g. C++ (but see "smart pointers")*
  - pseudo-procedure (no return value, no arguments)
  - provided by default but reimplementable
  - predefined name

**C++**

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

# Destruction (Finalization)

- *Atomic* view on an object's removal phase
- Constructor's inverse functionality
- **Finalizer:** garbage-collected languages
  *e.g. Java (`finalize()` <= 8, cleaners >= 9), unreliable*
  - Convention: *e.g.* `die()`, `close()`, `dispose()`, `release()`, *etc.*

**Java (don't try this at home!)**

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

# Destruction (Finalization)

- *Atomic* view on an object's removal phase
- Constructor's inverse functionality
- **Finalizer:** garbage-collected languages
  *e.g. Java (`finalize()` <= 8, cleaners >= 9), unreliable*
  - Convention: *e.g.* `die()`, `close()`, `dispose()`, `release()`, *etc.*

**Java (don't try this at home!)**

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

# Plan

Concept of Class

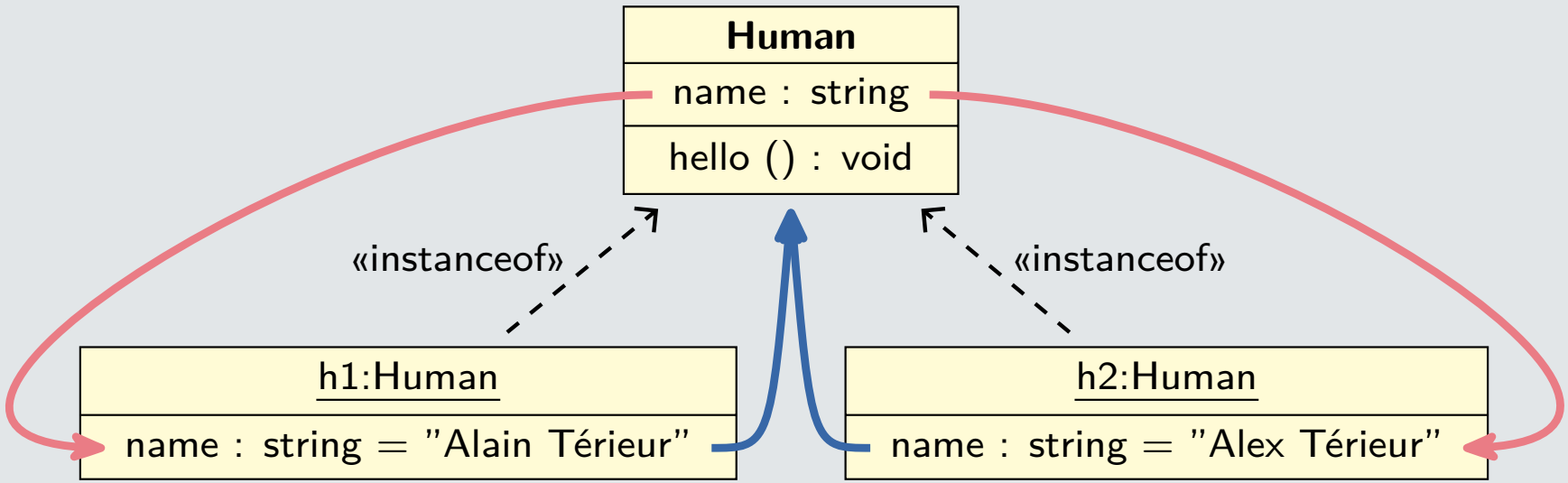
Concept of Object

Information Scope
- Instance Level
- Class Level

Information Accessibility

# General case: Instance Attributes

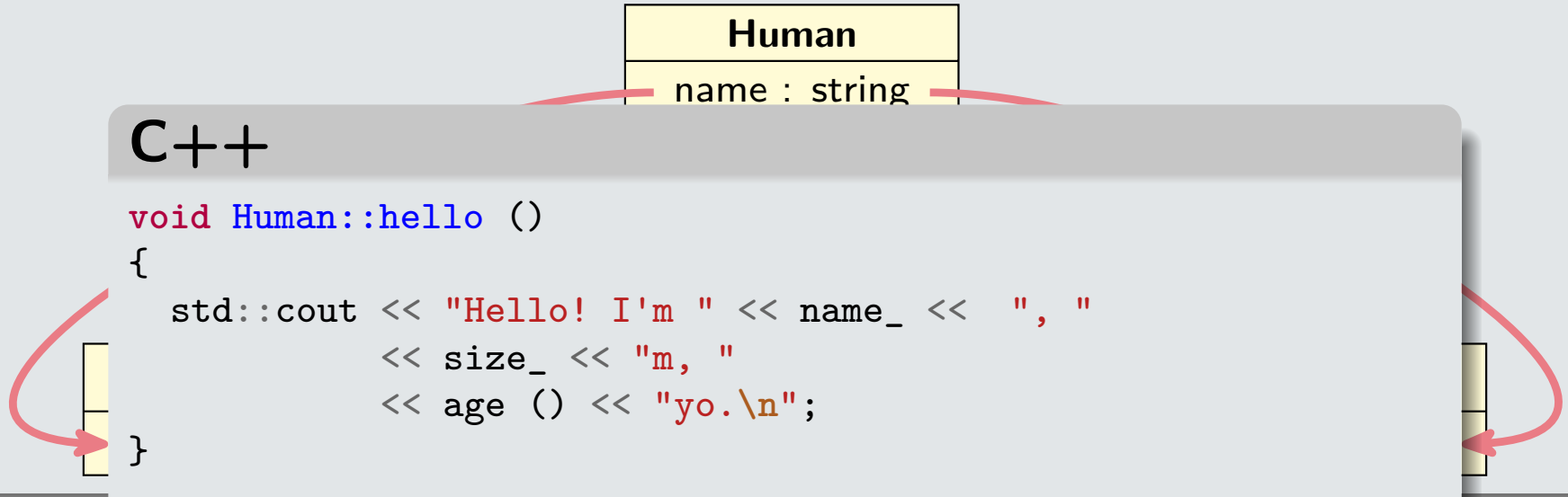
## UML




- Attributes: one declaration per class, one value per object
- Methods: one definition per class ("single dispatch")
  *But access to object-specific attribute values*

# General case: Instance Attributes

## UML




## C++

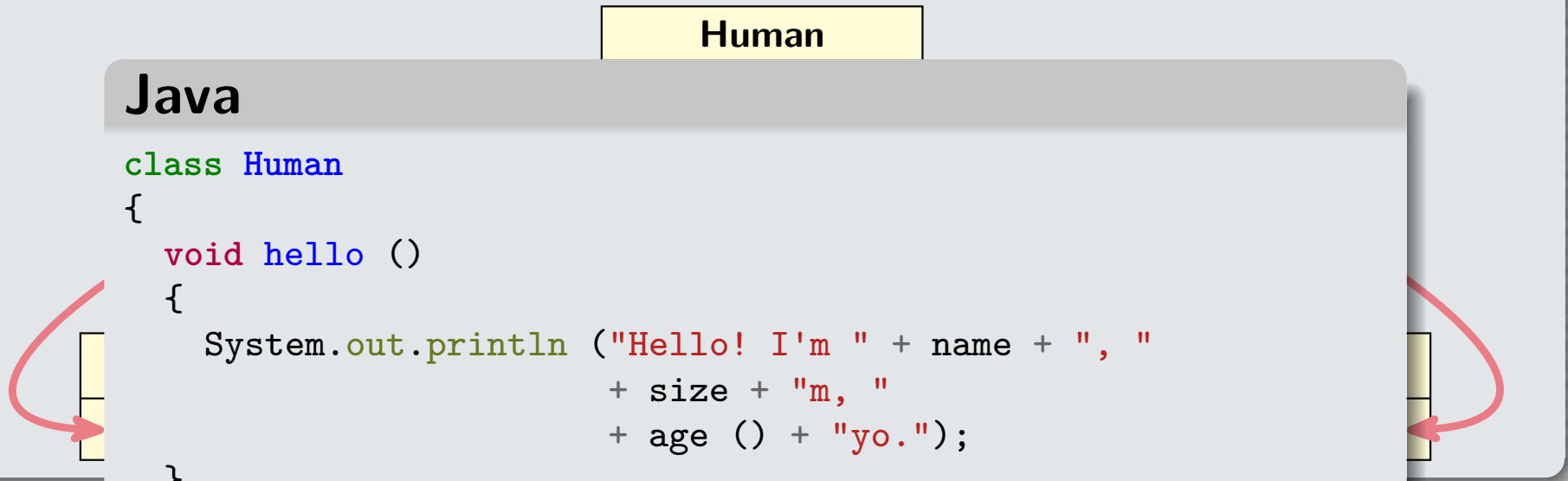
```
void Human::hello ()
{
  std::cout << "Hello! I'm " << name_ <<  ", "
            << size_ << "m, "
            << age () << "yo.\n";
}

h1.hello ();
h5->hello ();
```

- A
- Methods: one definition per class (“single dispatch”)

  *But access to object-specific attribute values*

# General case: Instance Attributes

## UML




## Java

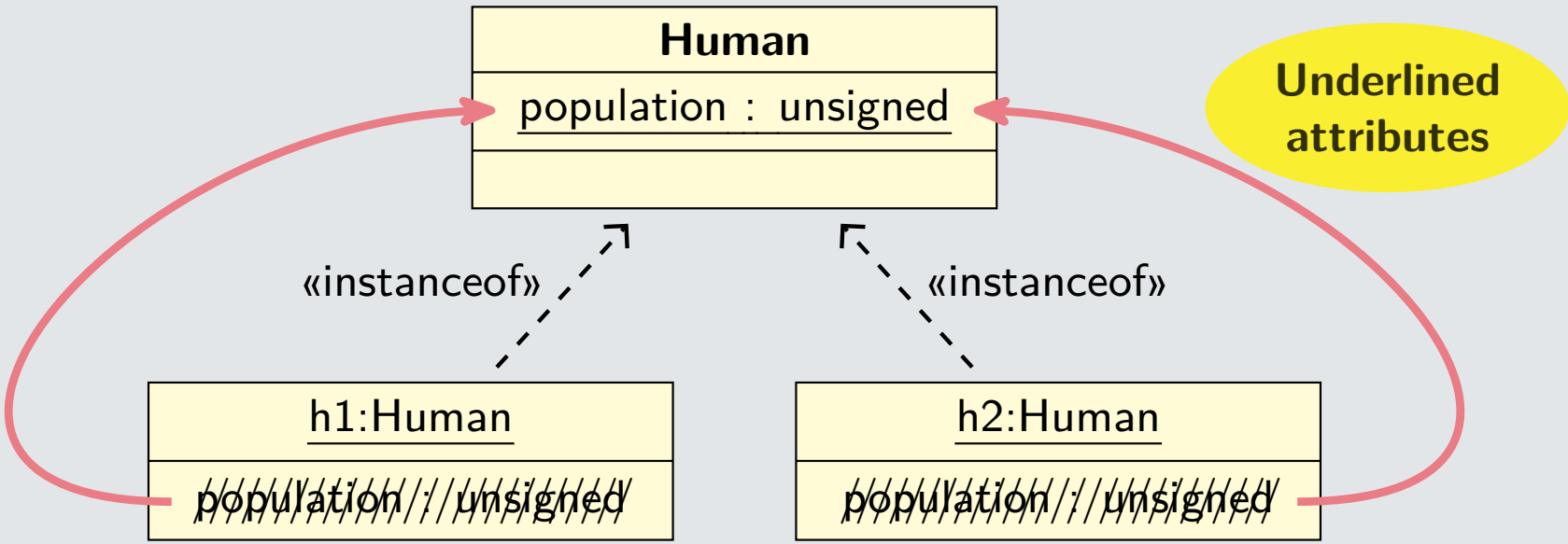
```
class Human
{
  void hello ()
  {
    System.out.println ("Hello! I'm " + name + ", "
                        + size + "m, "
                        + age () + "yo.");
  }
}

h1.hello ();
```

- A
- M

*But access to object-specific attribute values*

# Class Attributes

## UML




- Still declared in a class, but only one value for all objects
- Should be accessible without any instance
- `static` in C++ or Java

# Class Attributes

## 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;
}
```

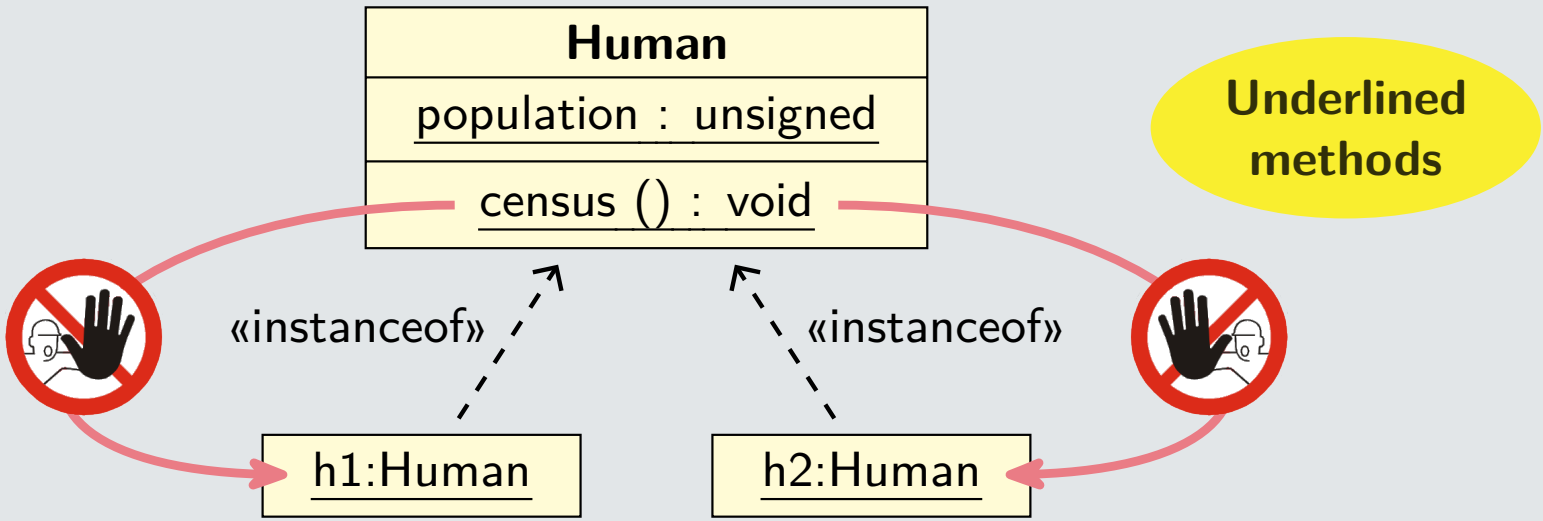
# Class Attributes

## 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;
  }
}
```

# Class Methods

## UML




- Methods accessing class attributes only
- Should be callable without any instance
- `static` in C++ or Java

# Class Methods

## 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 ();
```

lerlined
ethods

- Meth
- Shou
- stat

# Class Methods

## 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 ();
```

lerlined
ethods

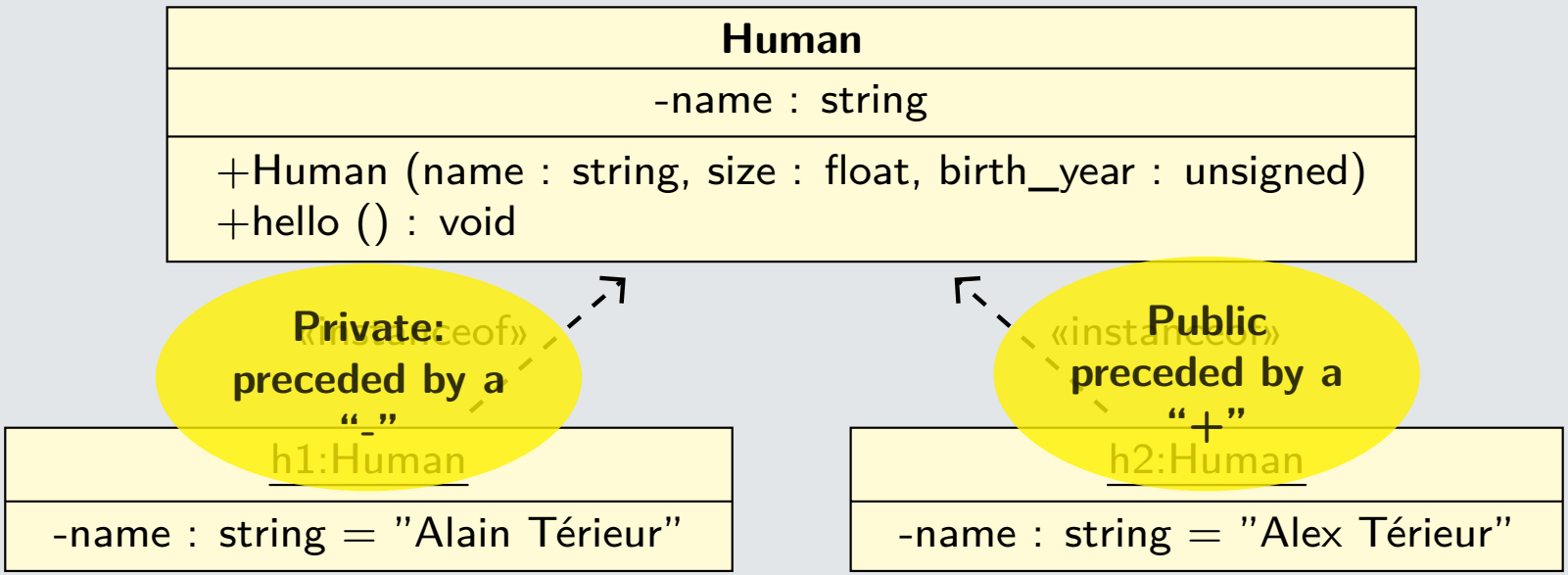
- Meth
- Shou
- stat

# Plan



# Protection Levels

## UML




- **Private Attribute / Method:** access restricted to the class
- **Public Attribute / Method:** open access

# Protection Levels

## 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_;
};
```

# Protection Levels

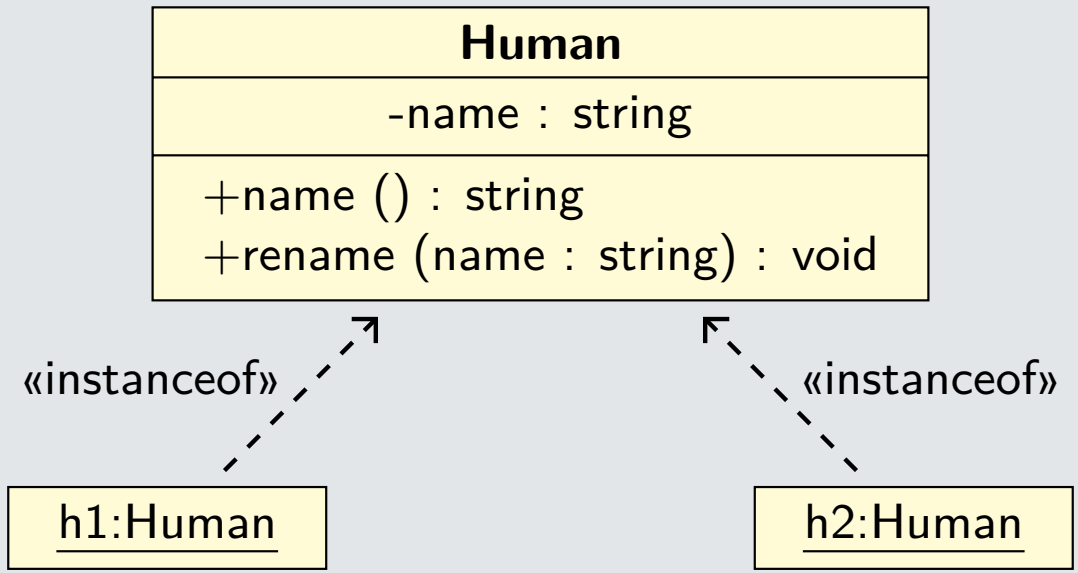
UML

## 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;
}
```

# Concept of Accessor

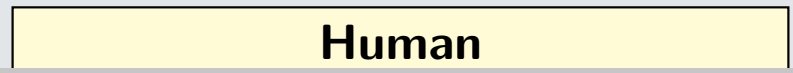
## UML




- **Accessors:** getter / setter
  - reading / writing of private attributes
- **Interface:** set of all public information

# Concept of Accessor

## UML



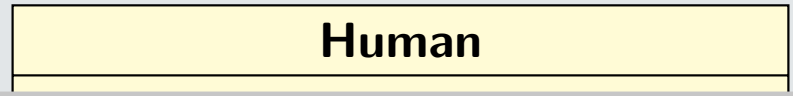

## C++

```
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_;
};
```

- **Acce**
  - reading / writing of private attributes
- **Interface:** set of all public information

# Concept of Accessor

## UML



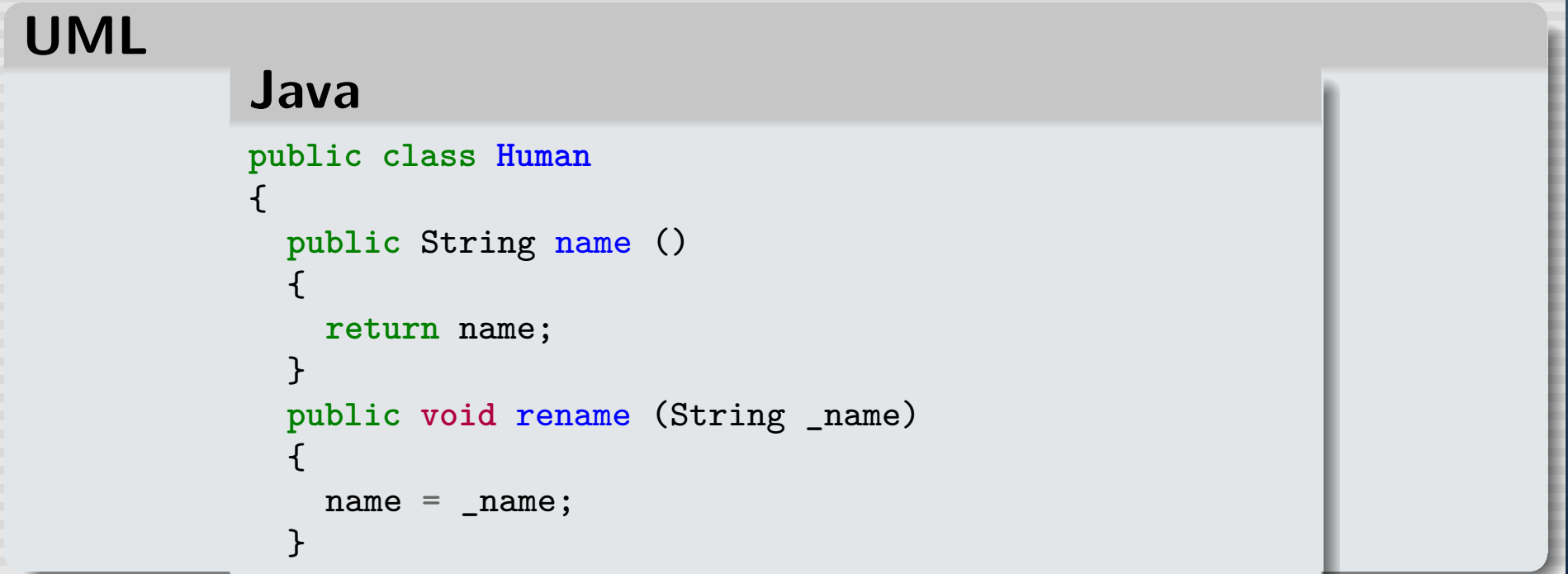

## C++

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

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

- **Acce**
  - reading / writing of private attributes
- **Interface:** set of all public information

# Concept of Accessor

## UML



## Java

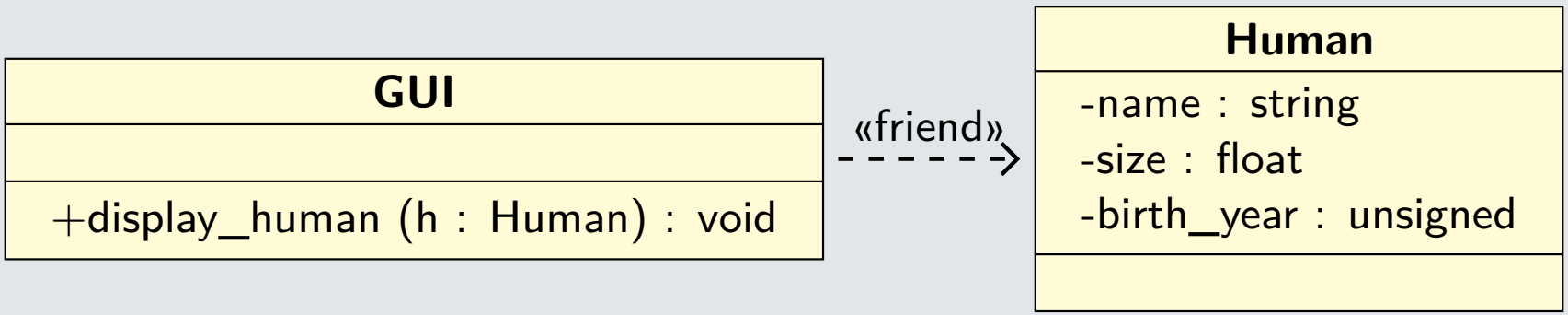
```
public class Human
{
  public String name ()
  {
    return name;
  }
  public void rename (String _name)
  {
    name = _name;
  }

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

- Acce
  - I
- Inter set of all public information

# Friendship

- Exception facility to work around privatization
- Priviledged access authorized case by case

## UML Example



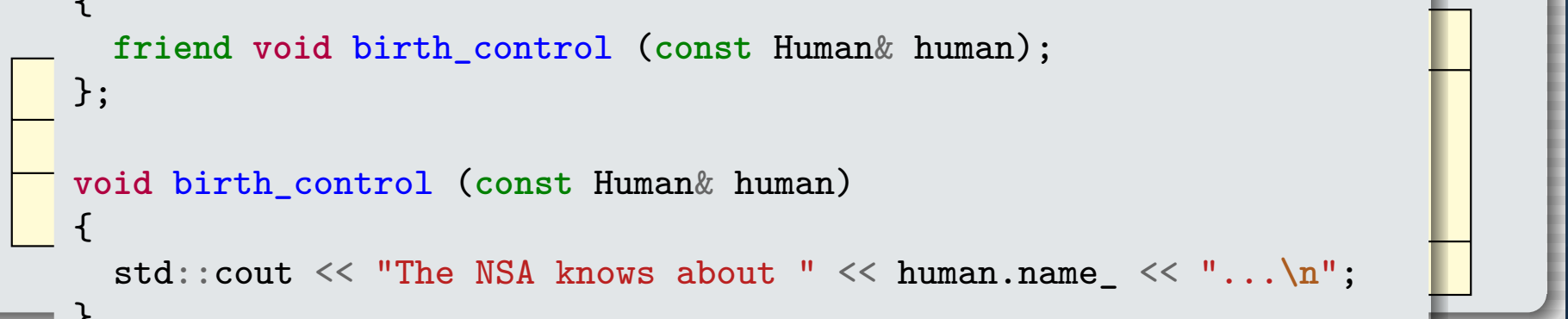

- Not necessarily available in every language / may vary
- Default protection level may vary as well

# Friendship

- Exception facility to work around privatization
- Priviledged access authorized case by case

C++

```
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);
```



# Plan

## Bibliography

# Bibliography


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.