Lecture #2 on Object-Oriented Modeling

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

2006

Outline

Outline

Outline

Outline

Inheritance

Consider that:

- inheritance is a key concept of object-orientation
- a lot of *different* kinds of inheritance exist

  with various meanings/semantics (read Meyer for details)
- the most prominent form is:

  “class inheritance” related to “sub-typing”

  mapping the “is a” relationship
- “polymorphism of methods” and “abstract data types”
  - are also key concepts of object-orientation
  - and rely on inheritance

see also:
`http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)`

Inheritance and Polymorphism

Two points of view

- type theory:
  - it defines the notion of “sub-typing”
  - and “sub-classing” reflects (tries to reflect) this notion
- semantics:
  - verifying “is a” is mandatory to get class inheritance
  - however it is not enough to get proper inheritance
  - having various semantics explain the different kinds of inheritance

Outline

UML?

The **U**nified **M**odeling **L**anguage:

- is an object modeling and specification language both textual and graphical
- is **the** language we should talk and understand
- comes from:
  - James Rumbaugh (OMT)
  - Grady Booch (Booch method)
  - Ivar Jacobson (Objectory)
  - Richard Soley (Object Management Group, OMG)

see also:
`http://en.wikipedia.org/wiki/Unified_Modeling_Language`

UML...

This lecture does not aim at teaching UML!

Resources from the Internet

- the “official” site:
  `http://www.uml.org/`
- from Cetus (the OO portal):
  `http://www.cetus-links.org/oo_uml.html`
- at a glance (one single page):
  Allen Holub’s UML Quick Reference
  `http://www.holub.com/goodies/uml/index.html`

Books on UML

- “The Unified Modeling Language User Guide,” by Grady Booch, James Rumbaugh, and Ivar Jacobson, 2nd ed., Addison-Wesley Professional, 2005.
- “The Unified Modeling Language Reference Manual,” by James Rumbaugh, Ivar Jacobson, and Grady Booch, 2nd ed., Addison-Wesley Professional, 2004.
- “The Complete UML Training Course,” by Grady Booch, James Rumbaugh, and Ivar Jacobson, Prentice Hall, 2000.
- and many more...

Outline

Different Ways of Describing a Program

three different ways

- FUNCTIONAL point of view:
  - what is it doing?
    for instance, its list of main tasks
    or of more “atomic” functionalities...
- STATIC point of view:
  - what’s in it?
    for instance, a list of classes
    and the methods of every class, etc.
- DYNAMIC point of view:
  - what happens?
    when you click on this button, then...

One Program, Three Axes

consider that

- a program is in a space
- this space has three axes:
  - functional
  - static
  - dynamic
- describing one axis is not sufficient

  you have to describe the three of them!
- all the axes are tightly linked altogether

One Program, One major axis?

to specify (and/or describe) a program

there is no major axis!

however

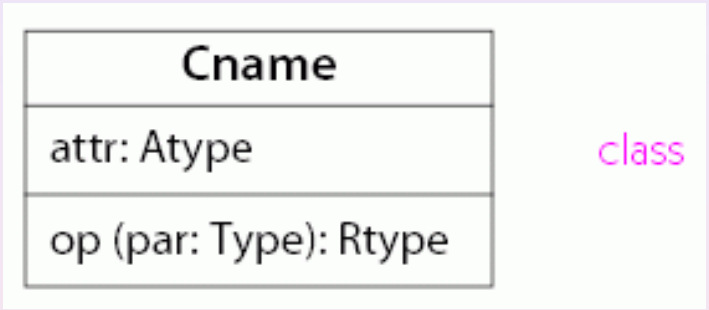
OO modeling is very “static-oriented”... ...because one models class hierarchies!

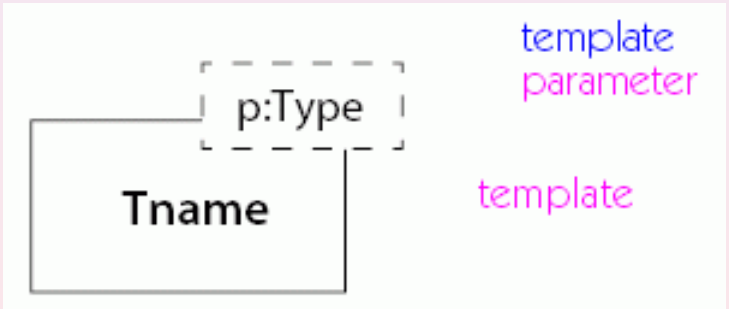
An Important Static Diagram

entity/relationship diagram $\rightarrow$ UML class diagram

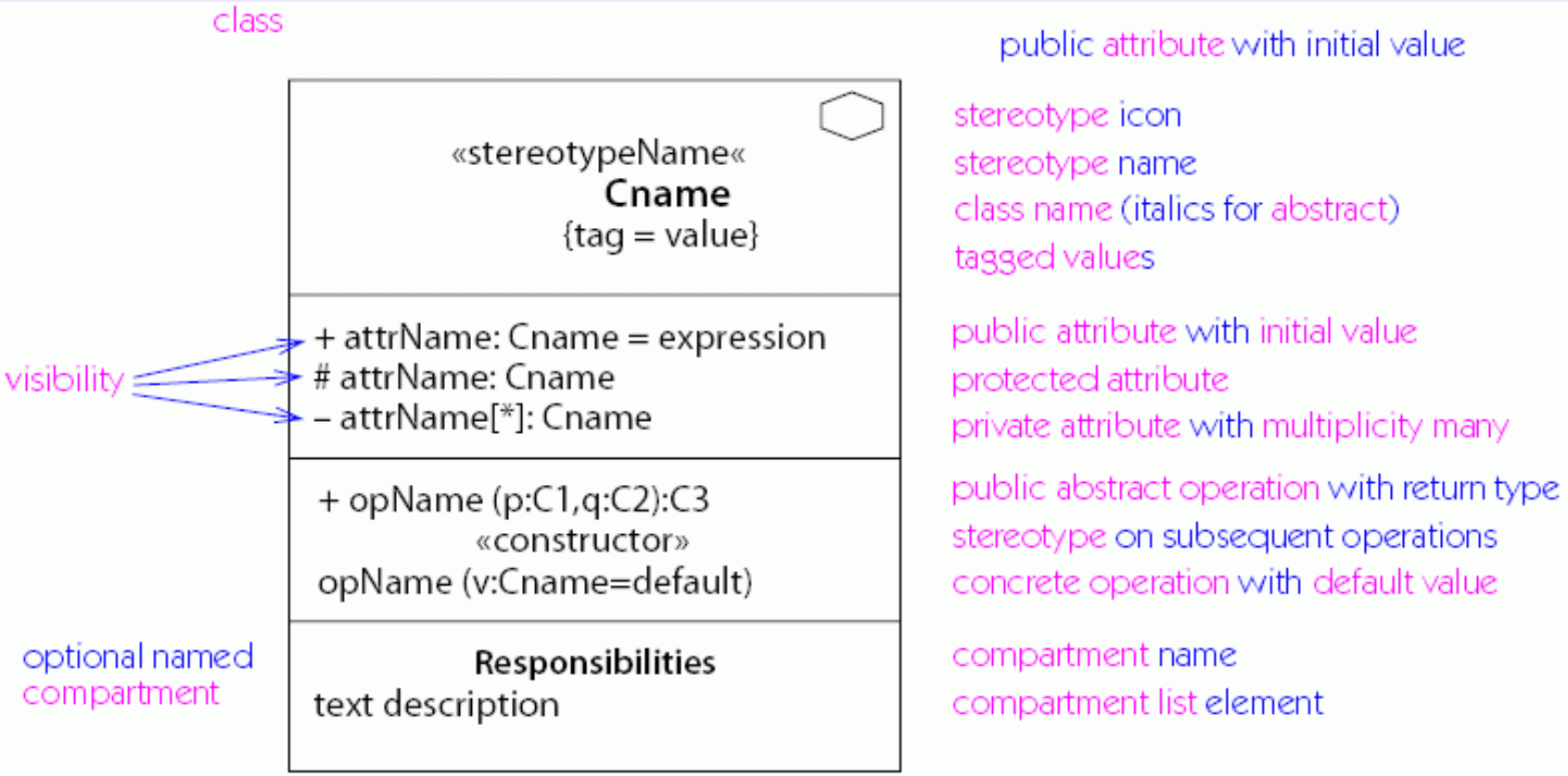
Outline

A Class in UML (1/3)

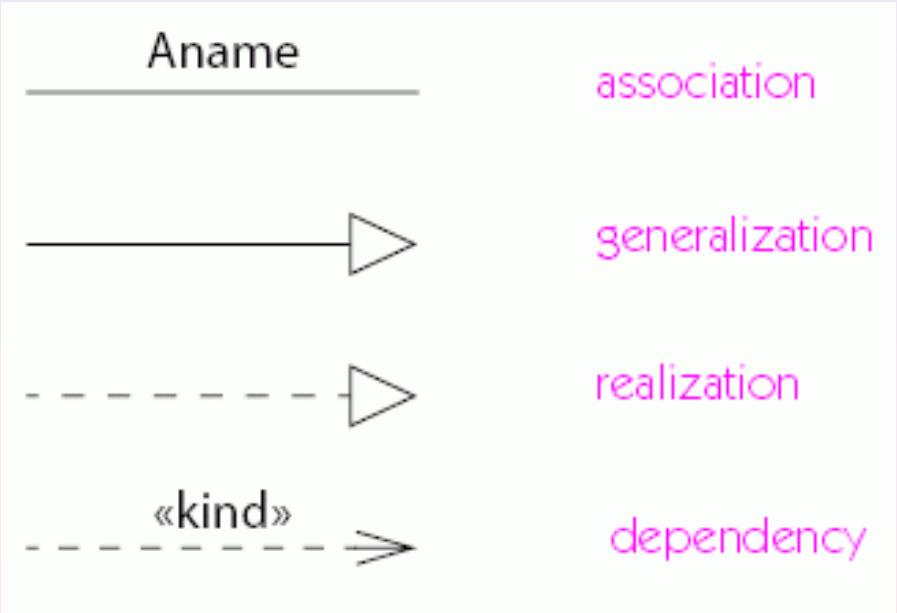




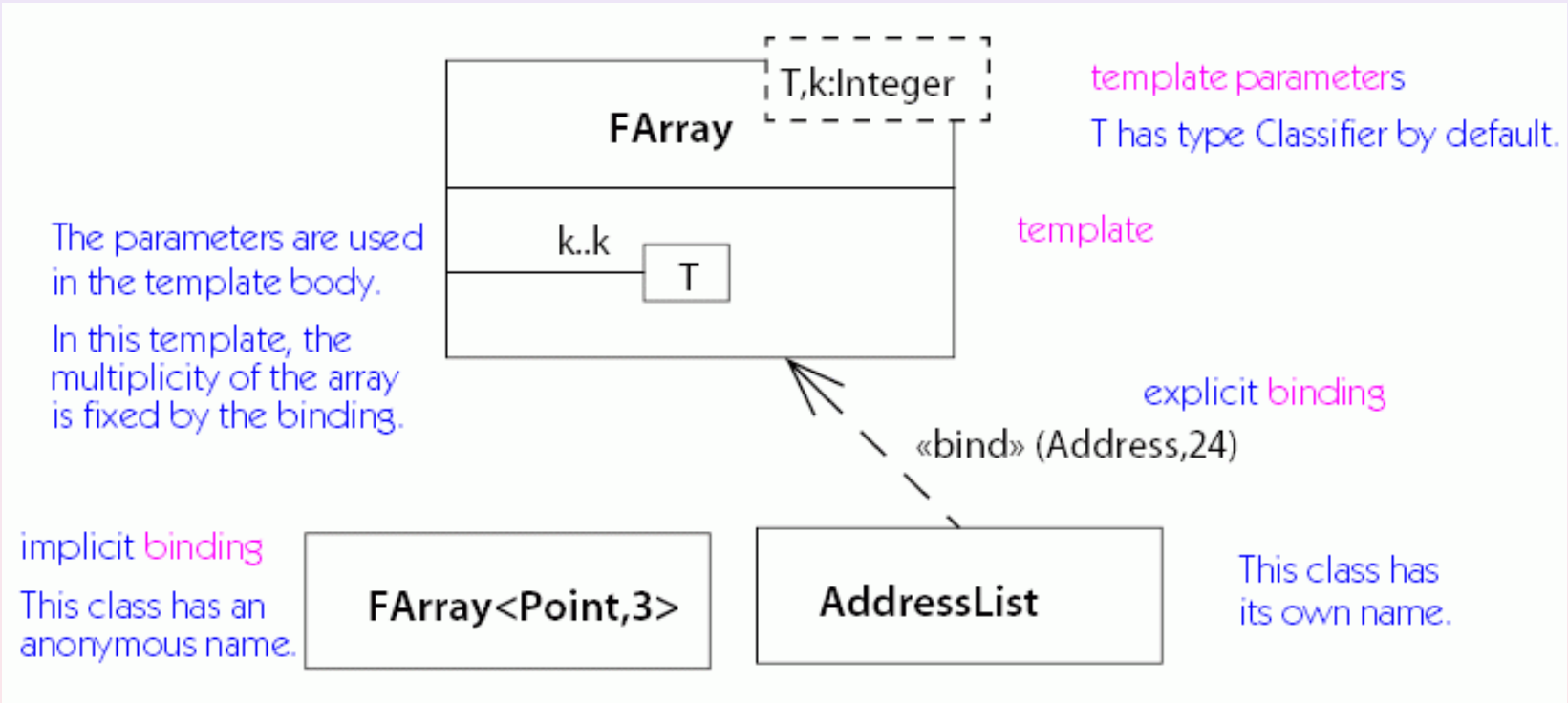
A Class in UML (2/3)



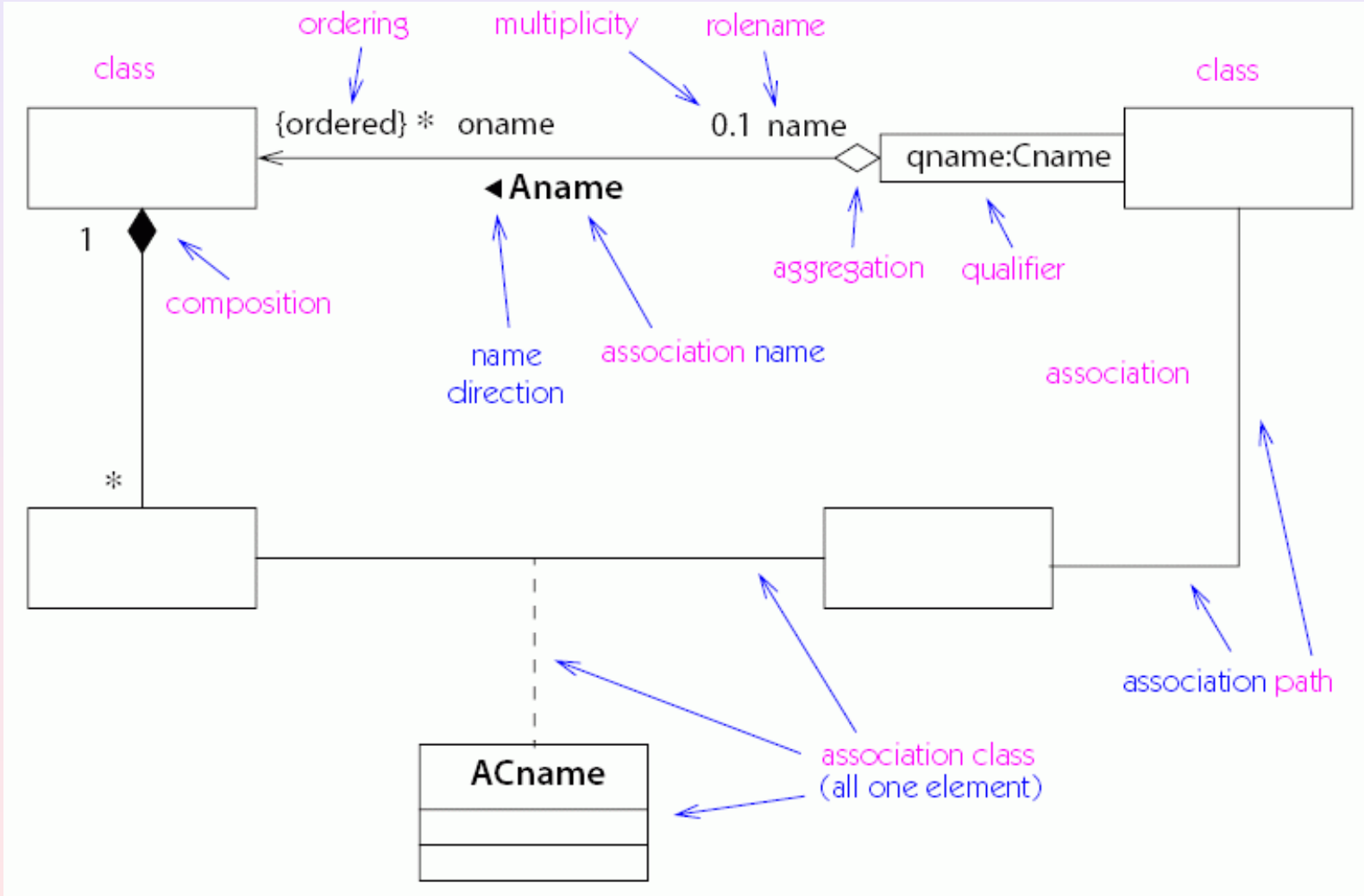
Relationships between Classes in UML (1/3)



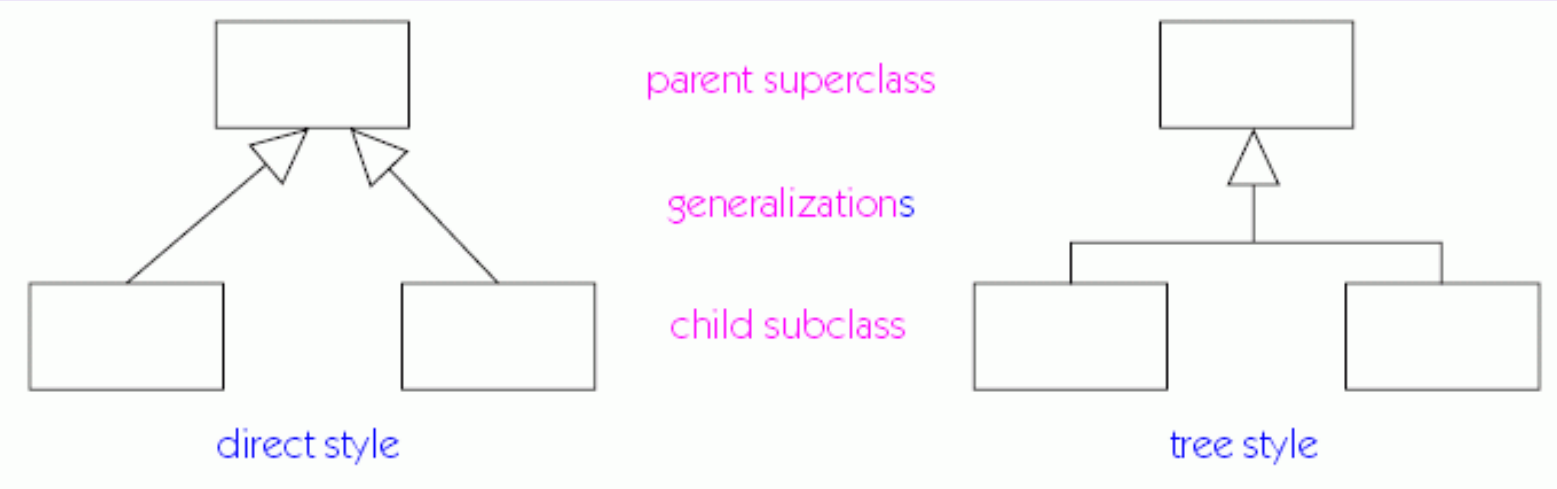
Relationships between Classes in UML (2/3)



Relationships between Classes in UML (3/3)



Inheritance Tree and Object in UML



Misc

note

{ expression }

constraint

Outline

Hints

First

- draw entities and their relationships
- split your program into several diagrams
  - so that *one* diagram shows *one* idea
  - so that each diagram is comprehensive

Then (and only then)

- show public stuff

Afterwards (and on specific separate diagrams)

- depict details

Exercise

< live! >

Outline

Variable and Type

C	translation
`int i;`	`i : int`
`int i = 3;`	`i : int = 3`

there is no ambiguity when we write:

variable : type = definition

Procedure

C	translation
`int f(float);`	`f : float -> int`
`int f(float arg);`	`f : (arg : float) -> int`

- the variable (entity; here a procedure) is named `f`
- the type of `f` is `float -> int`

  read: “takes float and gives int”

Type Definition

C	translation
`typedef def t;`	`type t = def`

Consider that `t` is just an alias (a name) for the type `def`

Enumeration

C

```
typedef enum { monday, /*...*/ sunday } day;
```

translation

```
type day = enum monday, .., sunday
```

Structure (1/2)

C

```
struct bar { bool b; double d; };
```

translation

```
type bar = { b : bool, d : double }
```

- The “`{ ... }`” notation means that we group a set of fields.
- Each field has a name (here `b` and `d`) and a type (resp. `bool` and `double`).

Structure (2/2)

a `struct` is a "product type"

with:

```
type day = enum monday, .., sunday
type month = enum january, .., december
type ydate = { d : day, m : month }
```

the values of `ydate` are:

- `(monday, january)`
- `(tuesday, january)`
- ...
- `(sunday, december)`

we have:

$$\text{card(ydate)} = \text{card(day)} \times \text{card(month)}$$

Union—or variant (1/2)

C
`union bar { bool b; double d; };`
translation
`type bar = [ b : bool | d : double ]`

- The "`[ ... ]`" notation means that we have exactly **one** of the fields.

Union—or variant (2/2)

we have a “sum type”

with:

```
type ydate = [ d : day, m : month ]
```

the values of `ydate` are:

- `monday`
- ...
- `sunday`
- `january`
- ...
- `december)`

we have:

$$\text{card(ydate)} = \text{card(day)} + \text{card(month)}$$

Method

C
`struct bar { int i; int f(float); };`

translation
`type bar = { i : int, f : float -> int }`

- The “`{ ... }`” notation means that we group a set of fields.
- Each field has a name (here `i` and d) and a type (resp. `int` and `double`).

About fields

In:

```
type bar = { i : int, f : float -> int };
```

we have two fields:

- τ_1 which is "`i : int`"
- and τ_2 which is "`f : float -> int`"

we can write:

$$bar = \{\tau_1, \tau_2\}$$

Outline

The Cornerstone (1/2)

consider the function foo:

```
foo : t -> void
```

and the variable v:

```
v : t'
```

what the relationship between the couple of types `t` and `t'` should be so that the following call is valid?

```
foo(v)
```

The Cornerstone (2/2)

we want:

- `t'` to be included in `t`
- `t` to be substituted by `t'`
- `t'` to be a sub-type of `t`

$$t' <: t$$

Simple Test

with:

```
type t = { i : int, f : int -> float }
foo : (arg : t) -> float =
  arg.f(arg.i)
  end
```

is the following program correct?

```
type t' = { i : int, f : int -> float, b : bool }
var : t' // = initialization
res : float = foo(var)
```