

# Featherweight Java

*Un “core-language object oriented” basato su Java*

## Cosa c'è in FJ

*Scopo:* astrarre i meccanismi fondamentali di Java per definire typing e semantica e provare type-safety in modo semplice.

*Vantaggio:* per ogni altra caratteristica aggiuntiva, si studia l'estensione di FJ con questa caratteristica e si estende la prova di type-safety in modo ancora...semplice.

*Meccanismi fondamentali*

*Termini:* oggetti, creazione di oggetti, invocazioni di metodi e selezione di campi, cast.

*Dichiarazioni:* classi, ereditarietà, riscrittura di metodi e binding dinamico.

# Cosa non c'è.....

*Sono volontariamente omessi tutti quegli aspetti che renderebbero la prova di "type-safety" solo più lunga, senza introdurre elementi significativi*

*(è comunque un frammento computazionalmente completo)*

Mancano (rispetto a full Java)

- parte imperativa: assegnamento e referenze
- controllo dell'accesso : tutto è pubblico
- eccezioni
- chiamate al *super* (tranne che nel costruttore)
- classi astratte (Interface) – inner classi –
- tipi di base (int, bool,...) (anche se li useremo talvolta in esempi)

FLP 08-09

3

## Un primo sguardo informale

Un programma FJ è formato da

**Parte Dichiarativa**

**termine (*main*)**

FLP 08-09

4

## ESEMPIO: Parte Dichiarativa D

```
class A extends Object { A() { super(); } }
```

```
class B extends Object { B() { super(); } }
```

```
class Pair extends Object {  
  Object fst; Object snd;  
  Pair(Object fst, Object snd)  
    { super(); this.fst=fst; this.snd=snd; }  
  Pair setfst(Object newfst)  
    { return new Pair(newfst, this.snd); }  
}
```

## ESEMPIO: D + Main

**D**

Lucido precedente

Espressione da valutare nel  
contesto delle dichiarazioni

```
new Pair (new A( ), new B( ) ).setfst (new B( ) )
```

Semantica

*Come risultato finale ci attendiamo :*

```
new Pair (new B( ), new B( ) )
```

## FJ Sintassi : *Termini e Valori*

$t ::=$	(termini)
$x$	<i>variabile</i>
$t.f$	<i>selezione campo</i>
$t.m(\overline{t})$	<i>invocazione metodo</i>
$\text{new } C(\overline{t})$	<i>creazione oggetto</i>
$(C) t$	<i>cast di tipo</i>
$v ::=$	(valori)
$\text{new } C(\overline{v})$	<i>creazione oggetto</i>

## FJ Sintassi : *Dichiarazioni*

- $K ::=$  *dichiarazione costruttore*  
 $C(C \overline{f}) \{ \text{super}(\overline{f}); \text{this}.\overline{f} = \overline{f}; \}$
- $M ::=$  *dichiarazione metodo*  
 $C m(C \overline{x}) \{ \text{return } t; \}$
- $CL ::=$  *dichiarazione classe*
- $\text{class } C \text{ extends } C \{ C \overline{f}; K \overline{M} \}$

dove  $\overline{C f}$  sta per  $C_1 f_1 \dots C_n f_n$ ,  $\overline{M}$  sta per  $M_1 \dots M_n$ , e simili.

## Notare!

- Si esplicita sempre la superclasse di una classe (eventualmente OBJECT che è assunta essere una classe vuota): i campi dichiarati dalla sottoclasse sono aggiunti a quelli della superclasse (e delle sue superclassi) e devono avere nomi distinti!
- L'oggetto è un new esplicito con il costruttore (e eventuali parametri), non ha un'identità (non ci sono referenze)
- È sempre esplicito l'oggetto su cui si invoca il metodo, anche nel caso del **this**
- Il costruttore ha una struttura prefissata: super(...) e inizializzazione dei nuovi campi con i (rimanenti) parametri (in ordine).

## Classe = Tipo

- Quando dichiaro una classe **A**, sto introducendo la definizione di un nuovo tipo di nome **A**
- La rappresentazione interna di questo tipo A è un **tipo-record**  
$$<< f_1: B_1; \dots; f_n: B_n >>$$
- Le istanze/oggetti di tipo A avranno una rappresentazione a record corrispondente

# SUBTYPING Nominale

- Come in Java, il sottotipo è dichiarato (sottoclasse)
- Assumiamo una **class table CT** : funzione che associa ad ogni nome di classe una dichiarazione
- CT(C) = class C extends D {...}  
C <: D
- C <: C
- C <: D , D <: E  
C <: E

C <: D

## Subtyping nominale vs. Strutturale

♦ **Nominale**: i nomi sono significativi! Da essi dipende l'insieme delle relazioni di sottotipo valide: il compilatore, al momento della dichiarazione di sottoclasse, deve solo controllare che la relazione di sottotipo sia rispettata (ad es. non ci sono nomi di campi uguali fra quelli ereditati e quelli definiti). Il confronto fra le rappresentazioni dei tipi avviene solo una volta sola.

♣ **Strutturale**: le relazioni di sottotipo valide dipendono dalla rappresentazione dei tipi (il subtyping è definito sui tipi in base alla loro struttura); ogni volta che in una derivazione di tipo devo provare una relazione di sottotipo, devo confrontare i tipi-record che li rappresentano

# Nominale vs. Strutturale:

## *Vantaggi*

### ◆ *Nominale:*

- + nomi/tipi sono utili anche a run-time (tag degli oggetti)
- + danno gratis i tipi ricorsivi e mutuamente ricorsivi (un tipo A può riferirsi ad A nella sua definizione, oppure A si riferisce a B che si riferisce ad A senza che abbia alcun senso domandarsi quale, fra A e B, sia definito per primo)
- + la relazione di sottotipo è decidibile in senso triviale

### ♣ *Strutturale*

- + sono molto più eleganti ed adatti a studi teorici, in quanto ogni espressione di tipo ha in sè tutte le informazioni per ragionare sulle sue proprietà

# Nominale vs. Strutturale:

## *Svantaggi*

### ◆ *Nominale:*

- 1) Una convenzione sui nomi ( la loro rappresentazione, le relazioni di sottotipo dichiarate, etc..) è preliminare alla stessa operazione di controllo che la definizione del tipo è corretta (es., che i metodi siano correttamente tipati)

Per esempio, una tabella delle classi del programma è costruita come passo preliminare, quindi è parametro del typing: il controllo della correttezza sia delle dichiarazioni che del main -correttezza rispetto ai tipi- usa le informazioni di questa tabella

- 2) Una convenzione sui nomi riguarda un “ambiente” : es., ambienti come reti etc. sono più refrattarie a tali convenzioni globali.

# FJ : la tabella CT delle classi

Una tabella di classi CT è una funzione da nomi di classi C a dichiarazioni di classi CL.

**Un programma è una coppia (CT, p) dove  $p=D t$  :**  
*in ciò che segue assumiamo di riferirci sempre a una fissata CT  
(se esaminiamo il programma p, è quella costruita dalle  
dichiarazioni di p)*

Si assume che CT soddisfi delle condizioni di consistenza (sanity conditions):

- per ogni C che compare in CT, C appartiene al dominio di CT (tranne che per OBJECT)
- non ci sono cicli nella relazione  $<:$  indotta da CT

## Definizioni Ausiliarie:

### *lookup campi*

- **fields(Object) =  $\emptyset$**  ( $\emptyset$  sequenza vuota)

- **CT(C) = class C extends D {  $\overline{C f}$ ; K  $\overline{M}$  }**

$$\underline{\text{fields(D)} = \overline{D g}}$$

$$\text{fields(C)} = \overline{D g}; \overline{C f}$$

**fields( C ):** restituisce la lista dei campi di C, ereditati e nuovi



## Definizioni Ausiliarie:

### *lookup tipi metodi*

- $CT(C) = \text{class } C \text{ extends } D \{ \overline{C} \text{ f}; K \overline{M} \}$   
 $B \text{ m } (\overline{B} \text{ x}) \{ \text{return } t; \} \in \overline{M}$   

---

$$\text{mtype}(\text{m}; C) = B \rightarrow B$$
- $CT(C) = \text{class } C \text{ extends } D \{ \overline{C} \text{ f}; K \overline{M} \}$   
 $\text{m is not defined in } \overline{M}$   

---

$$\text{mtype}(\text{m}; C) = \text{mtype}(\text{m}; D)$$

## Definizioni Ausiliarie:

### *lookup definizioni metodi*

- $CT(C) = \text{class } C \text{ extends } D \{ \overline{C} \text{ f}; K \overline{M} \}$   
 $B \text{ m } (\overline{B} \text{ x}) \{ \text{return } t; \} \in \overline{M}$   

---

$$\text{mbody}(\text{m}; C) = (\overline{x}; t)$$
- $CT(C) = \text{class } C \text{ extends } D \{ \overline{C} \text{ f}; K \overline{M} \}$   
 $\text{m is not defined in } \overline{M}$   

---

$$\text{mbody}(\text{m}; C) = \text{mbody}(\text{m}; D)$$

## Definizioni Ausiliarie: *riscrittura-metodi legale*

$\text{mtype}(m; D) = \overline{D} \rightarrow D_0$  implica  
 $\overline{C} = \overline{D}$  and  $C_0 = D_0$   
 $\text{override}(m, D, \overline{C} \rightarrow C_0)$

Il predicato  $\text{override}(m, D, C \rightarrow C_0)$  restituisce vero se il metodo  
 $m: C \rightarrow C_0$  è un corretto overriding in una sottoclasse di  $D$  (la riscrittura  
del metodo è legale solo se mantiene la stessa signature!)

## Semantica Operazionale

- *Regole di transizione* per ridurre termini a valori
- **Semantica *call-by-value*** : l'invocazione di un metodo è valutata solo dopo aver ridotto a valore sia il termine che rappresenta l'oggetto d'invocazione sia i parametri attuali (i valori a cui i termini si riducono sono termini di creazione-oggetti della forma  $\text{new } A(v_1 \dots v_n)$  )
- **Semantica *di computazione/small step***

# Regole di valutazione

$\frac{\text{fields}(C) = \bar{C} \ \bar{f}}{(\text{new } C(\bar{v})) . f_i \longrightarrow v_i}$	(E-PROJNEW)
$\frac{\text{mbody}(m, C) = (\bar{x}, t_0)}{(\text{new } C(\bar{v})) . m(\bar{u}) \longrightarrow [\bar{x} \mapsto \bar{u}, \text{this} \mapsto \text{new } C(\bar{v})] t_0}$	(E-INVKNEW)
$\frac{C <: D}{(D) (\text{new } C(\bar{v})) \longrightarrow \text{new } C(\bar{v})}$	(E-CASTNEW)

**Nota.** A) *this* è una variabile (esattamente come i parametri formali), rimpiazzata dall'oggetto di invocazione nel body del metodo; B) come in Java, un *cast* run-time non modifica l'oggetto, semplicemente ha successo o fallisce; C) il parametro attuale è un valore

# Regole di Congruenza

$\frac{t_0 \longrightarrow t'_0}{t_0 . f \longrightarrow t'_0 . f}$	(E-FIELD)
$\frac{t_0 \longrightarrow t'_0}{t_0 . m(\bar{t}) \longrightarrow t'_0 . m(\bar{t})}$	(E-INVK-RECV)
$\frac{t_i \longrightarrow t'_i}{v_0 . m(\bar{v}, t_i, \bar{t}) \longrightarrow v_0 . m(\bar{v}, t'_i, \bar{t})}$	(E-INVK-ARG)
$\frac{t_i \longrightarrow t'_i}{\text{new } C(\bar{v}, t_i, \bar{t}) \longrightarrow \text{new } C(\bar{v}, t'_i, \bar{t})}$	(E-NEW-ARG)
$\frac{t_0 \longrightarrow t'_0}{(C) t_0 \longrightarrow (C) t'_0}$	(E-CAST)

## ESEMPI: Valutazione...

- *Selezione campo*

`new Pair(new A(), new B()).snd`  
→ `new B()`

- *Cast*

`(Pair)new Pair(new A(), new B())`  
→ `new Pair(new A(), new B())`

## ESEMPI: Valutazione

- *Invocazione metodo*

```
new Pair(new A(), new B()).setfst(new B())
```

→ 
$$\left[ \begin{array}{l} \text{newfst} \mapsto \text{new B}(), \\ \text{this} \mapsto \text{new Pair}(\text{new A}(), \text{new B}()) \end{array} \right]$$

```
new Pair(newfst, this.snd)
```

i.e., `new Pair(new B(), new Pair(new A(), new B()).snd)`

## ESEMPI: Valutazione

- *Selezione campi + cast*

```
((Pair) (new Pair(new Pair(new A(), new B()), new A()).fst)).snd  
→ ((Pair)new Pair(new A(), new B())) .snd  
→ new Pair(new A(), new B()).snd  
→ new B()
```

**Definizione.** La riduzione  $\rightarrow^*$  è definibile come al solito: è la chiusura transitiva di quella a un passo.

## Nota sul *binding dinamico*

- Sistemi con tipi nominali (es. Java): ogni oggetto runtime ha un **tag** con il (concretamente, un puntatore al) suo tipo attuale (nome della classe con il cui costruttore l'oggetto è stato creato mediante "new"), che contiene un puntatore alla superclasse, etc...
- Il suddetto tipo attuale, con cui l'oggetto è etichettato, viene utilizzato per informazioni sul tipo dinamico, RTTI, per la ricerca dinamica dei body dei metodi associati all'oggetto (binding dinamico), etc.
- In FJ non ci sono le referenze agli oggetti, l'oggetto è direttamente un *new* di una classe; perciò il **tag runtime coincide con il tipo che è argomento della new.**

# Esercizio

Scrivere un programma con override di un metodo e descriverne la valutazione, evidenziando il binding dinamico per la chiamata del metodo riscritto.

## Sistema di Tipi

Insieme di regole, guidate dalla sintassi, per dimostrare asserzioni della forma

- $\Gamma \mid\!\!\!-\ t : C$   *$t$  ha tipo  $C$  in  $\Gamma$*
- ....  $m \dots$  OK in  $C$   *$m$  è ben tipato in  $C$*
- class  $C \dots$  OK  
*la definizione della classe  $C$  è ben tipata*

dove  $\Gamma$  è un insieme di assunzioni della forma  $x:C$ .

Si assume già data la tabella **CT**, su cui lavorano le funzioni ausiliarie.

## Typing dei termini (variabili e new)

$$\frac{x:C \in \Gamma}{\Gamma \vdash x : C} \quad (\text{T-VAR})$$

$$\frac{\begin{array}{l} \text{fields}(C) = \bar{D} \ \bar{f} \\ \Gamma \vdash \bar{t} : \bar{C} \quad \bar{C} <: \bar{D} \end{array}}{\Gamma \vdash \text{new } C(\bar{t}) : C} \quad (\text{T-NEW})$$

## Typing dei termini (selezione campi/invocazione metodi)

$$\frac{\Gamma \vdash t_0 : C_0 \quad \text{fields}(C_0) = \bar{C} \ \bar{f}}{\Gamma \vdash t_0.f_i : C_i} \quad (\text{T-FIELD})$$

$$\frac{\begin{array}{l} \Gamma \vdash t_0 : C_0 \\ \text{mtype}(m, C_0) = \bar{D} \rightarrow C \\ \Gamma \vdash \bar{t} : \bar{C} \quad \bar{C} <: \bar{D} \end{array}}{\Gamma \vdash t_0.m(\bar{t}) : C} \quad (\text{T-INVK})$$

# Typing dei termini (cast)

$$\frac{\Gamma \vdash t_0 : D \quad D <: C}{\Gamma \vdash (C)t_0 : C} \quad (\text{T-UCAST})$$

$$\frac{\Gamma \vdash t_0 : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)t_0 : C} \quad (\text{T-DCAST})$$

*Domanda* Perché 2 regole di tipo per il cast? Perché così è in Java.....

*Domanda:* perché c'è, invece, una sola regola di valutazione, quella per Up-cast, nella semantica operativa ?

## Nota sui cast

- **Typing:** ci sono 3 forme possibili di cast:
  - i) in 2 di esse, il *soggetto* è *sottoclasse* o *sopraclasse* del *target*, quindi il tipo statico (assunto dal type checker) è quello del target
  - ii) nella terza forma, il *target* ed il tipo del soggetto non hanno alcuna relazione di sottotipo: in tal caso vogliamo che il typechecker rifiuti come mal-tipata l'espressione, infatti così è in Java.....



## Ancora sui cast

**Valutazione:** quando riduco un cast, se il tipo run-time dell'oggetto è sottotipo del target, si elimina semplicemente il Cast (dunque, basta una sola regola di valutazione):

Negli altri casi mi arresto su un termine che non è un valore!

## Typing dei Metodi

$$\bar{x} : \bar{C}, \text{this} : C \vdash t_0 : E_0 \quad E_0 <: C_0$$
$$CT(C) = \text{class } C \text{ extends } D \{ \dots \}$$
$$\text{override}(m, D, \bar{C} \rightarrow C_0)$$

---

$$C_0 \text{ m } (\bar{C} \ \bar{x}) \{ \text{return } t_0; \} \text{ OK in } C$$

**Nota:** la definizione di *override* è un'implicazione; dunque, se  $m$  non occorre in  $D$ , l'antecedente **override(...)** è trivialmente vero!

# Typing delle Classi

$$K = C(\bar{D} \ \bar{g}, \ \bar{C} \ \bar{f}) \ \{\text{super}(\bar{g}); \ \text{this}.\bar{f} = \bar{f};\}$$
$$\text{fields}(D) = \bar{D} \ \bar{g} \quad \bar{M} \text{ OK in } C$$

---

$$\text{class } C \text{ extends } D \ \{\bar{C} \ \bar{f}; \ K \ \bar{M}\} \text{ OK}$$

*La definizione della classe  $C$  è ben tipata sse il costruttore ha la forma richiesta (chiama il `super` per inizializzare i campi della sopraclasse, poi inizializza i campi aggiuntivi di  $C$ , rispettando le concordanze di tipo dei parametri) ed ogni metodo è ben tipato in  $C$ .*

## Riassunto del Typing

- Il typing è in forma algoritmica (la subsumption è implicita, dove serve)
- Usando i tipi dei termini si può decidere se i metodi sono ben tipati o non (typechecking)
- Se i metodi sono ben tipati, si può decidere se la classe è ben tipata (ossia tutti i metodi sono ben tipati, i campi e il costruttore rispettano la forma richiesta) (typechecking)

## ...continua

**Un programma è una coppia (CT, p) dove  $p=D\ t :$**

Il programma p è ben tipato ed ha tipo C sse

- Tutte le dichiarazioni di D sono OK (ben tipate)
  - $\emptyset \mid \text{---} t : C$