

Equivalenza di tipi

Primi linguaggi → Solo tipi primitivi, ma successivamente vengono introdotti i tipi user-defined.

Come viene gestito un assegnamento? A seconda del **Type equivalence** - si divide in due possibili modi

① **Name equivalence** - **Tutto restrittivo** → I tipi di x e y devono avere lo stesso nome

② **Structural equivalence** - **Tutto flessibile** → I tipi di x e y devono avere la stessa rappresentazione interna (aumenta la possibilità di errori)

Euro x;
Dollar y;
 $z = x + y;$ // che senso ha?

immaginando che siano delle **typedef** ...
L'op. è possibile, ma non ha senso

C e C++ adottano entrambe le equivalenze, ma quelle strutturali non sono valide per le **struct** (richiederebbe di verificare la prop. di **biunivocità**)

```
typedef int money;  
typedef int apples;  
  
typedef struct{ int val; } S1;  
typedef struct{ int val; } S2;
```

```
int main() {  
    money x=0;  
    apples y=0;  
    int z = x+y; // non fa una piega  
    S1 a;  
    S2 b;  
    a = b; // errore di compilazione!
```

Compatibilità di tipi

Nei linguaggi OO l'equivalenza viene rimpiazzata da compatibilità/assegnabilità

Ogni sottotipo di T è assegnabile a T . simmetrica

compatibilità/assegnabilità → anti-simmetrica
(relazione d'ordine)

Nei linguaggi OO più comuni la compatibilità si basa su name:

- Due classi identiche ma con nome diverso sono diverse
- È monomio estendere una delle due

Linguaggio	Name equivalence	Structural equivalence
C	Sì (struct)	Sì (typedef)
Java	Sì (name compatibility, assegnabilità d'ordine anti-simmetrica)	No
ML	Sì (datatype)	Sì (type)

Caratteristiche utili alla classificazione dei linguaggi

Paradigma di riferimento

imperativo, OO, funzionale, logico

Scoping (statico/dinamico)

Gestione della memoria

Allocazione dinamica, presenza garbage collection, ...

Sistema di tipi

Forte/debole, statico/dinamico, encapsulation, equivalence/compatibility
polimorfismo (di 4 tipi), type inference...

Supporto alle eccezioni

Eventualmente integrato con type checking (vedi ML)

Supporto al parallelismo

Gestione thread, scambio di messaggi (*Remote Method Invocation*),
memory model, procedure asincrone, ...

Polimorfismo

Def: Uno stesso oggetto sentito (funzione, metodo variabili...) che cambia significato in base al contesto

In Java, $\underline{1 + 2}$ è olrem da $\underline{"a" + "b"}$

$+ : \text{int} \times \text{int} \rightarrow \text{int}$ $+ : \text{String} \times \text{String} \rightarrow \text{String}$



3/8

Ad hoc → Un operatore è designato a poter assumere arte "forme" diverse nelle specifiche del linguaggio

Universale → Numero illimitato di casi, come nel caso di overloading o di un ArrayList

classe parametrica che può contenere ogni tipo di classe

Polimorfismo ad hoc : Overloading, stesso nome ma implementazione diversa

23 + c (int, int) → int

12.34 + 1.0 (double, double) → double

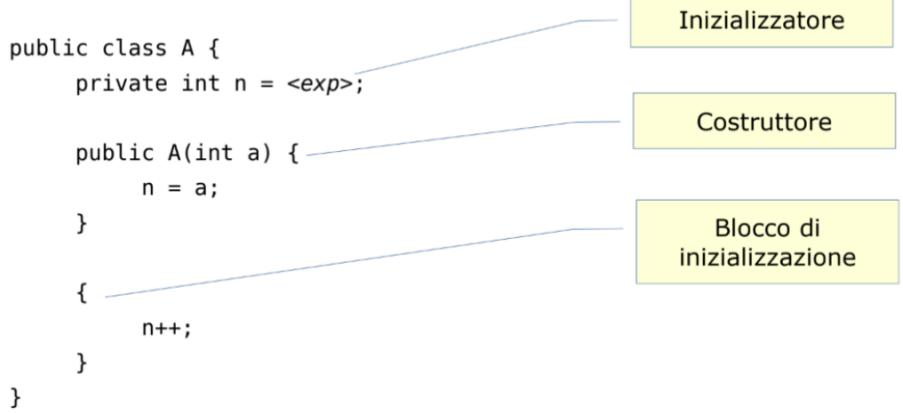
" " " : conversione, promotion

12.34 + 1 (double, double) → double
dovuta 1.0

Costruzione di oggetti

Elementi sintattici :

- Costitutore ;
- Invocazione di un costitutore a un altro, this e super ;
- blocchi di initializzazione nello scope delle classi, simili ai blocchi statici



l'concatenazione dei costruttori

l'invocazione esplicita:

Può essere chiamato con `this` (*sta nelle close*) o con `super` (*sta nelle superclose*) e dovrà comparire nella prima riga, altrimenti... *compile error*

Ricorda → `this` è il riferimento all'oggetto corrente, `super` si usa se si riferisce ad un elemento mascherato in una superclasse

In breve, hanno anche altri significati

Esempio 1

```
class A {  
    private int size;  
    public A() {  
        this(1000); // invoca l'altro costruttore  
    }  
    public A(int n) {  
        size = n;  
    }  
}
```

Esempio 2

```
class A {  
    public A() { this("Costruttore senza argomenti"); }  
    public A(String msg) { System.out.println(msg); }  
}  
class B extends A {  
    public B() {}  
    public B(int n) {  
        super("Valore: " + n);  
    }  
}
```

Chiamate implicite ad un altro costruttore:

Se un costruttore non inizia con this o super, il compilatore inserisce automaticamente il costruttore vuoto della super classe (`super();`)
↳ se il costruttore non esiste, **compila error**

Constructor chaining → invocazione fra costruttori

Sequenza di inizializzazione di un oggetto

- 1a) Se il costruttore inizia con `super(...)`, passo alle superclasse (**overloading**)
 - 1b) " " " " " `this(...)`, passo all'attuale costruttore
 - 1c) " " " non ha né `this` né `super`, passo al costruttore vuoto delle superclasse
- 2) Esegui i blocchi di inizializzazione e gli inizializzatori degli attributi
 - 3) Esegui il resto del costruttore
 - 4) Torna al chiamante

Concatenazione cieca → Se due o più costruttori si chiamano a vicenda, ci si trova in presenza di **mutua ricorsione non ben fondata, infinita**

- Ad esempio, tentando di compilare la seguente classe:

```
class A {  
    public A() { this(3); }  
    public A(int i) { this(); }  
}
```

- si ottiene il seguente errore di compilazione:

```
A.java:3: recursive constructor invocation  
    public A(int i) { this(); }  
               ^
```

