Indice

1 4-02-2019

DICHIARAZIONE ≠ ASSEGNAMENTO
L'assegnamento fa riferimento alla modifica di una variabile.
JAVA è un linguaggio *imperattivo* ad oggetti.

```
int n              \\ Dichiarazione
int m = n = 1;     \\ Inizializzazione
n = 2              \\ Assegnamento
```

Java è stato realizzato con un compilatore integrato, che non compila in assembly, questo compilatore invece produce un file sorgente che non è eseguibile direttamente dalla macchina, bensì è eseguibile da una virtual machine, la JVM(*java virtual machine*). In questo modo viene garantita la portabilità del codice in vari computer con CPU diversa (come il Phyton e .NET). Questa separazione non conta niente per il linguaggio, si riflette solamente sul modello architetturale.
U.M.L. = unified model language ⇒ rappresentazioni di gerarchie di classi.

> The file `ao.pdf` hasn't been created from `ao.dot` yet.
> Run '`dot -Tpdf -o ao.pdf ao.dot`' to create it.
> Or invoke LaTeX with the `-shell-escape` option to have this done automatically.

Tutte le sottoclassi sono dei sottoinsiemi.
Tutte le superclassi sono dei sovrainsiemi.
I linguaggi ad oggetti ci permettono di costruire *tipi* e di definire *valori*.

```
Animale a = new Animale(); \\ in Java gli oggetti sono valori
\\ dove: Animale -> tipo -> CLASSE
\\ a = nome variabile
\\ new Animale() ha un valore -> OGGETTO
```

Compilatore(compiling time):
-> controllo sintattico
-> controllo dei tipi, cioè che gli insiemi siano corretti

Esecuzione(RunTime):
-> Abbiamo a che fare con valori e non con tipi

I tipi di fatto sono una astrazione del linguaggio.
Il senso di un compilatore è quello di evitare di scrivere castonerie che a livello di esecuzione non avrebbero senso.

SOUNDNESS ⇒ un linguaggio è sound quando il compilatore ti da la certezza che funzioni

METODI: Ogni metodo dichiarato ha sempre un parametro implicito (il parametro *this*). Esso è sottointeso e viene passato automaticamente, più eventuali parametri che vengono passati all'interno del metodo

```
public class Animal {
```

```
        private int peso;
        ...
        public void mangia(Animali a){
                this.peso = this.peso + a.peso;
        }
}

Cane fido = new Cane();
a.mangia(fido);                                  \\ SUBSUMPTION (assunzione)
```

L'eredità è un meccanismo che garantisce il funzionamento del *polimorfismo*.Questa forma di polimorfismo è definita come *SUBTYPING*.
L'altra forma di polimorfismo sono i *GENERICS*, in modo tale da non perdere il tipo originario dell'oggetto passato al metodo.

```
\\ POLIMORFISMO SUBTYPING
\\ basato sui sottotipi ereditarieta
Object ident (Object x){
        return x;
}

\\ POLIMORFISMO GENERICS (parametrico)
\\ non perdo informazioni sui tipi
<T> T ident (T x){
        return x;
}
\\ questa funzione mi permette di riusare il metodo, in questo modo evito
\\ di fare CAST, e di sbagliare a farli
```

2 7-02-2019

Quali sono i tipi che risolvono il mio problema?
Nei linguaggi ad oggetti, lo strumento più potente è la classe. Quando definisco le entità che poi vado a tramutare in classi sto definendo DATI.
Prima di dire cosa faranno, vado a definire chi sono.
Le classi possono contenere dei metodi (funzioni che operano sugli oggetti della classe).
Definire sottoclassi significa definire sottoinsiemi nell'ambito dell'ereditarietà. Le nuove operazioni delle sottoclassi vanno inserite sapendo che le sottoclassi ereditano il set di funzioni delle superclassi.
L'*OVERRIDING* è il punto cruciale di tutta la programmazione ad oggetti. Se non potessi farlo significa che nelle sottoclassi non posso andare a specializzare un metodo. Specializzare un metodo significa cambiare l'implementazione della super classe senza cambiarne la firma.
@ serve per creare delle annotazioni nel codice, serve per il compilatore (es: @ override)
DINAMIC DISPATCHING: in fase di runtime serve a scegliere la versione giusta del metodo se ho degli over ride nelle mie classi. Se nella mia classe non esiste il metodo richiamato, il dinamic dispatching va a prendere l'implementazione del metodo dalla superclasse.
Le classi *STATICHE* sono quelle classi in cui non si può fare riferimento a se stessi. (Un esempio possono essere i metodi statici)
Le *COLLECTION* sono delle interfacce della libreria di JAVA e non si possono costruire.
Un *OGGETTO* è costituito da un insieme dei suoi campi e da puntatori ai metodi della classe ⇒ grazie a questo il dispatching funziona, funziona perchè il compilatore ha controllato i tipi e garantisce che nel compiling time tutto questo funzioni.
JAVA SE ⇒ Standard Edition
JAVA EE ⇒ Enterprise Edition
JAVA ME ⇒ Mobile Edition
JAVA JDK ⇒ linguaggio + tutte le librerie standard (java developement kit)
JAVA JRE ⇒ Solo a runtime, versione ridotta che serve solo a chi usa i programmi ma non al programmatore (java runtime enviroment)
File jar ⇒ Archivio di tutti i pacchetti del programma
JAVA JVM ⇒ (Java virtual machine) serve per eseguire i file .jar
La documentazione di java si trova on-line ed è diffusa in pacchetti che servono ad organizzare logicamente le classi, che sono organizzate in ordine alfabetico.
Le *COLLECTION* da sole non sono dei tipi, le collection di un "qualcosa" sono dei tipi.I tipi parametrici vogliono infatti un *argomento*

3 11-02-2019

ITERATORE: E' un pattner, uno stile di programmazione. Il pattern degli iteratori esiste in tutti i linguaggi ad oggetti. Con iteratore intendiamo lo scorrimento di una collezione di elementi.
ITERABLE è una super interfaccia, e l'interfaccia *COLLECTION* implementa questa super interfaccia. Iterable è super tipo di tutte le interfacce.
<? extends E>
SOTTOTIPO = 1) sei una sottoclasse (extends) 2) sei una sottointerfaccia (implements)
La *SUBSUMPTION* non funziona tra GENERICS. Per il parametro stesso c'è subsumption, ma non per le collection.
TIPO ESTERNO: funziona sempre la subsumption
TIPO INTERNO: non funziona, solo con <? extends E>
[Invarianza del subtyping]: Se ciò non fosse le assunzioni funzionerebbero anche nel tipo di ritorno e questo rischierebbe la totale spaccatura
Se cosi non fosse in java non verrebbero mai rispettate le regole delle classi.
Java di unico ha che esiste il wildchart (?), che è un modo controllato per risolvere questo problemino.
Prima dei generics (2003/2004) in java si programmava tutto a typecast. Per motivi di retrocompatibilità è possibile programmare in tutti e due i modi. E' comunque consigliato usare la porgrammazione con i GENERICS.
Metodi che ritornano un booleano iniziano con sempre come se fossero domande; es: hasNext, isEmpty etc..
Un iteratore non può essere costruito con un new perchè è un'interfaccia.

4 15-02-2019

```
public interface Iterator<T>{}
```

L'interfaccia è un contratto, nel senso che ti promette di fare qualcosa.
In java si scrive il codice ancora prima di sapere cosa andrà a fare.
Il contratto di iteratore è il seguente:
->boolean hasNext();
->T next();
->void remove()
Non bisogna sapere necessariamente come sono stati implementati, ti basta sapere che esistono per poter dire che sia un iteratore.
Esempio di definizione di un metodo con iteratore come input:

```
public statics void scorri(Iteratore <Integer> it){
        while(it.hasnext()){
                integer n = it.next();
        }

}
```

Esempio di utilizzo

```
scorri(new Iterator<>(){
        ...
        ...
        ...

});
```

Quest'ultima è un'espressione, o come meglio dire, un'oggetto fatto al volo. Questa sintassi è stata creata appositamente per le interfacce (dato che non si possono costruire), senza dover andare a definire una classe con la classica implementazione dell'interfaccia.
ANONYMOUS CLASS meccanismo comodo per design pattern come le call-back.
Questa implementazione garantisce che la funzione sia *SOUND*, e non crasherà mai a *RunTime*
IMPLEMENTARE INTERFACCE
1) Con implements:
-> controlla i metodi che hai implementato all'interno della classe
-> assicura che siano implementati tutti
Tipi delle interfacce
Iterator $\Rightarrow$ non è un tipo
Iterator<T> $\Rightarrow$ è un tipo

5 18-02-2019

CLASSE ASTRATTA perve per impedire la sua costruzione (non ne posso costruire quindi una istanza) e vengono definite astratte se anche un solo metodo è astratto.
Un array è una struttura dati lineare, omogenea e contigua in memoria

6 21-02-2019

BINDING avviene anche con i tipi
I type argument fanno binding con i type parameter, esattamente come avviene per le funzioni tra argomenti e parametri.
Quando si programma con i generics si PROPAGANO.
TYPE ERASURE: cancellazione dei tipi: java lo fa quando compila i generics mettendoci Object: il motivo è per mantenere la compatiblità con il vecchio codice che non aveva generics. Quindi i generics sono verificati dal compilatore e poi cancellati per eseguire.

7 25-02-2019

L'ereditarietà serve anche a modificare ei metodi della classe che viene ereditatà. E' l'unico modo che abbiamo per modificare delle cose anche se non sappiamo cosa e chi le ha costruite.
I metodi statici non si possono override perchè non sono presenti nelle virtual table.

8 28-02-2019

COVARIANZA e CONTROVARIANZA dei tipi
$C_1 < \tau_1 > \leq C_2 < \tau_2 > \Leftrightarrow C_1 \leq C_2 \bigwedge \tau_1 \equiv \tau_2$

Questa regola del type system di java si dice che il linguaggio NON è COVARIANTE, in quanto i generics non cambiano.

```
@Override
public PastoreTedesco m(Animale c){ return new PastoreTedesco;}
\\ il tipo di ritorno del metodo (PastoreTedesco) scende (sottoclassi)
\\ il tipo del parametro di ingresso (Animale) sale (superclasse)
```

In java è possibile controvariare solo il tipo di ritorno del metodo, solo scendendo (sottotipo)

```
public Cane m (Cane c){return c;}

@Override
public PastoreTedesco m(Cane c){return new PastoreTedesco();}
```

SOUND: un programma che compila e termina, a meno di una eccezione.
In java è possibile avvenga un segmentation fault non per un problema di casting, ma solamente se accediamo ad un indice di un array non abbiamo allocato.

Ci sono linguaggi dove non esistono gli array, quindi non accadrà mai segmentation fault e il codice terminerà sempre, ovviamente senza fare i controlli di semantica.
Recentemente è stato inserito un pattern che qualcosa la covarianza:

```
Arraylisti<? extends Animale> m = new Arraylist<Cane>();
```

Da questo si capisce che la covarianza può essere usata, ma solamente se esplicitata con il wildcard.
Sono molto usati perchè non sono tipi del primo ordine
Non posso definire una variabile:

```
? extends Animale m = new Cane();
\\ questa sintassi si puo usare solo come type argument
```

Significato: permettono la covariamza, sono tipi temporanei che non possono essere scritti nel codice, però possono essere sostituiti con il get().
Un altro DESIGN PATTERN: callback