Indice

1 4-02-2019 **2**

2 7-02-2019 **4**

3 11-02-2019 **5**

4 15-02-2019 **6**

5 18-02-2019 **7**

6 21-02-2019 **8**

7 25-02-2019 **9**

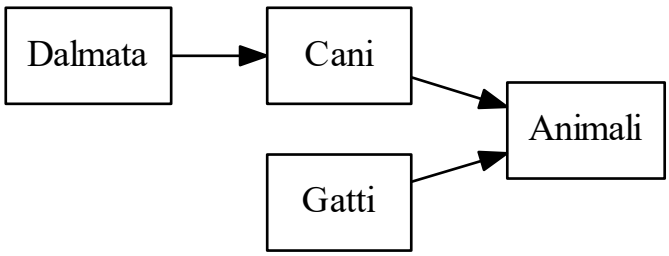
8 28-02-2019 **10**

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, bensi 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.



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)
IL POLIMORFISMO è uno strumento molto utile perchè ci permette di scrivere codice, funzioni che posso adoperare anche con tipi diversi!
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. In memoria infatti ci sono tutti i puntatori ai metodi di una classe, durante il run time viene eseguito il codice del puntatore corretto. (*vedere: virtual table*)
Le classi *STATICHE* sono quelle classi in cui non si può fare riferimento a se stessi. (Un esempio possono essere i metodi statici). Il riferimento a *this* nei metodi statici è assente. I metodi statici "appartengono" alla classe, non ad una sua istanza! Una classe statica interna non vede il riferimento this dell'altra classe.
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.
Ogni espressione ha un tipo!
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. L'iteratore serve quindi a scorrere una collection.
ITERABLE è una super interfaccia, e l'interfaccia *COLLECTION* implementa questa super interfaccia. Iterable è super tipo di tutte le interfacce.
Se una interfaccia rappresenta una super interfaccia significa che non ha un genitore, anche se in realtà estende *Object*
Una mappa è una struttura dati che mappa chiavi e valori, ha quindi due parametri: il tipo della chiave e il tipo del valore. Una mappa è una collection solo se vista come una collection di coppie.
Una collection è figlia di iterable (la posizione più alta nella gerarchia)
<? extends classe> rappresenta un tipo.

<? extends E>
SOTTOTIPO = 1) sei una sottoclasse (extends) 2) sei una sottointerfaccia (implements)

ArrayList ha come superclasse abstract arrayList. ArrayList implementa collection, quindi ne è sottotipo ma non sottoclasse.

La *SUBSUMPTION* non funziona tra GENERICS. Per il parametro stesso c'è subsumption, ma non per le collection. Il tipo con il ? accetta sottotipi di parametri.
TIPO ESTERNO: gode sempre della la subsumption, il tipo interno NO, 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
NOTAZIONE BNS
BNS è il nome della notazione e serve per poter dare delle regole grammaticali. E' una notazione che definisce la sintassi delle espressioni
Iterator da solo, sintatticamente, sarebbe un tipo. Ma il compilatore verifica che non è un tipo e da errore.

5 18-02-2019

una è *CLASSE ASTRATTA* quando ha almeno un metodo astratto, serve 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.
Una interfaccia è zucchero sintattico di una classe astratta con soli metodi astratti.
La differenza tra classe ed interfaccia non esiste.
Per leggere una collection si usano gi iteratori che servono per gettare in sequenza.

```
public static class Animale(){
        privae int peso;
}

public static class Cane extends Animale{
        private String nome;
        public void abbaia(){};
}

public static class PastoreTedesco extends Cane{

}
```

Se costruisco un oggetto di tipo pastoreTedesco, esso sarà grande quanto un tipo int (32 bit) ed una stringa (un puntatore). Il tutto grazie alla virtual table che tiene in memoria i puntatori dei vari campi di uno oggetto.

6 21-02-2019

REFLECTION è una tecnica per conoscere i tipi e il contenuto delle classi a runtime.
Se voglio conoscere il tipo dell'enclosing class (classe che contiene) posso fare : nome.classe.this.nome

BINDING avviene anche con i tipi
I parametri di una funzione sono binding nello scope della funzione.
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 butta via i generics generando classi non anonime e li sostituisce con 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

Se un oggetto ha dei campi esso pesa tanto quanto la dimensione dei campi.
Nei pc a 64 bit i puntatori pesano 8 byte

L'ereditarietà serve anche a modificare i metodi della classe che viene ereditata. E' l'unico modo che abbiamo per modificare delle cose anche se non sappiamo cosa e chi le ha costruite. Sopratutto se non le possediamo. Un esempio è la classe ArrayList, che deve essere ereditata per implementare un motodo che ci permetta di scorrerla all'indietro.

I metodi statici non si possono override perchè non sono presenti nelle virtual table (sono funzioni sciolte).

Regola ereditarietà costruttore: se non definisco nessun costruttore nella sottoclasse è come se chiamassi il costruttore della superclasse SENZA parametro.

8 28-02-2019

COVARIANZA e CONTROVARIANZA dei tipi
VARIANZA :
$C_1 < \tau_1 > \leq C_2 < \tau_2 > \Leftrightarrow C_1 \leq C_2 \bigwedge \tau_1 \equiv \tau_2$ Questa regola del type system di java si dice che il linguaggio NON è COVARIANTE, in quanto i generics non cambiano.

Esempio: ArrayList<cane> è minore uguale a List<cane> (sottotipo)
Esempio: ArrayList<cane> NON è minore uguale a List<Animali>
L'ultima formula non è covariante, se fosse possibile si avrebbe una doppia soluzione sul guscio interno e il tipo esterno

CONTROVARIANZA :
Quando eredito posso fare l'overide, quando lo faccio il tipo di ritorno è constrovariante (uno sale e uno scende. Il parametro sale (si specializza) e il ritorno scende (si despecializza).

```
\\nella classe Animale:
public Cane m(Cane c){
        return c;
}

\\nella classe Cane:
@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 può essere eseguito
SOUND JAVA: 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 covarianza, sono tipi temporanei che non possono essere scritti nel codice, però possono essere sostituiti con il get().
Un altro DESIGN PATTERN: callback