Indice

1 4-02-2019

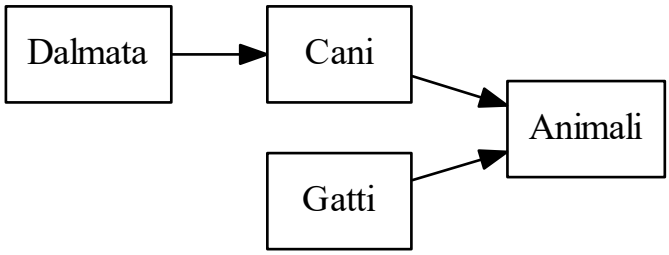
DICHIARAZIONE ≠ ASSEGNAMENTO
L'assegnamento fa riferimento alla modifica di una variabile.

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

JAVA è un linguaggio *imperattivo* ad oggetti.

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

PARAMETRO IMPLICITO
Ogni metodo dichiarato ha sempre un parametro implicito (il parametro *this*). Esso è sotto inteso 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)
```

POLIMORFISMO
L'eredità è un meccanismo che garantisce il funzionamento del *polimorfismo*.
• Polimorfismo per *SUBTYPING* o anche polimorfismo per inclusione: si riferisce al fatto che una espressione il cui tipo sia descritto da una classe A può assumere valori di un qualunque tipo descritto da una classe B sottoclasse di A (Vedi codice appena sopra)
• Polimorfismo dei *GENERICS*: si riferisce al fatto che il codice del programma può ricevere un tipo come parametro invece che conoscerlo a priori (polimorfismo parametrico). In questo modo non si perde il tipo originario passato dall'oggetto 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

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.
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.

OVERRIDING
L'*OVERRIDING* è il punto cruciale di tutta la programmazione ad oggetti. Fare overriding significa sovrascrivere un metodo ereditato dalla super classe per poterne specializzare il suo comportamento. 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)

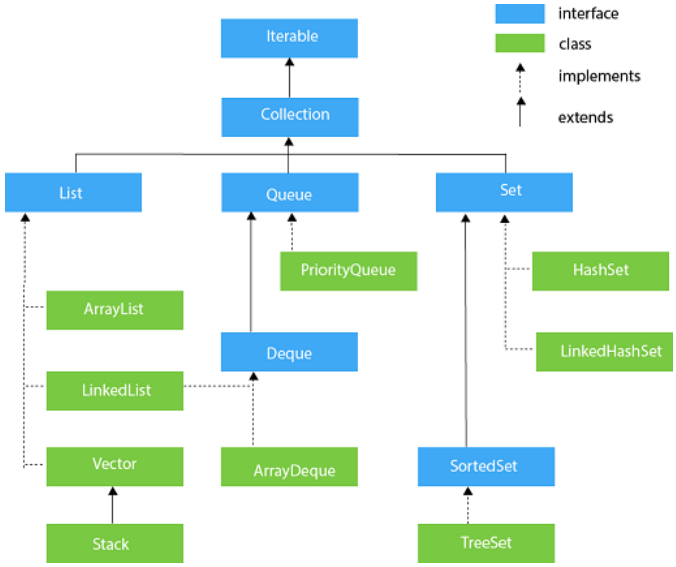
IL POLIMORFISMO è uno strumento molto utile perchè ci permette di scrivere codice, funzioni che posso adoperare anche con tipi diversi!

DINAMIC DISPATCHING
Il *DINAMIC DISPATCHING* serve in fase di runtime a scegliere la versione giusta del metodo da richiamare. Infatti se ho degli over ride nelle mie classi, sarà solo in fase di run time che Java deciderà quale metodo richiamare. Se nella mia classe non esiste il metodo richiamato, il dinamic dispatching va a prendere l'implementazione del metodo dalla superclasse. Nella memoria che contiene le informazioni degli oggetti ci sono tutti i puntatori ai metodi di una classe, in run time viene eseguito il codice del puntatore corretto. (*vedere: virtual table*). Un *OGGETTO* infatti è costituito da un insieme dei suoi campi e da puntatori ai metodi della classe ed è grazie a questo che il dispatching funziona: il compilatore controlla i tipi e garantisce che nel compiling time tutto questo funzioni.

Ogni espressione ha un tipo!

CLASSI E METODI STATICI
• I metodi statici sono quei metodi di una classe che appartengono alla classe, non alle istanze di una classe. Si possono richiamare senza creare un oggetto. I metodi statici possono quindi accedere solo a dati statici e non alle variabili di istanza della classe, possono solo richiamare altri metodi statici della classe, e sopratutto non possono usare il parametro implicito *this*.
• Le classi statiche in java possono solamente esistere se sono *innestate (nested)*. Esse possono accedere solamente dati statici della classe che le contiene. Una classe statica interna non vede il riferimento this dell'altra classe, essa può accedere solamente ai campi statici della classe che la contiene.

Le *COLLECTION* sono delle interfacce della libreria di JAVA e non si possono costruire.
Le *COLLECTION* da sole non sono dei tipi, le *COLLECTION* di un "qualcosa" sono dei tipi.I tipi parametrici vogliono infatti un *argomento*



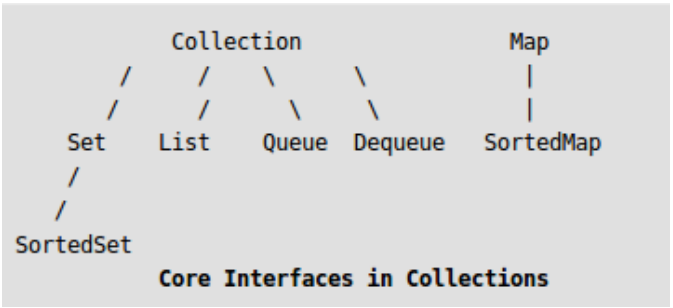
PACCHETTI JAVA
JAVA SE ⇒ Standard Edition
JAVA EE ⇒ Enterprise Edition
JAVA ME ⇒ Mobile Edition
JAVA JDK ⇒ linguaggio + tutte le librerie standard (java developement kit)
JAVA JRE ⇒ Solo a runtime, versione ridotta che serve solo a chi usa i programmi ma non al programmatore (java runtime enviroment)
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.

3 11-02-2019

• *ITERATORE*: E' un pattern, 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*: E' 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*

MAPPA
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. Infatti una *collection* è figlia di *iterable* (la posizione più alta nella gerarchia), ma una *mappa* NON è figlia di *iterable*



GENERICS
<? 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

INTERFACCE

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

L'interfaccia è un contratto, nel senso che mette a disposizione una serie di metodi che ogni classe che estende quell'interfaccia deve obbligatoriamente implementare, pena un errore durante la fase di compilazione.In java quindi si può scrivere del codice ancora prima di sapere come si potrebbe implementare.

Facciamo un esempio: il "contratto" di iteratore è il seguente:
• boolean hasNext();
• T next();
• void remove()
Data una certa classe che può non essere sotto al nostro controllo non abbiamo bisogna sapere necessariamente come sono stati implementati i suoi metodi, ma ci ti basta sapere che esistono per poter dire se sia o meno 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 istanziare direttamente), 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

CLASSE ASTRATTA
Una classe si dice astratta quando ha almeno un metodo astratto, essa serve per impedire la sua costruzione (non posso quindi istanziarla). Delle classi vengono definite astratte se anche un solo metodo è astratto. Una delle maggiori differenze tra classi astratte ed interfacce è che una classe può implementare molte interfacce ma può estendere una sola classe astratta. Una interfaccia è zucchero sintattico di una classe astratta con soli metodi astratti. Zucchero sintattico (Syntactic sugar) è un termine coniato dall'informatico inglese Peter J. Landin per definire costrutti sintattici di un linguaggio di programmazione che non hanno effetto sulla funzionalità del linguaggio, ma ne rendono più facile ("dolce") l'uso per gli esseri umani. La differenza tra classe ed interfaccia in realtà non esiste.

Un array è una struttura dati lineare, omogenea e contigua in memoria.

Per leggere una *collection* si usano gli iteratori che servono per farnel il get 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
La *reflection* è una *features* del linguaggio java che non tutti i linguaggi di programmazione posseggono (Ad esempio il C++ non la possiede). Essa ci permette diconoscere i tipi e il contenuto delle classi a runtime. Ad esempio se voglio conoscere il tipo dell'enclosing class (classe che contiene) posso fare : nome.classe.this.nome
Ad esempio se abbiamo degli oggetti che vengono passati dentro ad un metodo che come parametro ha il tipo Object, non saremo più in grado di distinguere il loro tipo di classe "originale", per superare questo problema possiamo invocare la funzione getClass() che ci ritorna il loro vero tipo.

BINDING
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
La cancellazione dei tipi in java avviene quando il compilatore "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

9 4-03-2019

DESIGN PATTERN dei tipi
->Iteratore
->Compact
callback
unary function
LAMBDA ASTRAZIONI -> servono per fare funzioni
FUNZIONI DI ORDINE SUPERIORE
Sono delle funzioni che prendono delle funzioni come parametri di ingresso

```
public interface Func<A, B>{
        B execute(A a);
}

public static <A,B> List<B> map(List<A> l, Func<A,B> f){
        List<B> r = new ArrayList<>();
        for(A x: l)
                r.add(f.execute(x));
        return r;

}
```

A e B sono *generics* locali al metodo(e solo al metodo)
I generics sulle classi servono per parametrizzare, non per fare polimorfismo

```
public static <A,B> List<B> map(List<A> l, Func<A,B> f){
\\ dove public static <A,B> dichiaro
\\ mentre in List<a> .. Func <A,B> uso
```

Funzione FILTER:

```
public static <A> List<A> Filter (List<A> l, Func<A,boolean> p){
        List<A> r = new ArrayList<>();
        for(A x : l)
                if(p.execute(x))
                        r.add(x);
        return r;
}

public static <A> void Filter2 (List<A>, Func<A, boolean> p){
        for(A a: l)
                if(p.execute(a))
                        l.remove(a);

}
```

Rimuovere un elemento da una coolection mentre la si usa è illegale
Posso chiedere all'iteratore di rimuovere l'elemento , esso rimuoverà quello a cui stiamo puntando

```
public static <A> void Filter2 (List<A>, Func<A, boolean> p){
        Iterator<A> it = l.iterator();
        while(it.hasNext()){
                A a = it.next();
                if(!(p.execute(a)))
                        it.remove();
        }

}
```

Posso usare Function<A,B> di java come funziona func?
Esempio di chiamata:

```
public static void main(String argv[]){
        List<String> strings = new ArrayList<>();
        string.add("ciao");
        string.add("pippo");
        string.add("unive");
        List<Integer> r = map(strings, new Func<String, Integers>{
                @Override
                public Integer execute(String a){
                        return a.length();
                }
        });
}
```

Data una lista di interi scartare gli elementi minori di zero

```
public static void main(String argv[]){
         List<Integer> interi  = new ArrayList<>();
         interi.add(89);
         interi.add(34);
         interi.add(-16);
         interi.add(560);
         interi.add(-1);
         interi.add(46);
         List<Integer> l = Filter(int s, new Func<Integer, boolean>{
                @Override
                public Boolean execute (Integer a){
                        return a>=0;
                }
         });
}
```

Oppure

```
        Filter2 (interi , new Func<Integer, Boolean>(){
```

```
        @Override
        public Boolean execute (Integer a)
                return a>=0;
});
```

Generics Locali (Polimorfismo parametrico di primo ordine)

```
public static <x> x ident2(x x){
        return x;
}

public static void main(){
        Cane fido = new Cane();
        Cane c2 = ident2(Fido);
        Gatto g = ident2(new Gatto());
}
```

10 7-03-2019

```
List<?> l1 = new ArrayList<Cane>();
\\ ? -> Da solo significa che indica un tipo che gerarchicamente
\\ e pie in alto di Object, viene detto top type

l1.get(int index) \\ritorna un capture of ? , qualcosa che sia figlio del top type
```

il tipo ? non può essere usato come tipo per una variabile, ? x non si può fare, però posso fare Object x = l1.get(..)

Mentre ? extends Animale -> qualsiasi cosa che sia figlio di animale
l2.get(0) -> ritorna un capture of ? extends Animale -> qualsiasi cosa figlia di animale (posso però fare Binding di qualcosa che sia al massimo Animale)

Posso subsumero solo il tipo esterno, se voglio subsumere anche il tipo interno devo usare le wildcards.
? super Animale -> qualcosa che sia più su di Animale (più generale)
l2.add(new Animale) -> ?? non compila perchè...

Riprenderemo la map vista l'altra volta. Ad esempio, per trasformare animali in piante:

```
public static class Vegetale{}

public static void main_map(){
        List<cani> l1 = new ArrayList<>();
        List<Vegetali> l2 = map(l1, new func<Animale, Vegetale>(){
                @Override
                public Vegetale execute(Animale a){
                        return null;
                }

        });
}
```

Questo non compila in quanto i generics non sono soggetti alla subsumption.
Per farlo compilare modifichiamo la funzione map:

```
public static  <x, y> List<x> map(List<x>, Func(? super <x, y> f)){
...

}
```

NESTED CLASS
La Nested Class è totalmente senza relazione rispetto alla enclosed class.

Nel caso precedente main_functional è la enclosed class, in quanto sto lavorando su quella.
Le nested class vedono i campi della enclosing class, ma se sono statiche non vedono i campi.

```
public static  <x, y> List<y> map(List<x>, Func<? super x ,? extends y> f)){
...

}
```

Questa è la versione più generale possibile.

Overloading
Permette di definire metodi con stesso nome ma firma diversa.

```
public static class c{
        public int m(){
                return 1;
        }
        public int m(int x){
                return x+1;
        }
        public int m(float x){
                return (int)(x-1.0f);
        }
        public int m(int x, int y){
                return x+y;
        }
}
```

L'overloading non è permesso cambiando il tipo di ritorno e lasciando il resto inalterato.
Devono essere diversi i parametri!
-ordine
-tipi
-numeri

```
public Number m(Number x){
        return x;
}
```