Indice

1 4-02-2019

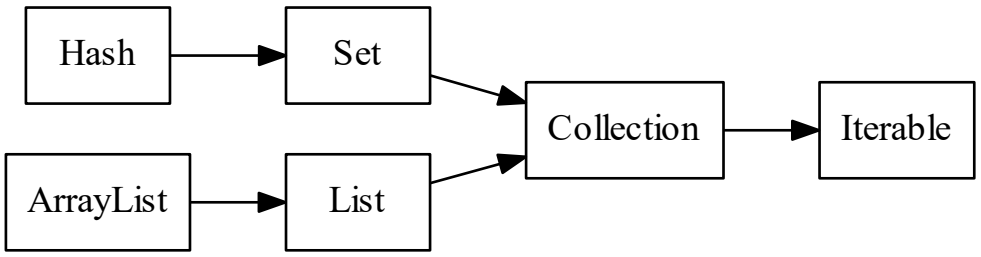
DICHIARAZIONE ≠ ASSEGNAMENTO
L'assegnamento fa riferimento alla modifica di una variabile.

```
1
2 int n       /* Dichiarazione */
3 int m = n = 1; /* Inizializzazione */
4 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*.

```
1   Animale a = new Animale();
2   /* in Java gli oggetti sono valori
3    * dove: Animale -> tipo -> CLASSE
4    * a = nome variabile
5    * 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.

```
1  public class Animal {
2    private int peso;
3    ...
4    public void mangia(Animali a){
5      this.peso = this.peso + a.peso;
6    }
7  }
8
9  Cane fido = new Cane();
10 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

```
1  /* POLIMORFISMO SUBTYPING
2   * basato sui sottotipi ereditarietá */
3  Object ident (Object x){
4    return x;
5  }
6
7  /* POLIMORFISMO GENERICS (parametrico)
8   * non perdo informazioni sui tipi */
9  <T> T ident (T x){
10   return x;
```

```
11      }
12      /* questa funzione mi permette di riusare il
13       * metodo, in questo modo evito di fare CAST,
14       * 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)

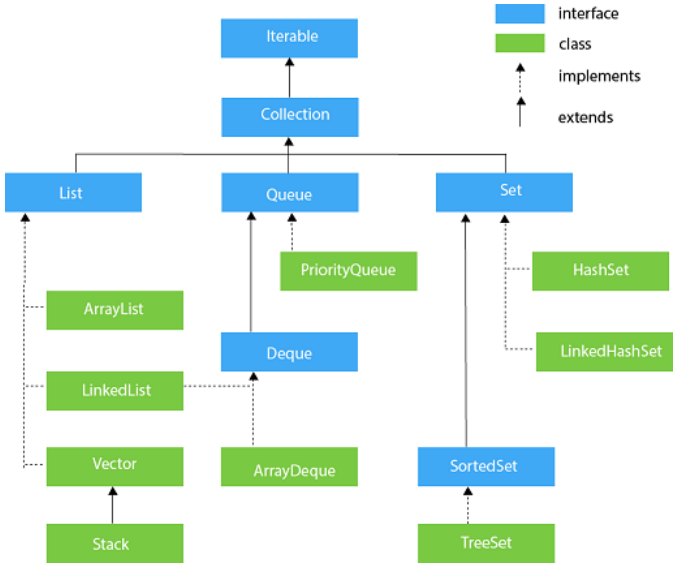
IL POLIMORFISMO è uno strumento molto utile perchè ci permette di scrivere codice, funzioni che posso adoperare anche con tipi diversi!

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*



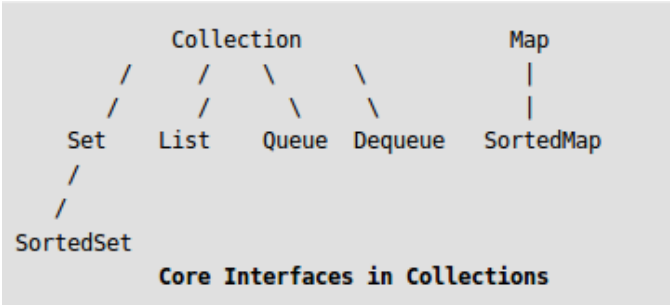
PACCHETTI JAVA
JAVA SE ⇒ Standard Edition
JAVA EE ⇒ Enterprise Edition
JAVA ME ⇒ Mobile Edition
JAVA JDK ⇒ linguaggio + tutte le librerie standard (java developement kit)
JAVA JRE ⇒ Solo a runtime, versione ridotta che serve solo a chi usa i programmi ma non al programmatore (java runtime enviroment)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



Core Interfaces in Collections

Ci riferiamo ad un oggetto usando la parola *sottotipo* quando esso è:
• O una sottoclasse (extends)
• O una sotto interfaccia (implements)
Ad esempio ArrayList ha come superclasse abstract arrayList. ArrayList implementa collection. Quindi ne è sottotipo ma non sottoclasse.

GENERICS
<? extends classe> rappresenta un tipo.
<? extends E>
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.
• Il TIPO ESTERNO gode sempre della la subsumption, • Il TIPO INTERNO non gode mai della subsumption, la subsumption si può fare solo usndo i generics(quindi con <? extends E>)
[Invarianza del subtyping]: Se ciò non fosse le subsumption funzionerebbero anche nel tipo interno 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 il problema della subsumption dei tipi interni.
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

```
1  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:

```
1  public statics void scorri(Iteratore <Integer> it){
2    while(it.hasnext()){
3      integer n = it.next();
4    }
5  }
```

Esempio di utilizzo

```
1  scorri(new Iterator<>(){
2    ...
3    ...
4    ...
5  });
```

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.

VTABLE E DIMENSIONE DI UNA CLASSE

Una virtual Table, o dispatching table, è un meccanismo utilizzato per supportare il dynamic dispatching (o anche chiamato run-time method binding).Quando una classe definisce dei metodi virtuali, il compilatore nasconde all'interno dei membri della classe una variabile che punta ad un array di puntatori a funzioni. Questi puntatori sono poi usati a runtime per invocare l'implementazione della funzione appropiata, questo perchè in compile-time non è detto che si sappia se la funzione richiamata sia quella del tipo definito o sia derivata da un'altra classe.

```
1  public static class Animale(){
2    privae int peso;
3  }
4
5  public static class Cane extends Animale{
6    private String nome;
7    public void abbaia(){};
8  }
9
10 public static class PastoreTedesco extends Cane{
11
12 }
```

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
Esistono due tipi di BINDING: static binding () e dynamic binding (un esempio è l'override dei metodi di una classe ereditata).
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 essi 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.Ricordiamo che 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
Scrivendo $C \leq A$ intendiamo che C è sottotipo di A. Definito l'operatore minore uguale passiamo alle definizioni vere e propie.

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* ci dice che il linguaggio non è COVARIANTE, in quanto i generics non cambiano.

- Esempio: ArrayList<cane> $\leq$ List<cane> (sottotipo)
- Esempio: ArrayList<cane> $\nleq$ List<Animali>

L'ultima formula non è covariante, se fosse possibile si avrebbe una doppia subsunzione sul guscio interno e sul tipo esterno

CONTROVARIANZA
Quando eredito un metodo di una classe posso sovrascriverlo usando l'overriding. Quando lo faccio un linguaggio di programmazione si dice controvariante se mi è possibile far scendere (specializzare) il tipo di ritorno, e al contempo mi è possibile far salire (rendere più generico) l'argomento di ingresso. Vediamo un esempio (*NON VALIDO IN JAVA*):

```
1  /* Data questa gerarchia di classi: */
2    public class Cane extends Animale{
3    /* nella classe Cane faccio l'override
4     * del metodo ereditato dalla classe
5     * Animale */
6    public Cane m(Cane c){
7      return c;
8    }
9    }
10
11  public class PastoreTedesco Extends Cane{
12   @Override
13   public PastoreTedesco m(Animale c){
14     return new PastoreTedesco;
15   }
16   /* rispetto al metodo originale
17    * il tipo di ritorno del metodo
18    *    (PastoreTedesco) scende (sottoclassi)
19    * il tipo del parametro di ingresso
20    *    (Animale) sale (superclasse) */
21  }
```

Detto questo è bene specificare che Java supporta i solo i tipi di ritorno controvarianti, è possibile controvariare SOLO il tipo di ritorno del metodo, ed è possibile farlo solamente scendendo (sottotipo). Ed è possibile Fare questo nella fase di overriding, non in quella di overloading.

Il seguente è l'esempio corretto in java:

```
1 public Cane m (Cane c){return c;}
2
3 @Override
4 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.
Un esempio di SOUND JAVA è il seguente:in java è possibile avvenga un segmentation fault non per un problema di casting, ma solamente se accediamo ad un indice di un array che non abbiamo allocato, quindi il programma termina a meno di una eccezione. Ci sono invece linguaggi dove non esistono gli array, quindi non accadrà mai segmentation fault e il codice terminerà sempre alla fine senza eccezioni, ovviamente senza fare i controlli di semantica.

Recentemente è stato inserito un pattern che permette anche in Java la covarianza: sono i generics con i wildcards.

```
1 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 i wildcard non sono tipi del primo ordine, infatti il *tipo* "? extends classe" non esiste!!!Non posso definire una variabile come segue:

```
1 ? extends Animale m = new Cane();
2 /* questa sintassi si puó usare solo come type argument */
```

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

9 4-03-2019

DESIGN PATTERN
• Iteratore
• Compact o Callback o Unary function

CLASSI ANONIME
Le classi *lambda* servono per fare funzioni al "volo",senza quindi avere il bisogno di implementare delle interfacce in classi separate. Vediamone un esempio:

```
interface HelloWorld {
  public void greet();
    public void greetSomeone(String someone);
}

public static void main (String args[]) {
    HelloWorld frenchGreeting = new HelloWorld() {
        String name = "tout le monde";
        public void greet() {
            greetSomeone("tout le monde");
        }
        public void greetSomeone(String someone) {
            name = someone;
            System.out.println("Salut " + name);
        }
    };
}
```

LAMBDA ASTRAZIONI
Le espressioni *lambda* servono per fare funzioni al "volo",senza quindi avere il bisogno di implementare delle interfacce in classi separate, classi anonime e classi innestate. Si ricorda che le espressioni lambda possono essere usate solo per implementare interfacce funzionali (cioè interfacce che possiedono un solo metodo astratto). Vediamone un esempio:

```
interface MyString {
  String myStringFunction(String str);
}

public static void main (String args[]) {
  MyString reverseStr = (str) -> {
    String result = "";
    return result;
  };
  reverseStr.myStringFunction("temp");
}
```

FUNZIONI DI ORDINE SUPERIORE
Sono delle funzioni che prendono delle funzioni come parametri di ingresso

```java
/* questa interfaccia é equivalente all'interfaccia java.util.Functional
 * essa infatti espone una serie di funzioni, senza implementazione, tra
 * cui anche le call-back */
public interface Func<A, B>{
  B execute(A a);

  /* questa é l'unica funzione esposta dall'interfaccia
   * un altro nome ragionevole per il metodo execute() é apply()
   * oppure call() il nome deve ricordare il fatto di richiamare
   * la funzione  */
}

/* questa funzione va ad utilizzare la funzione Func definita sopra */
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

```java
public static <A,B> List<B> map(List<A> l, Func<A,B> f){
/* dove in "public static <A,B>" dichiaro i parametri che useremo
 * mentre in "List<a> .. Func <A,B>" "uso" i parametri */
```

Funzione FILTER:

```java
/* questa funzione é simile alla funzione Func definita sopra
 * solo che rende piú chiaro il suo scopo: filtrare degli elementi
 * da aggiungere in una lista */
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;
}
```

La seguente *Filter2* NON funziona perché usa la *remove()* delle Collection, ma non è possibile rimuove un elemento in fase di scorrimento (è scritto nella documentazione)

```java
public static <A> void Filter2 (List<A>, Func<A, Boolean> p){
  for(A a: l)   /* il for each in Java essere zucchero sintattico */
    if(p.execute(a))
      l.remove(a);

}
```

Se non posso rimuovere come ho fatto sopra un elemento posso invece chiedere all'iteratore di rimuovere l'elemento stesso, esso rimuoverà quello a cui stiamo puntando. Quindi invece di usare un ciclo for come sopra, uso l'iteratore per scorrere la lista usando in metodo *hasNext*.

```java
/* questo funziona perché chiama la remove() dell'iteratore
 * mentre prima chiamavo il remove della lista */
public static <A> void Filter2 (List<A>l, Func<A, Boolean> p){
  Iterator<A> it = l.iterator();
  while(it.hasNext()){
    A a = it.next();
    if(!(p.execute(a)))
      it.remove();
  }
}

/* volendo posso usare le funzionalita delle nuove API FUNZIONALI
 * l.removeIf(a -> !p.execute(a)); */
```

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

```java
public static void main(String argv[]){
  List<String> strings = new ArrayList<>();
  string.add("ciao");
  string.add("pippo");
  string.add("unive");
  /* voglio calcolare la lunghezza della mia lista */
  List<Integer> r = map(strings, new Func<String, Integers>{

    /* "String" é la lista, "Integers" é la funzione
         * In realtá devo passare un oggetto di tipo Func<>
         * alla funzione map, perció passo una classe anonima,
         * all'interno della quale trovo solo un metodo
         * (che ha la stessa firma del metodo dell'interfaccia Func)*/

    @Override
    public Integer execute(String a){
      return a.length();
    }
  });
}
```

La seguente funzione data una lista di interi scarta gli elementi minori di zero: questo è il modo per non usare un for con un ciclo if innestato.

```
public static void main__filter(){
   List<Integer> interi  = new ArrayList<>();
   interi.add(89);
   interi.add(34);
   interi.add(-16);
   interi.add(560);
   interi.add(-1);
   interi.add(46);
   /* filter prende una lista e un predicato e produce una lista in uscita
      */
   List<Integer> l = Filter(ints, 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 Object ident__ugly(Object o) {
        return o;
    }
    /* con un metodo di subtyping che é POLIMORFISMO VERTICALE,
     * questa funzione NON é SOUND perché sono costretto a
     * fare un CAST di ció ció che ricevo */

    public static <X> X ident(X x) {
        return x;
    }
    /* con i generics, che é POLIMORFISMO PARAMETRICO, non
     * devo piú fare nessun cast. Sono sicuro del tipo di
     * ritorno */
```

```
1  public static void main__ident  (){
2    Cane fido = new Cane();
3    Cane c = (Cane) ident__ugly(fido); /* ritorna un cane
4                        * facendo un cast */
5    Cane c2 = ident(fido);  /* ritorna un cane senza dover
6                   * fare un fast */
7    Gatto g = ident(new Gatto());
8  }
```

10 7-03-2019

WILDCARDS
I tre tipi possibili di wildcards sono:
- *<?> top type* (o chiamato Unbounded Wildcard)
- *<? extends nametype>* (o chiamato Upper Bounded Wildcards)
- *<? super nametype >* (o chiamato Lower Bounded Wildcards)

Top Type
Il tipo "?" non può essere usato come tipo per una variabile quindi dichiarare"? x" non si può fare. Posso però dichiarare questo: "Object x = l1.get(..)"

```
List<?> l1 = new ArrayList<Cane>();
/* Il "?" da solo indica un tipo che gerarchicamente
 * e piú in alto di Object, viene detto top type.
 * In poche parole indica che non ci sono
 * restrizioni sul tipo di parametro passato,
 * e che quindi la lista contiene un tipo non
 * conosciuto */

l1.get(int index) /* il compilatore ritorna l'errore in
          * compile- time:  " capture of ? "
          * qualcosa che sia figlio del top type */

/* List<?> significa che creo una lista di un tipo
 * sconosciuto. In conseguenza di ció il compilatore
 * mi segnala errore quando cerco di richiamare il
 * metodo l1.add("qualcosa") perché non é di
 * tipo sconosciuto.
/*
```

Upper Bounded Wildcards
"? extends Animale" indica che come parametro posso passare un qualsiasi tipo 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)

```
  List<Animale> t3 = new ArrayList<Gatto>();
  /* é errato, si puó subscrivere solamente il guscio
   * esterno,per farlo con il guscio (tipo) interno
   * la versione corretta é fatta con il wildcard */

  List<? extends Animale> t3 = new ArrayList<Gatto>();
```

Vediamo ora un'altro esempio:

```
List<?> l1 = new ArrayList<Cane>();
List<? extends Animale> l2 = new ArrayList<Gatto>();
l1 = l2 /* é assegnazione valida */
l2 = l1 /* NON é assegnazione valida */
```

Lower Bounded Wildcards
"? super Animale" indica che come parametro posso passare qualsiasi tipo che sia più su di Animale (più generale).
In questo caso posso passare animale a tutto quello che sta sopra.
l2.add(new Animale) -> ?? non compila perchè...

Riprenderemo la map vista nella lezione scorsa.

```
public static <A,B> List<B> map(List<A> l, Func<A,B> f)();
```

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;
    }

  });
}
```

Questa funzione riportata sopra non compila in quanto i *generics* non sono soggetti alla subsumption. Per farla compilare modifichiamo la funzione map come segue:

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

}
```

Riporto anche una versione il più generale possibile:

```
1     public static <X, Y> List<Y> map(List<X> l, Func<? super X, ? extends
          Y> f) {
2         List<Y> r = new ArrayList<>();
3         for (X x : l) {
4             r.add(f.execute(x));
5         }
6         return r;
7     }
```

NESTED CLASS

```
1  public class Main__Functional {
2    public static void main__filter() {
3      ...
4    }
5  }
```

La Nested Class (o Inner Class) è totalmente senza relazione rispetto alla enclosed class (outer 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, compreso il parametro implicito this, solamente se non sono *statiche*.
Le due tipologie di classi (Inner e Outer) non servono a creare gerarchie, ma solo a creare ordine nel codice.

OVERLOADING
Il meccanismo dell'overloading permette di definire metodi con stesso nome ma firma diversa.

```
1  public static class c{
2    public int m(){
3      return 1;
4    }
5    public int m(int x){
6      return x+1;
7    }
8    public int m(float x){
9      return (int)(x-1.0f);
10   }
11   public int m(int x, int y){
12     return x+y;
13   }
14 }
```

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

-tipi
-numeri

L'overloading è del tutto gestito dal compilatore, e **non** viene quindi fatto durante la run-time.

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

Vediamo un esempio:

```
  int foo(){...};
  float foo(){...};
  ...
  ... = foo();
  /* come fa il compilatore a sapere
   * quale foo deve richiamare se
   * differiscono per il parametro di ritorno?
   * Non puó saperlo, ecco perché
   * overriding richiede i tipi di ritorno
   * uguali */
```

11 11-03-2019

WAITING FOR SOME DATA
Sembrerebbe che nessuno abbia preso appunti questo giorno

12 18-03-2019

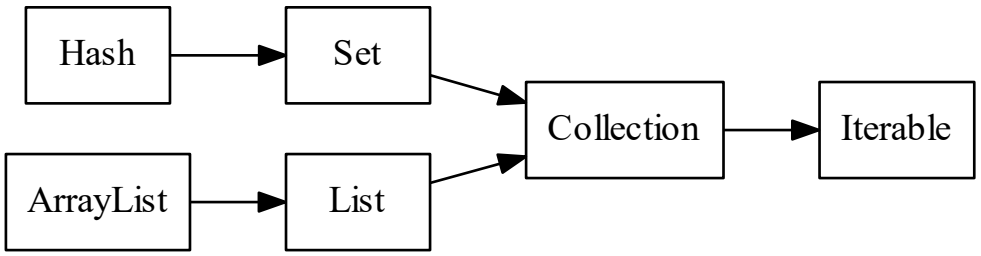
ECCEZIONI
Quando si lanciano eccezioni è bene ricordasi la differenza tra *throw* e *throws*

```
1 public void writeList() throws IOException {
2   if(true)
3       throw new IOException("demo");
4 }
```

• throws viene messa affianco alla firma del metodo, seguita da una lista dei eccezioni, e serve per dichiarare quali *TIPI* di eccezioni possono essere lanciate da un determinato metodo.
• throw seguito da un oggetto di tipo eccezione, serve per lanciare l'eccezione.
Si noti quindi che con throws si dichiarao i tipi che verranno lanciati ma poi lanciamo oggetti: questo rappresenta un tipo di subtyping.
Si possono lanciare solo eccezione figlie del tipo *Throwable*. Throwable è un super tipo di Exception e di tutte le classi lanciabili.
Come regola generale quindi throw ha bisogno di essere seguito da un'espressione con tipo compatibile per poter essere lanciata.
Il catch è lo strumento di binding per il throw.
Le eccezioni non ritornano per forza al chiamante se ritornano a chi se le prende. Infatti quando avviene una chiamata ad un metodo, questa appartiene ad una catena di chiamanti. Le eccezzioni possono quindi restituire qualcosa o al chiamante della funzione o ritornare qualcosa ad uno dei chiamanti della catena. Se risalendo questa catena l'eccezione non viene catturata nemmeno dal main questa passa direttamente alla *Java Virtual Machine*
Il sistema try-catch è stato ideato per evitare delle forti anomalie del programma.
Nel canale ufficiale del return vengo solamente ritornati i risultati "giusti", in caso contrario verrà lanciata un'eccezione: questo è lo stile richiesto per i linguaggi evoluti. Invece di complicare il tipo di ritorno usiamo le eccezioni.
Al posto delle eccezioni possiamo definire un tipo di ritorno che codifica il fatto che hai trovato o meno quello che cercavi. Questa tecnica non è molto leggibile per chi non ha scritto il codice, sarebbe meglio usare il design pattern " tipo-eccezione". Esso è utile anche perchè in questo modo non si può scrivere codice che non funzioni, mentre definendo un nuovo tipo è possibile.

13 21-03-2019

• -> ITERABLE:Posso solo scorrere
• —> COLLECTION: Posso scorrere, aggiungere e togliere
• ——> LIST: Posso scorrere e aggiungere o togliere con un indice
• ———-> ARRAYLIST
ArrayList è sottotipo di list, in quanto è una classe che implementa list!



Caratteristiche dell'interfaccia set:
• Gli elementi non sono duplicati
• Gli elementi non sono ordinati in base a come sono inseriti
• Non vengono inseriti metodi nuovi, eredita solo quelli del padre
• I metodi nuovi vengono messi nella classe che implementa l'interfaccia

Per evitare di riprodurre codice si usano le classi astratte dalle quali poi si erediterà.

Relazione di ordinamento: operatore binario che permette di mappare elementi di due insiemi diversi.

Una classe con metodi tutti statici non si può costruire. Rappresenta dunque un contenitore di metodi (è un pezzo di libreria).

Il professore questo giorno ha caricato su github del codice chiamato: TinyJDK. Lo riporto per completezza:

```
1 /* Classe: MyIterable.java */
2 public interface MyIterable<E> {
3
4     MyIterator<E> iterator();
5     int find(E x) throws Exception;
6
```

```
}
```

```
/* Classe: MyIterator.java */
public interface MyIterator<E> {
    boolean hasNext();
    E next();
}
```

```
/* Classe: MyCollection.java */
import java.util.Collection;
import java.util.function.Function;

public interface MyCollection<T> extends MyIterable<T> {
    void add(T x);
    void clear();
    void remove(T x);   // da decidere se ci piace o no
    boolean contains(T x);
    boolean contains(Function<T, Boolean> p);
    int size();


}
```

```
/* Classe: MyList.java */

public interface MyList<T> extends MyCollection<T> {
    void add(int i, T x);
    T get(int i);
    void set(int i, T x);
}
```

```
/* Classe: MySet.java */
public interface MySet<T> extends MyCollection<T> {
}
```

```
/* Classe: MyArrayList.java */
import java.util.Collection;
import java.util.function.Function;

public class MyArrayList<T> implements MyList<T> {

```

```
7       private Object[] a;
8       private int actualSize;
9
10      public static class MyException extends Exception {
11          public MyException(String s) {
12              super(s);
13          }
14      }
15
16  /*    T[] toArray() {
17          return (T[]) a;
18      }*/
19
20  /*    public static Exception returnNow() {
21          return new Exception("msg");
22      }
23
24      public static void throwNow() throws Exception {
25          throw new Exception("msg");
26      }
27
28      public static void caller() throws Exception {
29          Exception e = returnNow();
30          throwNow();
31      }
32
33      public static void caller2() {
34          try {
35              caller();
36          }
37          catch (Exception e2) {
38              // fai qualcosa con e2
39          }
40
41      }
42
43      public void m(int x) throws Exception {
44          MyException e = new MyException("error messagge");
45          if (x < 0) throw e;
46      }
47    */
48
49      public MyArrayList() {
50          clear();
51      }
52
53      public static class NotFoundException extends Exception {
54      }
55
56      @Override
57      public int find(T x) throws NotFoundException {
58          int cnt = 0;
59          MyIterator<T> it = iterator();
```

```
60          while (it.hasNext())
61          {
62              T y = it.next();
63              if (x.equals(y)) return cnt;
64              ++cnt;
65          }
66          throw new NotFoundException();
67      }
68
69
70
71
72      public static void main3() {
73          MyArrayList<Integer> c = new MyArrayList<>();
74          try {
75              int index = c.find(6);
76              System.out.println("found at index = " + index);
77          } catch (NotFoundException e) {
78              try {
79                  int index = c.find(7);
80              } catch (NotFoundException e1) {
81
82              }
83
84          }
85      }
86
87      @Override
88      public boolean contains(T x) {
89          for (int i = 0; i < actualSize; ++i) {
90              Object o = a[i];
91              if (o.equals(x)) return true;
92          }
93          return false;
94      }
95
96
97      @Override
98      public boolean contains(Function<T, Boolean> p) {
99          return false;
100     }
101
102     @Override
103     public int size() {
104         return actualSize;
105     }
106
107
108     @Override
109     public void clear() {
110         a = new Object[100];
111         actualSize = 0;
112     }
```

```
113
114     @Override
115     public void add(T o) {
116         a[actualSize++] = o;
117         if (actualSize >= a.length) {
118             Object[] u = new Object[a.length * 2];
119             for (int j = 0; j < a.length; ++j)
120                 u[j] = a[j];
121             a = u;
122         }
123     }
124
125     @Override
126     public MyIterator<T> iterator() {
127         return new MyIterator<T>() {
128             private int pos = 0;
129
130             @Override
131             public boolean hasNext() {
132                 return pos <= actualSize;
133             }
134
135             @Override
136             public T next() {
137                 return (T) MyArrayList.this.a[pos++];
138             }
139         };
140     }
141
142     @Override
143     public void add(int i, T x) {
144
145     }
146
147     @Override
148     public T get(int i) {
149         return (T) a[i];
150     }
151
152     @Override
153     public void set(int i, T x) {
154         a[i] = x;
155     }
156
157     @Override
158     public void remove(T x) {
159
160     }
161
162 }
```

```java
/* Classe: MyArrayListSet.java */
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.function.Function;

public class MyArrayListSet<T extends Comparable<T>> implements MySet<T> {
    private final Comparator<T> p;
    private final ArrayList<T> a;

    public MyArrayListSet(Comparator<T> p) {
        this.a = new ArrayList<T>();
        this.p = p;
    }

    @Override
    public void add(T x) {
        if (!contains(x)) {
            a.add(x);
            sort();
        }
    }

    private void sort() {
        Collections.sort(a, p);
    }

    @Override
    public void clear() {
        a.clear();
    }

    @Override
    public void remove(T x) {
        a.remove(x);
    }

    @Override
    public boolean contains(T x) {
        return a.contains(x);
    }

    @Override
    public boolean contains(Function<T, Boolean> p) {
        return a.contains(p);
    }

    @Override
    public int size() {
        return a.size();
    }

```

```
54      @Override
55      public MyIterator<T> iterator() {
56          return a.iterator();
57      }
58
59      @Override
60      public int find(T x) throws Exception {
61          return a.find(x);
62      }
63  }
```