Longo Giuseppe
175983

Progetto Liste – POO

- **Traccia**

PRIMA PARTE

Sono assegnate le seguenti interfacce appartenenti al package *poo.util*, che si ispirano alle omonime interfacce del *Collections Framework* di Java (package *java.util*), di cui mantengono gran parte dei metodi e con lo stesso significato, tipi di eccezioni etc.

```
package poo.util;
public interface Collection<T> extends Iterable<T>{
    boolean add( T e );
        boolean addAll( Collection<T> c );
        void clear();
        boolean isEmpty();
        boolean contains( T e );
        boolean containsAll( Collection<T> c );
        boolean remove( T e );
        boolean removeAll( Collection<T> c );
        boolean retainAll( Collection<T> c );
        int size();
        Object[] toArray();
        <E> E[] toArray( E[] a );
}//Collection
```

```
package poo.util;
import java.util.ListIterator;
import java.util.Comparator;
public interface List<T> extends Collection<T>{
        void add( int indice, T e );
        void addAll( int indice, Collection<T> c );
        T get( int indice );
        int indexOf( T e );
        int lastIndexOf( T e );
        ListIterator<T> listIterator();
        ListIterator<T> listIterator( int pos );
        T remove( int indice );
        T set( int indice, T e );
        void sort( Comparator<T> c );
        static <T> int binarySearch( List<T> l, T e ){/* TODO come parte progetto */}
}//List
```

Sfruttando la possibilità dei metodi default e metodi statici consentiti da Java 8 nelle interfacce, concretizzare, con l'aiuto degli iteratori, quanti più metodi è possibile direttamente nelle due interfacce Collection<T> e List<T>. Metodi che non possono essere concretizzati in Collection, concretizzarli, ove possibile, come parte dello sviluppo della classe astratta *AbstractCollection<T>*. Sebbene in *java.util* il lavoro di pre-implementazione di metodi è classicamente affidato alle due classi astratte: *AbstractCollection<T>* e *AbstractList<T>* (quest'ultima implementa *List<T>* ed estende *AbstractCollection<T>*), in questo progetto evitare l'introduzione di *AbstractList<T>* e limitarsi all'uso di metodi default nelle interfacce e alla sola classe astratta *AbstractCollection<T>* già citata.
Sviluppare quindi le classi concrete *ArrayList<T>* e *LinkedList<T>* che implementano *List<T>* ed estendono *AbstractCollection<T>*. La classe *ArrayList<T>* deve esporre i costruttori:

```
public ArrayList() //costruttore di default
public ArrayList( int capacita ) //costruttore normale
```

Longo Giuseppe
175983

La classe *LinkedList<T>* può basarsi sul solo costruttore di default. In più, la classe *LinkedList<T>* deve rendere disponibili altresì i metodi (con le relative eccezioni):
void addFirst(T e);
void addLast(T e);
T removeFirst();
T removeLast();
T getFirst();
T getLast();

Nel progetto di *ArrayList<T>* e *LinkedList<T>* rilevante è la realizzazione delle strutture di iterazione. Si ricorda che un iteratore, quando non è al di fuori della lista, si trova logicamente *tra* due elementi consecutivi e non *su* un elemento. Al fini di identificare correttamente, quando esiste, l'elemento corrente (definito da *previous() o next()*), può essere opportuno introdurre una enumerazione che esprime i possibili movimenti in avanti (*FORWARD*), indietro (*BACKWARD*) o l'assenza di movimento (*UNKNOWN*), e mantenere l'ultimo movimento effettuato con l'iteratore.

Il metodo *sort()* riceve un oggetto *Comparator<T>* e ordina, ad esempio con l'algoritmo *Bubble Sort* la lista.

La prima parte del progetto termina dopo aver testato "accuratamente" il corretto funzionamento delle classi *ArrayList<T>* e *LinkedList<T>.* Si suggerisce di predisporre una classe Main col metodo main che realizza "tutte" le condizioni di funzionamento della lista. Nei casi di dubbio, confrontarsi con le implementazioni di *java.util.ArrayList<T>* e *java.util.LinkedList<T>.* Come altro test, provare a rimpiazzare nelle classi *PolinomioAL* e *PolinomioLL* l'uso di *ArrayList/LinkedList di Java*, con le classi ottenute in questo progetto.

PARTE SECONDA

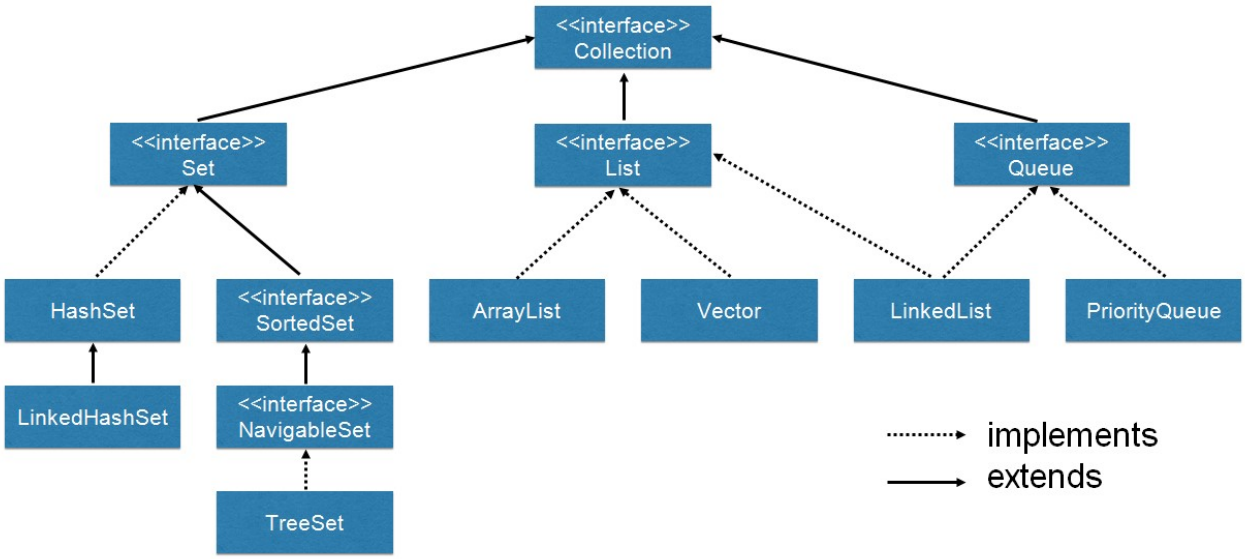
Successivamente realizzare una GUI che espone una struttura a menù e consente inizialmente di scegliere il tipo di lista concreta se ArrayList (con capacità di default o una specificata dall'utente) o LinkedList, ed il tipo degli elementi tra intero o String. La GUI deve permettere quindi di evocare tutte le operazioni sulle liste e mostrare su una text area lo stato corrente della lista, es. come [12, 4, 5, 15], o l'eccezione sollevata dall'ultima operazione. Quando si chiede di inserire un elemento (o rimuoverlo etc.) subito dopo occorre visualizzare sulla text area il nuovo contenuto della lista. Quando si accende un iteratore, visualizzare la posizione (^) dove si trova l'iteratore, facendo next si deve mostrare la nuova posizione mentre in un campo di testo Corrente si mostra l'elemento corrente o ? se esso non è definito. Si nota che scegliendo il comando iterator, la freccia si trova inizialmente prima del primo elemento, e risultano abilitati i comandi hasNext e next. Dopo una next() che ha successo, si abilita la remove() etc. Se invece si accende un ListIterator, si può specificare la posizione iniziale del list iterator come quella di default (identica a quella di iterator) o si può fornire un indice intero (compreso tra 0 e size()) che individua l'elemento della lista prima del quale va piazzata la freccia (specificando size(), la freccia va posta dopo l'ultimo elemento, specificando size()-1, la freccia va posta tra penultimo ed ultimo etc.). Con ListIterator devono poter essere accessibili tutti i metodi di ListIterator. Naturalmente, occorre garantire che di momento in momento risultano abilitati tutti e soli i menu item che rappresentano operazioni ammissibili nello stato attuale della lista.

Longo Giuseppe
175983

- **Relazione**

L'idea iniziale dietro alla implementazione di quanto definito nella richiesta, è stata innanzitutto quella di definire un package *ProgettoEsame* nel quale inserire le classi esplicitate nella prima parte della traccia.
In seguito, ho provveduto a creare un ulteriore package inserito dentro *ProgettoEsame* nel quale stendere tutte le classi destinate alla realizzazione della seconda parte del progetto (*CollectionTestGUI*).

La prima parte del progetto prevede la stesura di due classi già note da *java.util* e che risultano essere di utilità fondamentale nell'ambito della POO attuale: *ArrayList e LinkedList*.
Ecco qui introdotta la precisa strutturizzazione della interfaccia *Collection*, definita nel così detto *Collections Framework* , nel quale difatti le classi appena citate risultano essere il concretizzarsi di interfacce e classi astratte:



Come è possibile evincere dal diagramma qui di sopra, l'interfaccia *List* va ad estendere l'interfaccia principale che è quella *Collection*, iterabile e ovviamente di tipo generico; da qui *LinkedList* e *ArrayList* risulteranno implementare sia *List* sia *Collection* per polimorfismo ed ereditarietà.
In sé il Collection Framework si occupa appunto di permettere una rigida gestione di collezioni di oggetti (ovviamente pure quelli dati dalle classi *wrapper*) che in base al tipo di struttura dati utilizzata, possono essere ordinati, eleminati, modificati, ricercati in modo efficiente ma sempre in relazione a ciò che si è in procinto di programmare.
Se l'interfaccia *List* diviene fondamentale quando bisogna lavorare attraverso l'indicizzazione del contenuto, il *Set* invece prevede un insieme (matematico) di oggetti che può essere ordinato o meno (*TreeSet*) o istanziato sfruttando *l'hashing* per sistemare gli oggetti.
Inoltre, una *collection* può essere esplicitata anche come *Queue*, o meglio coda, che attua la politica FIFO (*first input first output*) e che per questo motivo può vedere una sua concretizzazione anche nella *LinkedList*.
Come ultima precisazione, occorre specificare che generalmente ogni classe concreta fa appoggio a una classa astratta che si presenta come "collante" tra una interfaccia e le sue varie concretizzazioni (*AbstractList*, *LinkedList*, ecc.). Questo permette non solo un strutturizzazione molto più ordinata, ma consente maggiori opportunità di polimorfismo oltre a risultare necessario per l'implementazione di alcuni metodi.

Infine risulta poi importante dire che il *Collections Framework* non si limita alla sola interfaccia *Collection*, ma prevede anche una seconda ”macro”-interfaccia che prende il nome di *Map* e che si accosta alla sua più immediata implementazione, la classe concreta *HashMap* che mappa appunto chiave-valore attraverso *l'hashing*.
Per concludere, sulle linee guida segnalate dalla traccia, in questa relazione mi occuperò di definire i ragionamenti logici che mi hanno portato alla stesura dei vari metodi, classi, interfacce, nonché l'organizzazione strutturale della prima e seconda parte del progetto. Inoltre, cruciale è stato attingere alla organizzazione di tali metodi all'interno di *java.util*, in particolare per la gestione delle varie eccezioni da sollevare.

- ## Collection.java

L'interfaccia Collection, pubblica ovviamente, risulta essere come richiesto, generica in T e *Iterable*.
Tra i metodi presenti, l'add(T e) è l'unico vincolato da una implementazione nelle classi erede ed è perciò stato esclusivamente dichiarato, insieme a clear() e size(). Infatti ho preferito consegnare anche questi ultimi due direttamente alle classi concrete per promettere così una reale efficienza in relazione alla struttura dati implementata, trattandosi comunque di metodi fortemente utilizzati all'interno di altri. Difatti avremo che col metodo *clear* si porterà la collezione allo stato iniziale, mentre il metodo *size* andrà a ritornare la dimensione effettiva della collezione (diversa dalla capacità).
Cardine è stata l'iterabilità di Collection per l'implementazione del resto dei metodi all'interno della stessa interfaccia, sfruttando il particolare modificatore *default* introdotto a partire da Java 8: *addAll, contains, containsAll, remove, removeAll e retainAll* sfruttano *for-each* e *iterator* per ritornare quanto richiesto da ogni metodo. Analizziamo alcuni di questi…

```
default boolean containsAll( Collection<T> c ){
        if ( c==null ) throw new NullPointerException();
        if ( this.size() < c.size() ) return false;
        Iterator<T> it = c.iterator();
        while ( it.hasNext() ){
                if ( !this.contains(it.next()) ) return false;
        }
        return true;
}//containsAll
```

Il metodo containsAll(Collection<T> T c) ha un parametro formale identificato come un'altra collection anch'essa generica in T e quindi dello stesso tipo di *this* definito in *runtime*. Dopo aver verificato esplicitamente che l'oggetto c risulti essere stato inizializzato, il containsAll predispone un controllo di compatibilità tra le size di entrambe le collection, così da uscire direttamente dall'algoritmo senza avviare cicli in caso *false*. In seguito, si attiva un iteratore su c che scansiona ogni elemento e lo mette a confronto con tutti gli elementi di this; per fare ciò ho preferito avvalermi del metodo contains(T e) già definito precedentemente. Questo ha comunque ispirato lo stesso containsAll che si avvale però di un ben più rapido *equals* per il confronto diretto tra i due elementi (dello stesso tipo). Inoltre, così come java stesso definisce, questo metodo potrà ritornare true anche se *c.size()* < *this.size,* confrontando così esclusivamente quella porzione di c delimitata dalla dimensione di *this*.

```
default boolean retainAll( Collection<T> c ){
        if ( c==null ) throw new NullPointerException();
        int sizePrima = this.size();
        Iterator<T> it = this.iterator();
        while ( it.hasNext() ){
                T e= it.next();
                if ( !c.contains(e) ){
                        it.remove();
                }
        }
        return this.size() != sizePrima;
}//retainAll
```

Anche il metodo retainAll (Collection<T> c) sfrutta la versatilità del *contains* ma non in modo del tutto analogo. Una volta verificato che c sia stata correttamente istanziata, definisco una variabile locale *sizePrima* che alla fine del metodo andrò a confrontare con this.size() per verificare che lo stesso abbia sortito effetto (così come definito in java.util). A questo punto attivo un iteratore su this e per ogni elemento catturato da it.next() vado a verificare che la collezione c lo contenga (sfruttando contains (T e)) : in caso negativo, sulla logica del *retain,* questo sarà eliminato.

Specifiche considerazioni invece vanno esplicitate per i due metodi *toArray*. Questi sfruttano una medesima logica di ragionamento: si istanzia l'array di destinazione e con un for-each su this, si va a popolare in modo crescente (avvalendosi di un indice) l'array appena creato, "castizzando" ogni oggetto al tipo richiesto.
Ma particolare è la situazione del <*E*> *toArray (E[] a*), il quale permette di esportare la collezione sotto forma di un array generico in E, quindi possibilmente diverso da T e già istanziato. Difatti, una volta verificata la *NullPointerException()* dell'array a, si procede con il suo popolamento. In particolare, per poter sollevare *l'ArrayStoreException(),* ho visto necessario l'utilizzo del costrutto try-catch che gestisse correttamente come definito da java, l'incompatibilità dei tipi utilizzati durante il *casting*. Inoltre il metodo stesso prevede che la copia per riferimento del casting venga effettuata su tutto l'array a, sovrascrivendo elementi già esistenti; eccezione non verrà sollevata se non tutta la collection this sarà consegnata all'array da ritornare.

```
default <E> E[] toArray( E[] a ){
        if ( a==null) throw new NullPointerException();
        int pos = 0; E elem;
        for ( T e : this){
                try{
                        elem = (E)e;
                }catch ( ClassCastException ex){ throw new ArrayStoreException(); }
                if ( pos<a.length ){
                        a[pos] = elem;
                        pos++;
                }
                else break;
        }
        return a;
}//toArray (generico in E)
```

Longo Giuseppe
175983

- ## AbstractCollection.java

AbstractCollection si presenta come una classe astratta appunto, pubblica e anch'essa generica in T, dovendo comunque implementare l'interfaccia Collection. Prima di Java 8, il modificatore default non era stato ancora introdotto e quindi era impossibile poter implementare dei metodi nell'interfaccia: ecco che divenivano di basilare importanza le classi astratte. Occorre notare come però alcune convenzioni di priorità (per logica di ereditarietà) siano state mantenute dato che Java non permette l'implementazione dei metodi *equals*, *toString* e *hashCode* come default. Ecco che la AbstractCollection è concepita esclusivamente per l'implementazione di questi tre metodi.

- ## List.java

L'implementazione della interfaccia List segue anche la gerarchia espressa dal *Collections Framework*: definire una serie di metodi destinati a collezioni si iterabili ma anche indicizzate.
Di basilare importanza, per riuscire a istanziare ogni metodo richiesto come default, è stato dichiarare i metodi *listIterator()* e *listIterator(int pos)* che definiscono una lista iterabile anche con un iteratore molto più completo, in grado di spostarsi sia in avanti sia indietro, aggiungere e settare elementi e tutto ciò sfruttando gli indici caratteristici delle liste. Inoltre, proprio a favor di ciò, è stata introdotto sempre in List, una istanza di tipo enumerativo *Spostamento*: essa vedremo risulterà di fondamentale importanza per una efficiente gestione dei metodi di *remove* e *set*.
A primo impatto risulta evidente l'istanziazione di quattro metodi destinati all'ordinamento e alla ricerca binaria (due per ognuno). Difatti venendo richiesto dalla traccia l'implementazione di un metodo non presente "ufficialmente" in List (binarySeach, definito nella classe di utilità Collections, facente sempre parte del Collections Framework), mi son permesso di aggiungerne altri due, che a mio parere vanno a completare l'usabilità effettiva di entrambi: *binarySearch(Comparator<T> comp, T e) e <E> void sort(List<E> l*) (con E oggetto comparabile ovviamente). Ciò che si evince dal complesso è che sarà possibile invocare il metodo di sort o di ricerca binaria, sia attraverso l'utilizzo di oggetti costituiti da tipi necessariamente comparabili sia con oggetti di qualsiasi genericità senza restrizioni ma implementando dei metodi *comparator*.
Altro elemento cruciale che sta dietro alla logica della mia implementazione, è stato l'introduzione di una nuova classe composta da soli metodi statici definita come *RicercaEOrdinamento.java.* Questa raccoglie tutti i metodi effettivi di ricerca e ordinamento, appunto, affiliati a quanto detto fin ora. L'effettiva utilità di questa classe di servizio, può manifestarsi verso molteplici fronti in realtà: innanzitutto mi ha consentito di gestire liberamente i vari algoritmi con metodi private e package che implementati invece nella interfaccia sarebbero stati pubblici. Difatti l'unica alternativa per gestire il tutto sarebbe stata inserirli in una classe astratta, ad esempio *AbstractList,* andando però contro gli stessi "bound" definiti dalla traccia.
Un'altra agevolazione che può essere concessa da questa "suddivisione" si manifesta con la possibilità per il programmatore di introdurre facilmente altri algoritmi alla base dell'ordinamento e della ricerca, vedi ad esempio ricerca lineare o *bubble sort* al posto o in aggiunta a ricerca binaria e *merge sort.*

```
static <E extends Comparable<? super E>> int binarySearch( List<E> l, E e ){
        if ( l==null ) throw new IllegalArgumentException();
        return RicercaEOrdinamento.binarySearch(l, e, 0, l.size());
}//binarySearch (ordinamento naturale)
…
static <T extends Comparable<? super T>> int binarySearch(List<T> l, T elem, int in, int fin){
        int med = (in+fin)/2;
```

```
        if ( med<=0 || med>=l.size() ) return -1;
        if ( elem.equals(l.get(med)) ) return med;
        if ( elem.compareTo(l.get(med)) > 0) return binarySearch(l,elem,med+1,fin);
        else return binarySearch(l,elem,0,med-1);
} binarySearch (ordinamento naturale)
```

Prendendo in esame, ad esempio, uno dei due metodi binarySearch, vedremo come il modificatore static derivi proprio dal fatto che risulta necessario consegnare al metodo una List composta da elementi *Comparable* che potranno così procedere con un confronto naturale. Inoltre ho predisposto, sulle linee richieste, che il generico E debba essere comparabile o su se stesso o sfruttando un *compareTo* già definito nel suo possibile "padre" (classe super). Per quanto riguarda il metodo di ricerca binaria in sé invece, è palese sia stata sfruttata una strutturizzazione ricorsiva, la quale varierà nella versione con *comparator* solamente per l'utilizzo del metodo *compare* al posto del *compareTo* e per la richiesta di un ulteriore parametro formale rappresentato ovviamente dal comparator. Da sottolineare inoltre, risulta il fatto che non è stato introdotto nessun metodo di controllo per verificare che la lista sia stata precedentemente ordinata in ordine crescente, quindi si consegna questa discrezione direttamente al programmatore (senza ordinamento si perderebbe completamente il guadagno in termini di complessità generati dal metodo).

Per quanto riguarda gli altri metodi, tutti seguono più o meno la stessa logica: inizializzano un *listIterator* in una posizione specifica e da qui eseguono le operazioni richieste.

```
default void add( int indice, T e ){
        if ( indice < 0 || indice >= this.size() ) throw new IndexOutOfBoundsException();
        ListIterator<T> it = this.listIterator(indice);
        it.add(e);
}//add

default void addAll( int indice, Collection<T> c ){
        if ( indice < 0 || indice >= this.size() ) throw new IndexOutOfBoundsException();
        if ( c==null ) throw new IllegalArgumentException();
        ListIterator<T> it = this.listIterator(indice);
        for ( T e : c )
                it.add(e);
}//addAll
```

Difatti, il metodo *add* non fa altro che sfruttare l'add previsto dalla classe *ListIterator,* utilizzando l'indice dato dal parametro formale per definire la posizione dalla quale inizializzarlo. Analogamente, il metodo addAlll sfrutterà lo stesso *add* per le singole aggiunte, accostato però da un for-each per scandire la collection c (la cui corretta inizializzazione è stata già verificata). Inoltre, mi è doveroso specificare come il controllo per alcune eccezioni sia stato attuato per una più immediata comprensione del codice, dato che alcune di queste vengono comunque sollevate o controllate in altri algoritmi (vedi, ad esempio, l'iniziazione del listIterator).

Comunque anche il metodo *get* si basa sulla stessa politica di add e addAll, ma non fa altro che andare a ritornare l'elemento nella immediata posizione successiva (difatti il listIterator inizializza il cursore a indice-1, poiché il conteggio delle posizioni inizia da prima il primo elemento della lista che stiamo iterando).

Con il metodo *indexOf* invece, ho optato per l'utilizzo di un iteratore che ovviamente va a iterare su *this*. Per poter ritornare l'indice della posizione dell'oggetto, inizialmente avevo pensato di definire un metodo *getIndiceListIterator* che ritornasse appunto l'indice del cursore già utilizzati per definire i metodi *nextIndex* e *previousIndex* che vedremo nel *listIterator.* Alla fine comunque ho optato verso una strada molto più schematica introducendo un contatore *ris* che si incrementa ad ogni *next:* nel momento in cui *l'equals* dà *true,* il programma termina e viene ritornato ris (se l'elemento non sarà presente nella lista, l'output dell'algoritmo sarà -1).

```
default int indexOf( T e ){
        int ris = 0;
        Iterator<T> it = this.iterator();
        while ( it.hasNext() ){
                T elem = it.next();
                if ( elem.equals(e) ) return ris;
                ris++;
        }
        return -1;
}//indexOf
```

Una pressoché analoga concezione ho ideato per il metodo *lastIndexOf* che ha invece ritorna l'indice dell'ultima ricorrenza dell'elemento estratto dal parametro formale. In questo caso si è proceduto a instanziare sempre localmente, un *listIterator* inizializzato all'ultima posizione disponibile (size-1). Qui il contatore *ris* sarà decrementato seguendo i *previous* all'interno del ciclo: trovato l'elemento, si ritorna *ris* (anche qui il risultato potrà essere -1 se l'elemento passato non sarà mai trovato).

```
default int lastIndexOf( T e ){

        int ris = this.size();
        ListIterator<T> it = listIterator(ris);
        while ( it.hasPrevious() ){
                T elem = it.previous();
                if ( elem.equals(e) ) return ris;
                ris--;
        }
        return -1;
}//lastIndexOf
```

Per quanto riguarda invece i metodi *remove* e *set*, essi sono stati ragionati in modo prettamente analogo: anzi, l'unica basilare differenza si basa sul fatto che uno deve rimuovere l'elemento e l'altro modificarlo. Per questo motivo mi sono avvalso dell'utilizzo di un *listIterator* inizializzato alla posizione definita dal parametro *indice,* dopo aver controllato che questo non andasse oltre la lunghezza effettiva di *this.* A questo punto definisco una variabile locale nella quale salvare il risultato da porre in output dato dalla *next* dell'iteratore (difatti entrambi i metodi richiedono che sia ritornato l'elemento antecedente alla modifica) ed eseguo rimozione o settaggio sfruttando appunto i metodi già disponibili nel *listIterator.*

```
default T remove( int indice ){
        if ( indice<0 || indice>=this.size() ) throw new IndexOutOfBoundsException();
        ListIterator<T> it = listIterator(indice);
        T ris = it.next();
        it.remove();
        return ris;
}//remove

default T set( int indice, T e ){
        if ( indice<0 || indice>=this.size() ) throw new IndexOutOfBoundsException();
        ListIterator<T> it = listIterator(indice);
        T ris = it.next();
        it.set(e);
        return ris;
}//set
```

Introduco ora le logiche dietro alla implementazione delle due classi concrete previste dalla traccia: *ArrayList* e *LinkedList*, sempre a carattere generico. Entrambe sono articolate da una struttura dati pienamente diversa se non opposta, ma comunque accomunati dall'idea di avere un "padre" di tipo List: quindi è proprio il polimorfismo che permette loro di avere a disposizione tutti i metodi che abbiamo previsto sia per *Collection* e *AbstractCollection*, sia per *List*. Ne deriva conseguentemente che entrambe le classi saranno obbligate ad implementare tutti quei metodi che nelle interfacce erano stati esclusivamente dichiarati.

Inoltre essendo stata richiesta la gestione dell'eccezione *ConcurrrentModificationException,* ho proceduto, in entrambi gli eredi, così come avviene canonicamente, con l'introduzione di un contatore modifiche *concurrentALCounter* che al momento dell'attivazione di un iteratore (iterator o listIterator) potrà consentire di sollevare questa eccezione. Lo scopo è evitare che il programmatore, una volta applicata qualche iterazione, possa tornare ad utilizzare lo stesso oggetto su una lista che magari ha subito modifiche da metodi al di fuori dell'iteratore attivo, così da non incombere a successivi drastici errori.

Considerazioni diverse vanno perciò sottolineate in relazione all'implementazione degli iteratori, tali *iterator e listIterator*.

```
@Override
public Iterator<T> iterator() {
        return new myIterator();
}//iterator


@Override
public ListIterator<T> listIterator() {
        return new myListIterator();
}//listIterator
```

Entrambi prevedono l'introduzione di una classe *private* predisposta prontamente a fini di utilità: *myIterator* andrà ad implementare l'interfaccia *Iterator*, mentre *myListIterator* implementerà *ListIterator.*
Difatti, avendo dichiarato inizialmente che l'interfaccia *Collection extends Iterable*, si era reso necessario definire delle classi che istanziassero tutti i metodi richiesti dagli iteratori, implementando di conseguenza le rispettive interfacce coi metodi da definire in *override.* Ed ecco qui che i metodi pubblici iterator e *listIterator* ritorneranno un nuovo oggetto di tipo, rispettivamente, iterator e *listIterator*, appunto. Ovviamente la logica dietro alla implementazione di ciascuno di essi in ciascuno degli eredi sarà completamente diversa, dal momento che entrambi si fondano su strutture dati estremamente diverse.

- ## ArrayList.java

ArrayList rappresenta una delle due classi concrete prevista dalla traccia. Si tratta di una struttura dati basata sull'utilizzo di un array che verrà gestito in modo da adattarsi alle varie configurazioni previste da una *list*.
La sua reale efficienza si manifesta principalmente attraverso la rapidità d'accesso con cui è possibile pervenire a ciascun elemento, così come avviene in un array appunto: ad esempio, il metodo di *get(int indice)* si risolverà con una complessità pari a O(1) mentre in una lista linkata la complessità si ridurrà sempre a O(n) (con n numero di iterazioni per raggiungere l'elemento). Sulla base di ciò mi son quindi permesso di andare a

ridefinire in *override*, alcuni metodi che sfruttando questa eccezionale proprietà dell'array, riducendo pesantemente la loro complessità, quali appunto *get* e *set*.
Oltre all'introduzione delle varie variabili di istanza, ho provveduto poi alla creazione di due costruttori: uno di *default,* che inizializza l'array con una dimensione prestabilita (ho scelto 10 così come fa Java nella sua implementazione) e un altro in cui è concesso al programmatore di istanziare l'array con una dimensione scelta arbitrariamente (ovviamente compresa tra 0 e *MAX_VALUE*).
Fulcro di questa struttura dati mi è risultata essere stata l'introduzione di un metodo *resize* che sfruttando il la funzione della classe di utilità *Arrays*, *copyOf*, mi ha permesso una rapida gestione del ridimensionamento dell'array alla base della struttura: la sua dimensione difatti verrà raddoppiata nel momento in cui è stato occupato completamente, oppure verrà dimezzata se l'array risulta essere occupato per metà.

Tra i metodi previsti obbligatoriamente dalle interfacce, posso ad analizzare *add*, *iterator* e *listIterator*.

```
@Override
public boolean add(T e) {
        size++; resize();
        concurrentALCounter++;
        contenuto[size-1] = e;
        return true;
}//add
```

Ovviamente, il metodo *add* è predisposto all'aggiunta di un elemento in coda alla lista. Cruciale però diviene innanzitutto il *resize* iniziale introdotto da un incremento della *size*: solo così difatti sarà possibile gestire la capacità effettiva della lista per dar posto a un nuovo elemento. Inoltre, così come in ogni altro metodo, ad ogni operazione ci si predisporrà all'incremento del contatore desinato a sollevare la *ConcurrentModificationException.*

```
private class myIterator implements Iterator<T>{

        private int concurrentCounter = concurrentALCounter;

        private int cur = -1;
        private boolean rimuovibile = false;

        ...

}//myIterator
```

Ecco qui introdotta *myIterator*, quella classe privata che come citato precedentemente, si predispone a contenere tutti i metodi che concorrono al reale funzionamento dell'oggetto *iterator: hasNext*, *next* e *remove.*

```
@Override
public boolean hasNext() {
        if ( concurrentALCounter!=concurrentCounter )
                throw new ConcurrentModificationException();
        if ( cur==-1 ) return size>0;
```

```
        return cur < size-1;
}//hasNext
```

È nel metodo *hasNext* che ho predisposto la verifica della ConcurrentModificationException: se i due contatori risulteranno diversi, ciò significherà che sono stati invocati dei metodi che hanno modificato gli elementi dell'array-list dopo l'inizializzazione dello stesso (sia metodi propri dell'array-list sia di un altro iterator). Questo controllo verrà effettuato solo in questo metodo, poiché sia *next* sia *remove* verranno ricondotti sempre ad eseguire un *hasNext.* Infatti, il metodo *next*, che dovrà ritornare l'elemento già "oltrepassato" dal cursore, effettuerà un *hasNext* per verificare che vi siano ancora elementi e in caso contrario, sollevare una specifica eccezione *NoSuchElementException.*

```
@Override
public T next() {
        if ( !hasNext() ) throw new NoSuchElementException();
        cur++;
        rimuovibile = true;
        return contenuto[cur];
}//next
```

A questo punto si procede con l'incremento del cursore (inizializzato a -1) e si ritorna l'elemento in posizione *cur.* Da notare l'introduzione di una variabile di istanza *boolean rimuovibile:* questa difatti, messa a *true*, indicherà l'avvenuta corretta di una *next* e permetterà di eseguire l'operazione di *remove* senza generare eccezioni (*IllegalStateException*).

```
@Override
public void remove(){
        if ( rimuovibile==false ) throw new IllegalStateException();
        rimuovibile = false;
        for ( int i = cur; i <= size-2; i++ ){
                contenuto[i] = contenuto[i+1];
        }
        contenuto[size-1] = null;
        cur--;
        size--; resize();
        concurrentALCounter++;
        concurrentCounter++;
}//remove
```

L'operazione di *remove* in sé si basa sull'andare a traslare di un posto a sinistra tutti gli elementi della lista, a partire dall'elemento da eliminare. In questo modo l'elemento da cancellare verrà immediatamente settato col suo successivo mentre il contenuto della ultima posizione verrà posto a *null.*
In realtà la funzionalità del metodo sarebbe stata analoga anche decrementando direttamente la *size* e saltando questa ultima operazione: infatti gestendo correttamente gli incrementi e decrementi sulla dimensione dell'array, non si potrebbe più avere realmente accesso a quel determinato elemento che difatti risulterebbe eliminato. Si veda quindi il tutto come forma d'aiuto concorrenziale al *Garbage Collector* per aumentarne efficienza e rapidità.

```
private class myListIterator implements ListIterator<T>{

        private int concurrentCounter = concurrentALCounter;

        private int cur = -1;
        private boolean editabile = false;
        private Spostamento direzione;

        …

}//myListIterator
```

Per quanto riguarda invece *myListIterator* essa si predispone ancora come una classe privata ma indetta stavolta a implementare tutte le operazioni di un *listIterator.* Ciò che ci si presenta di nuovo rispetto a un normale iterator, è la presenza di metodi fondamentali tali *add*, *set*, *previous* e di altri minori ruotanti intorno a questa nuova gestione dell'iterazione. Se i metodi quali hasNext e next risultano essere prettamente analoghi a quelli di un iterator, eccezione va fatta invece per l'operazione di *remove*, che qui è messa a gestire non solo la rimozione derivante da una next, ma anche quella collegata a una operazione di *previous.*

```
@Override
public void remove() {
        if ( editabile==false ) throw new IllegalStateException();
        editabile = false;
        int toDelete = cur;
        if ( direzione==Spostamento.PREVIOUS ) toDelete = cur+1;
        for ( int i = toDelete; i <= size-2; i++ ){
                contenuto[i] = contenuto[i+1];
        }
        contenuto[size-1] = null;
        if ( direzione==Spostamento.NEXT ) cur--;
        size--; resize();
        concurrentALCounter++;
        concurrentCounter++;
}//remove
```

Per ovviare a ciò è stata istanziata precedentemente un variabile locale di tipo enumerativo, *direzione,* la quale tiene sotto controllo il tipo di operazione effettuata. Essa è proprio alla base logica della implementazione di questo metodo: se equivale a *PREVIOUS* difatti, sappiamo che il cursore va già a puntare una posizione diversa da quella dell'elemento da eliminare, che dovrebbe essere quello ritornato dall'operazione di *previous.* Vado perciò a istanziare un cursore temporaneo definito *toDelete* che in caso di NEXT resterà uguale al corrente effettivo dell'iteratore, mentre in caso di PREVIOUS sarà incrementato di uno per far si che punti all'effettivo elemento eliminare. Stesso controllo verrà poi effettuato per il decremento del cursore, il quale avverrà solo in caso di *next* (in caso di *previous* esso è già stato decrementato).

Da sottolineare infine, che stesse analoghe considerazioni verranno prese nel caso dell'operazione di *set,* dove verrà ancora sfruttata la variabile enumerativa per determinare l'elemento di quale posizione modificare.

Seguirà anche qui l'incremento dei contatori per la *ConcurrentModificationException;*

Una delle nuove operazioni introdotte dal *listIterator* si identifica invece con *add.*

```
@Override
public void add(T e) {
	editabile = false;
	if ( size==0 ){
		contenuto[0] = e;
		size++; resize();
	}
	else{
		size++; resize();
		for ( int i = size-1; i>=cur+2; i-- ){
			contenuto[i] = contenuto[i-1];
		}
		contenuto[cur+1] = e;
	}
	cur++;
	concurrentALCounter++;
	concurrentCounter++;
}//add
```

Qui nell'array-list, *l'add* si articola sostanzialmente in due casi: size= 0 e size≠0. La prima situazione implica che la list in questione deve essere è ancora vuota, e quindi viene posto l'elemento dato dal parametro formale nella posizione 0 e si procede con l'incremento della *size* e controllo per *resize.* L'altro caso invece predispone prima che venga creato spazio per il nuovo elemento *(size++ e resize*) per poi procedere con un ciclo *for* da *size-1* a *cur+2*, con lo scopo di traslare a destra ogni elemento della lista a partire da *cur+1*: a questo punto quindi l'elemento subito dopo il corrente viene settato a quello da aggiungere. Come ultima operazione, seguito dall'incremento dei *counter*, vi è ovviamente l'incremento del cursore il cui *previous*, sulle direttive di Java, deve ritornare l'elemento appena aggiunto.

Per concludere, la classe prevede anche l'implementazione di un *listIterator* che possa essere inizializzato da una posizione diversa da quella di default: dopo aver controllato la validità di questa, ci si appella a una ulteriore classe privata *myListIteratorPOS* alla quale è delegato il compito di gestire il resto.
Essendo che questo iteratore alla fine va a prevedere metodi del tutto analoghi a un *listIterator,* ho deciso di rendere l'oggetto erede di *listIterator* implementando così un costruttore al quale passare la posizione da cui essere inizializzato. Basandosi su un array, è stato relativamente intuitivo andare a predisporre il *cur* del costruttore super a *pos*(izione)*-1.*

```
@Override
public ListIterator<T> listIterator(int pos) {
	if ( pos<0 || pos>size ) throw new IndexOutOfBoundsException();
	return new myListIteratorPOS(pos);
}//listIterator (da posizione)
```

```
private class myListIteratorPOS extends myListIterator{

        private myListIteratorPOS(int pos){
                super();
                super.cur = pos-1;
        }//costruttore da posizione

}//myListIteratorPOS
```

- ## LinkedList.java

Con la *LinkedList* ci si accosta a una struttura dati molto più articolata. Essa rientra nel genere di lista concatenata (o linkata) doppia, cioè costruita attraverso oggetti *nodi* che portano con sé il riferimento al nodo successivo e a quello precedente (*prior e next*), oltre ovviamente all'informazione stessa che il nodo deve rappresentare (*info*). Ed è per questo motivo che come prima cosa ci si prospetta a istanziare una classe (*static* per sottolineare il totale "distaccamento" dall'oggetto *outer)* attraverso la quale verranno creati tutti i nuovi nodi da aggiungere alla lista linkata.
La logica di implementazione di questa classe sfrutta la dichiarazione di due variabili di istanza che andranno a puntare direttamente ai due nodi di *inizio* e *fine*: questi risulteranno infatti di cruciale importanza di fronte a metodi che devono agire direttamente sugli estremi della *linkedList.* Difatti la linked-list, per la sua particolare strutturizzazione, presenta una eccezionale complessità in termini di rimozione e aggiunta di elementi, a differenza dell'array-list in cui bisogna avviare cicli di spostamento e ridimensionamento dell'array, come abbiamo visto. Per esempio, la linked-list si presta egregiamente sotto le vesti di *coda,* tanto che nella istanziazione ufficiale di Java essa va a implementare l'interfaccia *Queue*.
La linked-list inoltre si differenzia dall'array-list anche per l'introduzione, così come previsto pure dalla traccia, di quattro nuovi metodi: *removeFirst*. *removeLast*, *getFirst* e *getLast*. Tutti e quattro vanno ad interagire direttamente col primo e ultimo elemento della lista e da qui resa evidente l'intuitività dell'utilizzo dei nodi *inizio* e *fine* per agire con e sulla lista linkata.

```
public T removeLast(){
        if ( size==0 ) throw new NoSuchElementException();
        T ris = fine.info;
        fine = fine.prior;
        if ( size==1 ) inizio = null;
        else fine.next = null;
        size--;
        concurrentALCounter++;
        return ris;
}//removeLast
```

Ad esempio, il metodo *removeLast* si predispone, come è facilmente intuibile, all'eliminazione di dell'ultimo nodo della linked-list. Esso diviene perciò uno dei metodi base, ad esempio, quando si stia implementando la lista come *queue.* Dopo aver verificato che vi siano elementi da eliminare, il metodo salva l'elemento da ritornare in una variabile locale (corrispondente all'elemento da eliminare) e pone il nodo *fine* uguale al suo precedente e il suo successivo a *null*: in questo modo cade completamente ogni possibile riferimento a quel nodo che presto verrà raccolto dal *Garbage Collector* e disassegnato dalla memoria.

Inoltre, vado a gestire anche il caso in cui la lista conta un solo elemento. Difatti, in questa specifica situazione, *inizio* e *fine* punteranno allo stesso elemento, e quindi risulterà necessario andare a staccare il riferimento anche dal nodo *inizio.* Seguiranno poi ovviamente sia il decremento della dimensione della linked-list, sia l'incremento del *concurrentALCounter* già analizzato precedentemente.

```
public void addFirst( T e ){
        Nodo<T> nuovo = new Nodo<T>();
        nuovo.info = e;
        if ( size==0 ){ inizio = nuovo; fine = nuovo; }
        else{
                inizio.prior = nuovo;
                nuovo.next = inizio;
                inizio = nuovo;
        }
        size++;
        concurrentALCounter++;
}//addFirst
```

Anche addFirst si predispone come un altro metodo caratteristico della linked-list. Innanzitutto viene istanziato un nuovo nodo il quale predisporrà come *info*, l'elemento generico che viene passato dal parametro formale. A questo punto il metodo interpone due casi: size=0 e size≠0. Nella prima situazione, la linked list risulta ancora vuota ed è per questo motivo che sia *inizio* che *fine* vengono fatti puntare al nodo appena creato. Nell'altro caso invece, la logica di implementazione che ho adottato presuppone di indirizzare il *prior* dell'inizio corrente al nodo da aggiungere, mentre il nodo *next* di quest'ultimo sarà linkato all'*inizio* corrente. Solo alla fine *inizio* sarà indirizzato al nuovo nodo che, aggiunto in prima posizione, rappresenterà il nuovo inizio della lista.
Analogamente avviene l'implementazione *dell'addLast*, che utilizza *fine* al posto e di *inizio*.
Per quanto riguarda invece i nodi previsti obbligatoriamente dall'interfaccia, spicca l'*add* che anche qui, così come nella implementazione ufficiale di Java, ho ricondotto all'*addLast* dato che esplicitano la medesima richiesta.

Come nell'array-list, anche in questa classe mi son predisposto alla creazione di due classi *inner* private che andassero ad esplicitare le funzioni di *iterator e listIterator*. Entrambe presentano una visione nella loro implementazione sempre associabile alla stessa vista per l'altra *list* analizzata.
Qui in particolare però, ci si prospetta verso una diversa istanziazione dei vari metodi, i quali fanno riferimento a un *cur* (elemento corrente o cursore) che non rappresenta più direttamente l'indice stesso di posizione dell'elemento, ma si manifesta necessariamente come un nodo.

```
 @Override
public T next() {
        if ( !hasNext() ) throw new NoSuchElementException();
        if ( cur==null ) cur = inizio;
        else cur = cur.next;
        rimuovibile = true;
        return cur.info;
```

```
}//next
```

Ad esempio, il metodo *next* si predispone innanzitutto ad andare a verificare se il nodo *cur* sia diverso da *null.* Difatti, gestendo correttamente l'*hasNext*, non rimane che il caso in cui l'iteratore sia stato appena inizializzato: ciò implica che *cur* debba essere ricondotto all'*inizio* della lista.
Se l'*if* considerato dà come esito *false,* non resta altro che porre il cursore uguale al suo successore, cioè far sì che *cur* abbia il riferimento al nodo *next*, e così di seguito.

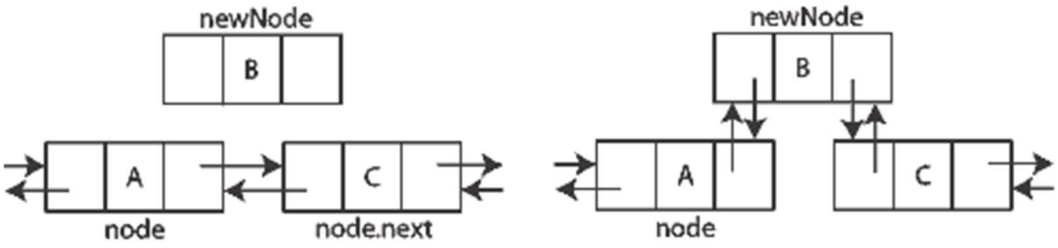
Per quanto riguarda il *listIterator,* e quindi i metodi della classe privata *myListIterator,* passo ora ad analizzarne alcuni tra i più articolati. In generale però, oltre alle ormai consolidate variabili di istanza già viste in array-list (*enum direzione, editabile,…),* ho reso necessaria l'introduzione di una nuova variabile di tipo *int, indice,* la quale è adibita a mantenere aggiornato il numero della posizione espressa dal *cur* (la sua implementazione non si era resa necessaria nel caso dell'array-list perché il *cur* è esso stesso un indicatore numerico di posizione). Essa sarà poi particolarmente utile per la gestione dei metodi di *nextIndex* o *previousIndex* e soprattutto per l'implementazione del *listIteratorPOS*.

```
@Override
public void add(T e) {
        editabile = false;
        Nodo<T> nuovo = new Nodo<>();
        nuovo.info = e;
        if ( cur==null ){
                if ( size==0 ){
                        fine = nuovo;
                        inizio = nuovo;
                }
                else {
                        inizio.prior = nuovo;
                        nuovo.next = inizio;
                        inizio = nuovo;
                }
        }
        else if ( cur==fine ){
                fine.next = nuovo;
                nuovo.prior = fine;
                fine = nuovo;
        }
        else {
                cur.next.prior = nuovo;
                nuovo.next = cur.next;
                nuovo.prior = cur;
                cur.next = nuovo;
        }
        cur = nuovo;
        indice++;
        size++;
```

```
        concurrentALCounter++;
        concurrentCounter++;
}//add
```

Il metodo dell'*add* si predispone suddiviso in una serie di casi principali, così come poi vedremo per lo stesso metodo del *remove.* Innanzitutto vado a verificare che *cur* possa puntare a *null:* ciò implica che la posizione in cui vorremmo aggiungere l'elemento sia quella iniziale. A questo punto perciò, se la linked-list risulta vuota, si passa ad inizializzarla collegando sia *inizio* che *fine* al nuovo elemento, mentre se questa presenta già almeno un elemento si passa a sistemare i riferimenti affinché il nuovo elemento risulti in testa.
Se invece la verifica della prima condizione si pone a *false,* il compilatore passa a convalidare o meno un *else if* che gestisce il caso in cui il cursore punti alla *fine,* e quindi si trovi dopo l'ultimo elemento.
Infine ho predisposto la gestione del generico caso in cui *cur* sia strettamente incluso all'interno della lista, predisponendo, in ordine, che:

1. Il precedente del successore di *cur* sia indirizzato al nuovo elemento;
2. Il successore del nuovo elemento venga ricondotto al successore del corrente;
3. Il *prior* del nuovo elemento abbia come riferimento *cur;*
4. Il successore del corrente sia indirizzato al nuovo elemento.



Segue poi l'operazione comune a tutti i casi, di spostare ovviamente il riferimento del cursore al nuovo nodo. Infine il metodo si conclude con l'incremento dei contatori *size e indice* e dei *counter* adibiti al sollevamento della *ConcurrentModificationException.*

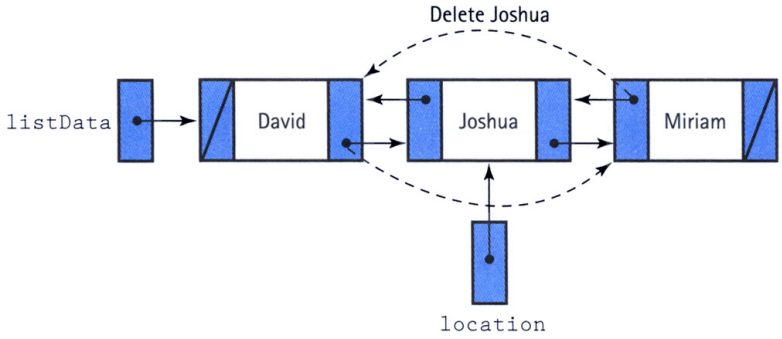
L'altro metodo che vado ad analizzare è il *remove,* un altro la cui implementazione è stata stravolta passando dalla vecchia struttura dati a questa. Come già preannunciato, anche qui si predispone l'utilizzo del tipo enumerato *direzione* per gestire l'elemento da eliminare in merito al tipo di spostamento effettuato. Inoltre, così come avveniva per l'array-list, ho ideato di introdurre una nuova variabile locale rappresentante l'effettivo elemento da eliminare, definita *curToDelete:* questa, inizialmente istanziata uguale a *cur,* verrà tramutata a *cur.next* se si è iterato sulla lista attraverso una operazione di *previous.* Questo perché sappiamo, come già detto, che l'elemento di ritorno di questo metodo non corrisponde al nodo al quale punta effettivamente *cur.* Difatti, a operazione terminata, il cursore si trova tra *cur* e il suo successore e ovviamente questo si riferisce a quel nodo che ancora deve essere "attraversato".

Una vota definito l'elemento da eliminare, viene interrogato il *curToDelete* con lo scopo di verificare se questo è linkato a uno degli estremi della lista (*inizio* o *fine)* o si trovi strettamente incluso all'interno di questa. Nella prima situazione, ho implementato una gestione abbastanza simile in cui indirizzo i nodi di riferimento degli estremi rispettivamente al successivo, nel caso di *inizio*, e al precedente nel caso di *fine.*

Seguono poi due accezioni particolari che distanziano i due casi. Verificatosi *curToDelete==inizio* difatti, occorre evidenziare che se la linked-list presentava un solo elemento, è ora necessario predisporre che anche la *fine* punti a *null*. La seconda situazione riguarda il controllo basato sul tipo di operazione effettuata, che se equivale a *NEXT* pone *cur* = *null* nel primo *if, cur* = *fine* nell'*else if*. Questo perché una volta effettuata una *remove*, è necessario che *cur* punti all'elemento immediatamente precedente per rendere "legale" tutta l'iterazione.

Se saltano entrambe le condizioni sin ora presentate, ho predisposto l'ultimo caso alla base dell'effettivo "bypass" senza particolari situazioni da gestire:

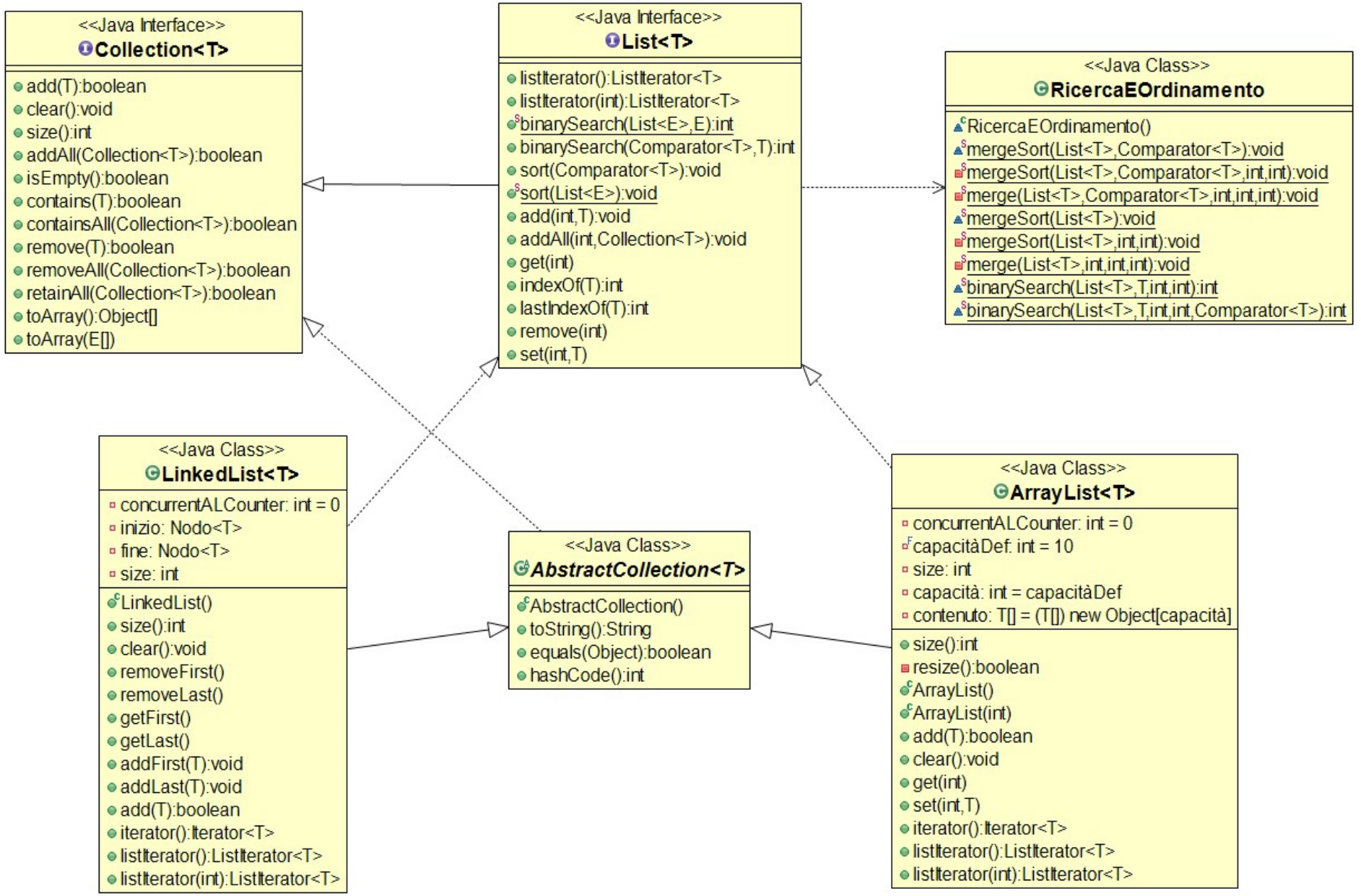
1. il riferimento del *next* del nodo precedente a *curToDelete,* viene fatto puntare al successore di *curToDelete* stesso;
2. Il *prior* del successore di *curToDelete* viene linkato al precedente di *curToDelete;*



Come ultima operazione poi, come è stato presentato negli altri casi, vado a determinare ciò che andrà ora a puntare il cursore*:* difatti, sempre in caso di *NEXT,* predispongo che *cur* indirizzi al suo precedente per gli stessi motivi che si possono evincere da quanto definito nell'introduzione del metodo e nei primi due casi.

Per concludere, seguono poi il decremento della *size* e l'aggiornamento dei *counter,* che ovviamente, fuori da qualsiasi *if,* vengono effettuati a prescindere dai casi.

Vedremo ora proposta una esemplificazione rapida di questa prima parte di progetto attraverso diagramma UML.



Longo Giuseppe
175983

Longo Giuseppe
175983

La seconda parte del progetto va a richiedere una GUI su una possibile implementazione di metodi e classi fin ora descritti. Ovviamente questa fa riferimento all'utilizzo del framework proprio di Java, *Swing*, con la necessaria implementazione di qualche componente risalente a *Awt*. Difatti, il framework *Swing* nacque come un aggiornamento o sostituto al vecchio *Awt* con lo scopo di eliminarne tanti problemi di cui quest'ultimo si faceva responsabile. Innanzitutto, *Swing* si distacca quasi completamente dal sistema in cui viene eseguito, con componenti interamente sviluppati in Java che addirittura proprio per questo motivo, implementano un *LookAndFeel* proprio di Java. Questo implica che tutto il framework gira allo stesso modo su ogni macchina virtuale, permettendo così una più facile gestione di bug ed errori.

Per la realizzazione della interfaccia grafica, ho predisposto innanzitutto la creazione di un nuovo package *CollectionTestGUI* nel quale inserire tutte le classi che concorrono alla creazione della GUI, appunto, e che importano le strutture dati analizzate sin ora.
La classe CollectionsTestGUI difatti, non rappresenta altro che l'oggetto che una volta inizializzato nel *main* (presento dentro la stessa classe), non andrà a far altro che creare un evento che si metterà in coda in attesa che il *thread EDT* (controllore degli eventi) lo vada a processare.
Perciò il costruttore di *CollectionsTestGUI* conterrà al suo interno l'implementazione di un metodo *Runnable* che andrà a sua volta a creare ed avviare il primo oggetto *frame* base nella logica di questa implementazione, *FrameToChoose.* In esso, l'utente sarà invitato alla scelta sia del tipo di *list* su cui lavorare (array-list o linked-list) sia sul tipo da implementare (String o int).
A questo punto, verrà avviato il *frame* principale di questa interfaccia, *FrameOfMethods,* nel quale l'utente potrà dilettarsi nella scelta dei metodi da invocare, osservando i risultati e gestendone di nuovi. Difatti, a ciascun bottone premuto spesso verrà corrisposta l'invocazione e la nascita di un altro frame per, ad esempio, la gestione dell'input da tastiera quando richiesto: ecco quindi giustificata l'implementazione di classi quali *FrameForCollectionsButtons, FrameForLinkedLiatButtons, FrameForListButtons, FrameForListIterator.*

Fondamentale è stata poi l'istanziazione della classe *CollectionToGrafica:* essa rappresenta l'oggetto "collante" tra i metodi di *Swing* e quelli provenienti dalle strutture dati su cui lavorare. Si parla di oggetto poiché difatti essa risulta generica in T e sarà quindi ogni volta il *listener* istanziato nel *frame* iniziale a chiarirne il tipo. Una volta creato, il *CollectionToGrafica* (*ctg*) verrà passato come parametro formale al costruttore del *FrameOfMethods.* A tutte le altri classi contenenti l'implementazione degli altri *frame* invece, verrà passato sia il *FrameOfMethods (fom)* che lo ha invocato, così da aver pieno accesso a componenti utili per una più semplice gestione, sia un tipo enumerativo indicante il metodo da dover implementare\eseguire.
Particolare risulta poi la presenza di una classe di utilità *EsempiComparator*, predisposta per un programmatore che voglia dilettarsi nella introduzione di altri *comparator* oltre a quelli già proposti come esempio (ciò comunque richiederà l'implementazione di una serie di modifiche nella parte predisposta alla gestione grafica a discrezione sempre del programmatore).

Concludo presentando un esaustivo grafico UML di quanto implementato in questa seconda parte.

Longo Giuseppe
175983