# Corso di Programmazione Orientata agli Oggetti – Relazione Progetto LinkedList<T>

**Studente: De Marco Alessandro - N.M. 190020**

**Progetto assegnato**

*Prima parte*

*Si tratta di completare l'implementazione di LinkedList<T> pre-implementando nell'interfaccia List<T> quanti più metodi è possibile, prestando attenzione alle eccezioni coinvolte, e concretizzando in una classe astratta AbstractList<T> che implementa List<T>, i metodi canonici toString(), equals() e hashCode().*
*Occorre testare accuratamente il progetto e assicurarsi che la classe LinkedList<T> si comporti esattamente come la classe java.util.LinkedList<T> della libreria di Java, relativamente ai metodi previsti in List<T>.*
*Successivamente si deve modificare l'interfaccia List<T> in cui il metodo sort è ora un metodo static e generico che riceve la lista da ordinare ed un comparatore e provvede ad ordinare la lista utilizzando esclusivamente i servizi di ListIterator<T> disponibile nell'interfaccia.*

*Seconda parte*

*Realizzare un front-end grafico (GUI) che consenta di richiedere una qualsiasi operazione su una linked list di interi, evocandola attraverso la scelta di un elemento di menu. In una text area si deve poter seguire in modo chiaro l'effetto dell'operazione effettuata, ad es. lo spostamento del caret ^ a seguito del movimento di un list iterator etc. Il contenuto di una lista deve poter essere salvato su file o caricato da file, in formato serializzato.*

- **Interfaccia List<T>**

Di seguito i metodi `default` pre-implementati nell'interfaccia List<T>:

`int size()` ad ogni next() dell'iteratore (finché questo hasNext()) incrementa di 1 il contatore *c* e ne ritorna il risultato;

`boolean contains(T x)` effettua una ricerca lineare dell'elemento *x* nella lista tramite l'iteratore. Ritorna true se incontra un elemento che è equals a *x*, false altrimenti;

`void clear()` dentro un ciclo l'iteratore effettua una next() seguita da una remove() finché questo hasNext();

`void add(T x)` e `void addLast(T x)` aggiungono l'elemento *x* con una add() dell'iteratore portato in ultima posizione;

`void addFirst(T x)` effettua una semplice aggiunta dell'elemento *x* in prima posizione tramite l'iteratore;

`T getFirst()` verifica preliminarmente con hasNext() che la lista sia non vuota, nel caso in cui sia vuota lancia una NoSuchElementException. Successivamente ritorna l'elemento con una next() dell'iteratore;

`T getLast()` verifica preliminarmente con hasNext() che la lista sia non vuota, nel caso in cui sia vuota lancia una NoSuchElementException. Successivamente l'iteratore viene portato in ultima posizione e ritorna l'ultimo elemento della lista con una previous();

`T removeFirst()` verifica preliminarmente con hasNext() che la lista sia non vuota, nel caso in cui sia vuota lancia una NoSuchElementException. Salva il primo elemento della lista in una variabile, lo rimuove e ritorna il valore della variabile;

`T removeLast()` verifica preliminarmente con hasNext() che la lista sia non vuota, nel caso in cui sia vuota lancia una NoSuchElementException. L'iteratore viene portato in ultima posizione, salva l'ultimo elemento della lista in una variabile, lo rimuove e ritorna il valore della variabile;

`void remove(T x)` effettua una ricerca lineare dell'elemento *x* nella lista tramite l'iteratore. Se incontra un elemento che è equals ad *x* effettua una remove() ed interrompe il ciclo;

`boolean isEmpty()` ritorna il valore invertito di hasNext() dell'iteratore;

`boolean isFull()` ritorna false in quanto non si verifica mai che la lista sia piena.

Il metodo generico statico `static <T> void sort` che riceve come parametri una List<T> ed un Comparator<T> mira ad ordinare la lista secondo le condizioni di ordinamento del Comparator sfruttando esclusivamente i metodi forniti dal ListIterator (con l'algoritmo bubble sort). Il metodo controlla preliminarmente se la lista contiene almeno due elementi, termina nel caso in cui ne contenesse meno:

```
if( !lit.hasNext() ) return;
lit.next();
if( !lit.hasNext() ) return;
```

A questo punto vengono inizializzate alcune variabili: boolean *switches*, inizialmente impostata a true, indica se sono necessari ancora scambi da fare; int *pos*, che indica la posizione corrente dell'iteratore, viene inizialmente impostata ad 1 in quanto durante il controllo del numero minimo di elementi necessari nella lista che era stato svolto in precedenza, l'iteratore aveva già effettuato una next() e pertanto era avanzato di una posizione; int *limit*, inizialmente impostata alla dimensione della lista, rappresenta l'indice al di sopra del quale la lista risulta ordinata; int *lsp*, inizialmente impostata a 0, indica la posizione relativa all'ultimo scambio effettuato. Viene introdotto il ciclo while principale, con condizione che *switches* sia uguale a true, all'interno del quale l'algoritmo opera come segue: tramite l'iteratore preleva l'elemento corrente con next() nella variabile *cur*, torna indietro con previous(), imposta *switches* a false e innesta un ciclo, il quale gira finché *pos* sia minore di *limit*, dentro il quale dichiara una variabile *pre*, confronta l'elemento precedente di cur a *cur* e nel contempo assegna il suo valore a *pre*. Se *cur* risulta minore di *pre* procede con lo scambio impostando il valore di *cur* nel suo precedente e il valore di *pre* nel corrente, aggiorna poi la posizione dell'ultimo scambio (ossia *pos*) ed imposta *switches* a true:

```
T pre=null
if( c.compare(cur, pre=lit.previous())<0 ) {
    lit.set(cur);
    lit.next();
    lit.next();
    lit.set(pre);
    lsp=pos;
    switches=true;
}
```

Se le condizioni della precedente disuguaglianza non si fossero verificate, l'iteratore avanza di due posizioni. Aggiorna *cur* con l'elemento successivo della lista (null se in ultima posizione) ed incrementa *pos*. Fuori dal ciclo innestato aggiorna *limit* con la posizione dell'ultimo scambio e riporta l'iteratore in prima posizione ripristinando il valore di *pos:*

```
limit=lsp;
while( lit.hasPrevious() ) lit.previous();
lit.next(); pos=1;
```

I metodi `ListIterator<T> listIterator()` e `ListIterator<T> listIterator(int from)` rimangono astratti dal momento che non è possibile definirli nell'interfaccia.

- **Classe astratta AbstractList<T>**

Definisce i metodi canonici `toString()`, `equals(Object o)` e `hashCode()`.

`public String toString()` sfrutta uno StringBuilder e l'iteratore. Allo StringBuilder viene inizialmente aggiunta "[", in un ciclo l'iteratore scansiona la lista e ogni elemento viene aggiunto allo StringBuilder (insieme a ", " se l'elemento non è l'ultimo della lista). Per chiudere allo StringBuilder viene aggiunta "]". Viene ritornato lo StringBuilder in formato stringa.

`public boolean equals(Object o)` dopo le verifiche canoniche effettua il cast di *o* a List<T> per controllarne l'uguaglianza effettiva con this: prima controlla la size delle due liste e successivamente ogni elemento che le compone tramite l'utilizzo di due iteratori (uno per lista).

`public int hashCode()` - implementazione tratta dal libro "Effective Java" di Joshua Bloch –
Assegnato un valore non zero alla variabile di ritorno. Tramite l'iteratore, per ogni elemento *f* presente nella lista, viene inizializzata una variabile *c* e si controlla il tipo di *f*:
se *f* è istanza di Boolean a *c* viene assegnato 0 se *f* è true, 1 altrimenti;
se *f* è istanza di Byte a c viene assegnato il valore di *f*;
se *f* è istanza di Character a *c* viene assegnato il corrispondente valore in intero di *f*;
se *f* è istanza di Short oppure Integer a *c* viene assegnato il valore di *f*;
se *f* è istanza di Long viene effettuato il cast di *f* a long e a *c* viene assegnato il risultato dopo un cast ad intero dell'operazione XOR fra *f* ed *f* shiftato a destra di 32 bit:

```
else if( f instanceof Long ) {
    long l=(long)f;
    c=(int)(l^(l >>> 32));
}
```

se *f* è istanza di Float a *c* viene assegnato il valore di *f* convertito ad intero;
se *f* è istanza di Double si converte *f* a long e si procede come nel caso di *f* istanza di Long;
se *f* è istanza di un qualsiasi altro oggetto, ed *f* non è null, a *c* viene assegnato il valore ottenuto chiamando hashCode() di *f.*
Alla variabile di ritorno viene assegnato il risultato ottenuto tramite la moltiplicazione fra il suo valore corrente ed un numero primo (in questo caso 37) e sommando *c*. Al termine del ciclo viene ritornato il valore hash della lista.

- **Classe concreta LinkedList<T>**

Variabili di istanza: `private int count=0;` (contatore modifiche)
`private Nodo<T> first=null, last=null;`
`private int size=0;`

Metodi:

`public int size()` ritorna il valore di `size`.

`public boolean contains(T x)` effettua una ricerca lineare di x nella lista.

`public void clear()` annulla `first` e `last` e azzera `size` rendendo la lista garbage.

`public void add(T x)` chiama addLast(x).

`public void addFirst(T x)` crea un nuovo Nodo *n* assegnandogli x ad info, `first` a next e `null` a prior. Si possono incontrare 2 casi distinti: nel caso in cui la lista non fosse vuota, assegna *n* come elemento precedente del capolista e ridefinisce n come nuovo `first`; nel caso in cui la lista fosse vuota, ridefinisce *n* anche come nuovo `last`.

`public void addLast(T x)` crea un nuovo Nodo *n* assegnandogli x ad info, `null` a next e `last` a prior. Similmente ad addFirst(), nel caso in cui la lista non fosse vuota, assegna *n* come elemento successivo dell'ultimo elemento della lista e ridefinisce n come nuovo `last`; nel caso in cui la lista fosse vuota, ridefinisce *n* anche come nuovo `first`.

`public T getFirst()` se la lista non è vuota, ritorna `first`.

`public T getLast()` se la lista non è vuota, ritorna `last`.

`public T removeFirst()` se la lista è vuota lancia una NoSuchElementException, altrimenti memorizza il valore di `first` in una variabile, `first` assegna il suo next a se stesso; se il nuovo `first` è null, ovvero la lista conteneva un solo elemento, allora assegna null anche a `last`, altrimenti annulla l'elemento precedente a `first` assegnando null al prior di quest'ultimo. Ritorna infine il valore dell'elemento appena rimosso.

`public T removeLast()` se la lista è vuota lancia una NoSuchElementException, altrimenti memorizza il valore di `last` in una variabile, `last` assegna il suo prior a se stesso; se il nuovo `last` è null, ovvero la lista conteneva un solo elemento, allora assegna null anche a `first`, altrimenti annulla l'elemento successivo a `last` assegnando null al next di quest'ultimo. Ritorna infine il valore dell'elemento appena rimosso.

`public void remove(T x)` mediante una variabile *cor* attraversa la lista. Se esiste un elemento che verifica equals con x si possono distinguere 3 casi: x è uguale al primo elemento della lista `first`, allora procede analogamente a removeFirst(); x è uguale all'ultimo elemento della lista `last`, allora procede analogamente a removeLast(), con l'unica differenza di non verificare se `last` è nullo, in quanto se così fosse stato, x sarebbe stato anche uguale a `first` e si sarebbe verificato il primo caso; x è uguale ad un generico elemento della lista (né `first` né `last`), allora "sgancia" l'elemento dalla lista assegnando l'elemento successivo al next dell'elemento precedente e l'elemento precedente al prior dell'elemento successivo:

```
cor.prior.next=cor.next;
cor.next.prior=cor.prior;
```

`public boolean isEmpty()` verifica che esisti `first`.

Classe inner `ListIteratorImpl`:

Variabili di istanza:
```
private Nodo<T> previous, next;
private Move lastMove=Move.UNKNOWN;  (dall'enum Move nella classe outer)
private int itCount=count;  (contatore modifiche)
```

Costruttori:

```
public ListIteratorImpl() {
            previous=null; next=first;
        }
```

`public ListIteratorImpl(int from)` lancia una IllegalArgumentException se l'argomento non è compreso fra 0 e la dimensione della lista. Se l'argomento è uguale alla dimensione della lista assegna `last` a `previous` e null a `next`, posizionando l'iteratore dopo l'ultimo elemento della lista, altrimenti in un ciclo avanza la posizione dell'iteratore di `from` posizioni.

Metodi:

`public boolean hasNext()` verifica che `next` non sia nullo;

`public T next()` controlla che non siano state apportate modifiche alla lista tramite i metodi esterni all'iteratore dopo l'apertura di questo, o che l'iteratore abbia un nodo successivo, altrimenti lancia rispettivamente una ConcurrentModificationException o una NoSuchElementException. A questo punto imposta il verso di spostamento assegnando a `lastMove` la costante di enumerazione ***FORWARD***, il puntatore immaginario dell'iteratore si sposta in avanti di una posizione e viene ritornato l'elemento appena superato.

`public boolean hasPrevious()` verifica che `previous` non sia nullo.

`public T previous()` controlla che non siano state apportate modifiche alla lista tramite i metodi esterni all'iteratore dopo l'apertura di questo, o che l'iteratore abbia un nodo successivo, altrimenti lancia rispettivamente una ConcurrentModificationException o una NoSuchElementException. A questo punto imposta il verso di spostamento assegnando a `lastMove` la costante di enumerazione ***BACKWARDS***, il puntatore immaginario dell'iteratore si sposta indietro di una posizione e viene ritornato l'elemento appena superato.

`public void remove()` controlla che non siano state apportate modifiche alla lista tramite i metodi esterni all'iteratore dopo l'apertura di questo, o che ci sia stato un recente spostamento del puntatore dell'iteratore, altrimenti lancia rispettivamente una ConcurrentModificationException o una IllegalStateException. Assegna ad una variabile Nodo *r* il nodo da rimuovere in base al verso di spostamento: il nodo viene "scollegato" come visto in precedenza nel metodo omonimo della classe outer. Infine imposta ***UNKNOWN*** come verso di spostamento.

`public void add(T elem)` controlla che non siano state apportate modifiche alla lista tramite i metodi esterni all'iteratore dopo l'apertura di questo, altrimenti lancia una ConcurrentModificationException. Crea un nuovo Nodo assegnandogli `elem` e gli attuali `next` e `previous`, se l'iteratore si trova in prima posizione assegna il nuovo nodo a `first`, altrimenti al successivo di `previous`; se l'iteratore si trova in ultima posizione lo assegna a `last`, altrimenti al precedente di `next`:

```
if( previous==null )
    first=a;
else
    previous.next=a;
if( next==null )
    last=a;
else
    next.prior=a;
```

in ogni caso `previous` diventa il nodo appena aggiunto e imposta ***UNKNOWN*** come verso di spostamento.

`public void set(T elem)` controlla che ci sia stato un recente spostamento del puntatore dell'iteratore, o che non siano state apportate modifiche alla lista tramite i metodi esterni all'iteratore dopo l'apertura di questo, altrimenti lancia rispettivamente una IllegalStateException o una ConcurrentModificationException. Imposta la info del nodo `next` o `previous` in base al verso di spostamento.

`public int nextIndex()` e `public int previousIndex()` in quanto non supportati da questa versione del ListIterator lanciano una UnsupportedOperationException.

- **Classe interfaccia grafica LinkedListGUI**

Fornisce all'utente un'interfaccia front-end tramite la quale può comodamente gestire una o più LinkedList di Integer. Gran parte della classe è costituita dalla classe non pubblica FrontEnd che estende JFrame, dentro la quale sono definiti pannelli, altre classi frame e un ActionListener.

In testa alla classe FrontEnd vengono introdotti come variabili di istanza un File di salvataggio, i menu edit e iteratore, gli elementi dei vari menu, una stringa titolo della finestra, una List<Integer> ed il suo iteratore, un pannello principale ed un pannello iteratore, un'area di testo (dove comparirà la lista), un'etichetta rappresentante la dimensione in tempo reale della lista, una variabile booleana modifiche non salvate (inizialmente impostata a false) ed alcune finestre relative alle operazioni sulla lista.

Nel costruttore si omette l'azione di chiusura di default della finestra, sostituendola con un WindowAdapter che fa comparire un messaggio di conferma all'utente (il quale verrà approfondito più avanti. Viene istanziato l'ActionListener, vengono creati i menu e disabilitate le operazioni non consentite (in particolare viene disabilitato `closeIter` per dichiarare esplicitamente che l'iteratore non è aperto).

Nel momento in cui l'utente apre per la prima volta una lista (sia essa nuova o un file già esistente), alla finestra viene aggiunto il pannello principale `MainPanel` contenente l'area di testo con stampata la lista e la size in alto a sinistra. L'area di testo non è editabile e sfrutta il fine linea per andare a capo.
L'utente è ora abilitato ad utilizzare tutte le operazioni consentite dalla lista, ad eccezione del caso in cui essa sia vuota perché vengono disabilitate le operazioni di get, rimozione, clear, contains, ordinamento e apertura dell'iteratore a partire da una determinata posizione.

Nel menu "Edit", quando l'utente seleziona "Get First" oppure "Get Last", se non già esistente, viene istanziata una nuova `GetFrame`, contenente un campo testo non editabile e una etichetta adattiva gestiti dal metodo `public void set` ricevente un boolean e un intero: in base al tipo di get che si vuole effettuare, la chiamata di `set` stamperà prima nell'area di testo l'elemento restituito dalla get della lista, poi imposterà titolo ed etichetta. Quando l'utente chiude la finestra questa viene resa non visibile.

Quando l'utente seleziona "Add Element", "Add First", "Add Last" oppure "Remove Element", se non già esistente, viene istanziata una nuova `AddRemoveFrame`, contenente un campo testo editabile e una etichetta adattiva. Al fine di ridurre il numero di finestre da istanziare, il metodo `public void setMode` imposta il tipo di operazione da effettuare in base all'intero che riceve come argomento: 1 corrisponde a Add First, 2 a Add Element, 3 a Add Last e 4 a Remove Element. Impostata l'operazione e resa visibile la finestra l'utente può inserire un intero nel campo testo, premendo invio l'ActionListener (implementato come lambda expression) provvederà ad effettuare l'aggiunta / rimozione sulla lista, ad aggiornare i menu con i metodi `menuE()` e `menuD()` (i quali verranno approfonditi più avanti), ad aggiornare la lista nell'area di testo e l'etichetta size. Quando l'utente chiude la finestra questa viene resa non visibile e il campo testo svuotato.

Quando l'utente seleziona "Remove First" oppure "Remove Last" viene effettuata la rimozione in testa / coda e vengono aggiornati menu, area di testo ed etichetta size.

Quando l'utente seleziona "Clear List" viene effettuata la clear() della lista e aggiornati menu, area di testo (stampando per semplicità "[]") ed etichetta.

Quando l'utente seleziona "Contains Element", se non già esistente, viene istanziata una nuova `ContainsFrame`, contenente due etichette e un campo testo editabile posto fra queste ultime. L'utente può inserire un intero nel campo testo e premere invio per effettuare la ricerca. Premendo invio l'ActionListener (implementato come lambda expression) imposta sull'etichetta alla destra del campo testo il risultato della ricerca. Quando l'utente chiude la finestra questa viene resa non visibile, il campo testo svuotato e l'etichetta del risultato reimpostata.

Quando l'utente seleziona "Sort", viene creata e resa visibile una nuova finestra a indicare che l'operazione di ordinamento è in corso. Subito dopo viene chiamato il metodo sort della lista, e al termine di questo la finestra viene resa nascosta, viene aggiornata l'area di testo e la finestra distrutta con il metodo `dispose()`.

Quando l'utente seleziona "Generate HashCode" viene creata e resa visibile una nuova `HashFrame`, contenente un campo testo non editabile dentro il quale viene stampato l'hash code della lista. Quando l'utente chiude la finestra questa viene distrutta.

Nel menu "Iterator", quando l'utente seleziona "Open Iterator" viene istanziato il ListIterator della lista e, se non già esistente, viene creato il pannello `ItPanel` e aggiunto a quello principale. `ItPanel` occupa uno spazio sottostante l'area di testo e pertanto la finestra viene ridimensionata. Il pannello principale viene aggiornato (con `validate()` e `repaint()`) e vengono aggiornati i menu, in particolare, una volta aperto l'iteratore vengono disabilitate le operazioni di aggiunta, rimozione, svuotamento e ordinamento in modo tale da evitare la gestione di eventuali ConcurrentModificationException. Il pannello è composto da un'area di testo a due righe, non editabile e scrollabile orizzontalmente, cinque JButton (Previous, Next, Remove, Add, Set) disposti in fila e due campi testo per le operazioni di add e set posti sotto i rispettivi bottoni. Per l'area di testo è stato impostato un font monospaziato. Sulla prima riga dell'area di testo viene stampata la lista per intero in orizzontale, mentre sulla seconda uno StringBuilder contenente il caret "^" indicante la

posizione corrente. Ad ogni operazione dell'iteratore vengono aggiornate le posizioni del caret dello StringBuilder e del caret dell'area di testo principale (reso visibile e statico):
- spostamento in avanti del caret dello StringBuilder: cancellato il carattere "^", tramite un ciclo che va da 0 alla lunghezza in caratteri dell'elemento successivo della lista più due, appende il carattere spazio allo StringBuilder; terminato il ciclo appende "^", infine aggiorna la posizione del caret non visibile in modo da centrare "^" nell'area di testo dell'iteratore: se lo spazio fra la posizione di "^" e la lunghezza della lista è inferiore a 42 caratteri imposta il caret alla fine della lista, altrimenti 42 caratteri dopo "^":

```
itArea.setCaretPosition(((list.toString().length()-caret.length())<42) ?
list.toString().length()+1 : caret.length()+42);
```

- spostamento all'indietro del caret dello StringBuilder: cancella l'ultima sezione di caratteri dello StringBuilder, ovvero la lunghezza in caratteri dell'elemento precedente della lista più tre caratteri ("^" più due corrispondenti a spazio e virgola di un elemento della lista) e appende "^"; infine aggiorna la posizione del caret non visibile in modo da centrare "^" nell'area di testo dell'iteratore: se lo spazio fra l'inizio della lista e "^" è inferiore a 44 caratteri imposta il caret all'inizio della lista, altrimenti 43 caratteri prima di "^";
- il caret dell'area di testo principale viene posizionato seguendo la lunghezza dello StringBuilder meno un carattere.
Nella classe `ItPanel` è presente una variabile di istanza *dir* integer a indicare il verso di andamento dell'iterazione (avanti se dir=1, indietro se dir=2).

Quando l'utente clicca sul bottone Previous l'iteratore esegue previous(), i caret indietreggiano di una posizione, si aggiorna l'area di testo dell'iteratore, si aggiorna l'abilitazione dei bottoni (se disabilitati, Remove e Set vengono abilitati, se l'iteratore non ha un elemento precedente Previous viene disabilitato) e *dir* viene impostata a 2.

Quando l'utente clicca sul bottone Next l'iteratore esegue next(), i caret avanzano di una posizione, si aggiorna l'area di testo dell'iteratore, si aggiorna l'abilitazione dei bottoni (se disabilitati, Remove e Set vengono abilitati, se l'iteratore non ha un prossimo elemento Next viene disabilitato) e *dir* viene impostata ad 1.

Quando l'utente clicca sul bottone Remove vengono disabilitati i bottoni Remove e Set, se *dir* è uguale ad 1 il caret dello StringBuilder si sposta indietro di una posizione, l'iteratore esegue remove() e vengono aggiornati le aree di testo, l'etichetta size, i bottoni e i caret. Se dopo la rimozione la lista dovesse risultare vuota vengono disabilitate le operazioni get e contains.

Presumendo che l'utente abbia inserito un intero nel campo testo di "Add", quando egli clicca sul bottone vengono disabilitati i bottoni Remove e Set, l'iteratore aggiunge l'elemento presente nel campo testo, i caret si spostano in avanti di una posizione, il campo testo viene svuotato e vengono aggiornati le aree di testo, i menu e i bottoni.

Presumendo che l'utente abbia inserito un intero nel campo testo di "Set", quando egli clicca sul bottone l'elemento del campo testo viene memorizzato in una variabile temporanea, se *dir* è uguale ad 1 il caret dello StringBuilder viene aggiornato, l'iteratore esegue set(), e vengono aggiornate le aree di testo.

Quando l'utente seleziona "Open Iterator From" viene creata una nuova `ItFrame`, contenente un campo testo dentro il quale l'utente inserisce la posizione della lista da cui desidera far partire il list iterator. A questo punto vengono eseguite le stesse operazioni di quando l'utente seleziona Open Iterator.

Ogni volta che l'utente apre l'iteratore, sia selezionando Open Iterator che Open Iterator From, viene eseguito il metodo `public void refresh` della classe `ItPanel`, ricevente un intero relativo alla posizione di partenza dell'iteratore: il metodo si occupa di reimpostare bottoni, caret, ListIterator, posizione, la variabile *dir* e i campi testo.

Quando l'utente seleziona "Close Iterator" il pannello dell'iteratore `ItPanel` viene rimosso dal pannello principale `MainPanel`, il caret dell'area di testo di `MainPanel` viene reso non visibile, `MainPanel` viene

aggiornato, la finestra ridimensionata, la variabile del list iterator impostata a null e vengono riabilitate quelle opzioni dei menu non consentite durante l'uso dell'iteratore.

Nel menu "File", quando l'utente seleziona "New", se è presente una lista le cui modifiche non sono state salvate viene mostrato all'utente un messaggio di conferma e, se l'utente seleziona la risposta affermativa, viene creata una nuova LinkedList vuota. A questo punto vengono aggiornati i menu, in particolare se prima di selezionare New l'utente aveva lasciato aperto l'iteratore della lista precedente, questo e il relativo pannello vengono rimossi mediante le stesse operazioni che svolge l'opzione Close Iterator. Vengono poi aggiornati il titolo della finestra, l'area di testo (inizialmente con la lista vuota), l'etichetta size (inizialmente 0) e il file di salvataggio viene impostato a null.

Quando l'utente seleziona "Open" se è presente una lista le cui modifiche non sono state salvate viene mostrato all'utente un messaggio di conferma e, se l'utente seleziona la risposta affermativa, viene mostrato un JFileChooser (filtrato all'estensione file .list) tramite il quale l'utente può selezionare il file che desidera aprire. Quando l'utente seleziona il file, se questo è valido, viene subito assegnato alla variabile `saveFile` e viene aperto un ObjectInputStream su di esso. L'ObjectInputStream legge l'oggetto lista e i menu, l'area di testo e l'etichetta size vengono aggiornati con le medesime operazioni di quando l'utente seleziona New (con l'unica differenza che non è certo se la lista aperta sia vuota). Nel caso in cui l'ObjectInputStream fallisca nel tentativo di leggere un file viene mostrato all'utente un messaggio di errore.

Quando l'utente seleziona "Save", se le modifiche della lista corrente non sono state salvate e se il file di questa esiste e corrisponde ad un path, viene aperto un ObjectOutputStream sul file, la lista viene sovrascritta e la variabile `unsavedChanges` impostata a false. Se invece il file di salvataggio non esiste, ovvero è stata creata una nuova lista, le operazioni eseguite sono quelle equivalenti all'opzione Save As…. Nel caso in cui l'ObjectOutputStream fallisca nella scrittura del file viene mostrato all'utente un messaggio di errore.

Quando l'utente seleziona "Save As…" viene mostrato un JFileChooser tramite il quale l'utente può scegliere la directory e il nome del file da salvare. Una volta scelto il file, la variabile di salvataggio viene aggiornata e viene aperto un ObjectOutputStream che provvederà a scrivere il file. Il titolo della finestra principale viene aggiornato con il nome del file in uso e la variabile `unsavedChanges` impostata a false. Nel caso in cui l'ObjectOutputStream fallisca nella scrittura del file viene mostrato all'utente un messaggio di errore.

Quando l'utente seleziona "Exit" o clicca sul pulsante di chiusura della finestra principale viene chiamato il metodo `private boolean confirmExit()`: se sulla lista sono state apportate modifiche e questa non è stata salvata, mostra all'utente un OptionDialog dal quale l'utente deciderà se salvare le modifiche e uscire dall'applicazione, uscire dall'applicazione senza salvare oppure annullare l'operazione; tramite un blocco switch – case il metodo gestisce la scelta dell'utente: se questo sceglie di salvare vengono eseguite le stesse operazioni di Save / Save As… e il metodo ritorna true; se l'utente sceglie di non salvare il metodo ritorna true; il metodo ritorna false se l'utente sceglie di annullare l'uscita, se chiude la finestra di dialogo, se annulla il salvataggio durante l'uso del JFileChooser oppure se si verifica un errore durante il salvataggio del file. Se il metodo `confirmExit()` ritorna true (ovvero è stato dato il consenso di uscita) la finestra viene resa non visibile, prima di essere distrutta con `dispose()` e il programma terminato.

Il metodo `private void menuD()` viene chiamato ogni qual volta la lista diventa vuota e disabilita gli elementi dei menu relativi alle operazioni non consentite su una lista vuota (get, rimozione, svuotamento, ricerca, ordinamento e iterazione a partire da una posizione a scelta).

Il metodo `private void menuE()` viene invece chiamato in seguito alle operazioni che comportano un riempimento della lista e abilita gli elementi dei menu disabilitati `menuD()`.

L'esecuzione del programma viene affidata all'Event Dispatch Thread di AWT.

DATA: 22/01/2019