```java
import java.util.*;
/**
 * Una matrice bi-dimensionale di oggetti di tipo T con un numero fissato
 * di righe ed un numero variabile di colonne (tutte le colonne hanno lo
 * stesso numero di posizioni).
 */
public class BiArrayList<T> {
    // campi
    private int rows;
    private int cols;
    // ArrayList di ArrayList, con un array normale non riesco a gestire i gene
rics
    private ArrayList<ArrayList <T>> matrice;
    /*
     *         row 1                   row 2                       row n
     * [ [col1][col2][col3] ] [ [col1][col2][col3] ] ... [ [col1][col2][col3] ]
     */

    /**
     * @pre : n > 0
     * @post: crea una BiArrayList vuota, con n righe e 0 colonne
     */
    public BiArrayList(int n) {
        rows = n;
        cols = 0;
        // inizializzo l'array
        matrice = new ArrayList<ArrayList<T>>();
        // e n ArrayList vuoti = n righe, 0 colonne
        for (int r = 0; r < n; r++)
            matrice.add(new ArrayList<T>());
    }

    /**
     * @pre: TRUE, @result = numero di righe di this
     */
    public int rows() {
        return rows;
    }

    /**
     * @pre: TRUE, @result = numero di colonne di this
     */
    public int cols() {
        return cols;
    }

    /**
     * @pre: c != null && c.length == rows()
     * @post: aggiunge c come ultima colonna di this.
     * Lancia l'eccezione se la precondizione e' violata
     *
     */
    public void insert (T[] c) throws IllegalArgumentException {
        // verifica le precondizioni
        if(c == null || c.length != rows())
            throw new IllegalArgumentException();
        // scorri tutta la matrice, aggiungendo ad ogni riga una colonna
        // per ogni ArrayList colonna (matrice.get(r)) aggiungi un elemento
        int r;
        for(r = 0; r < rows(); r++) {
            matrice.get(r).add(c[r]);
        }
        cols++;
    }

    /**
     * @pre: x != null & 0 <= i < rows() && 0 <= j < cols()
```

```
     * @post: modifica la posizione (i,j) della matrice settandola
     * ad x. Eccezione se la precondizione e' violata
     *
     */
    public void set(int i, int j, T x) throws IllegalArgumentException {
        // verifica le precondizioni
        if(x == null || (i < 0 || i >= rows()) || (j < 0 || j >= cols()))
            throw new IllegalArgumentException();
        matrice.get(i).set(j, x);

    }
    /**
     * @pre: 0 <= i < rows() && 0 <= j < cols()
     * @result = il valore memorizzato in posizione (i,j)
     * @post: Eccezione se la precondizione e' violata
     */
    public T get(int i, int j) throws IllegalArgumentException {
        // verifica le precondizioni
        if((i < 0 || i >= rows()) || (j < 0 || j >= cols()))
            throw new IllegalArgumentException();
        return matrice.get(i).get(j);
    }

    /**
     * @result = un iteratore che restituisce gli elementi di this
     * enumerandoli in ordine di riga e di colonna
     *
     */
    public Iterator<T> iterator() {
        // creo un nuovo array list a cui aggiungo gli elementi riga per colonn
a
        int r, c;
        List<T> all = new ArrayList<T>();
        for(r = 0; r < rows; r++)
            for(c = 0; c < cols; c++)
                all.add(matrice.get(r).get(c));
        return all.iterator();
    }
}
```

```java
import java.util.*;

class UndoBuffer extends Buffer {
    // classe interna Backup, associa all'indice modificato il valore prima del
la modifica
    private class Backup {
        private int i;
        private char c;
        Backup(int _i, char _c) {
            i = _i;
            c = _c;
        }
    }
    // campo cronologia mantiene la sequenza di Backup
    List<Backup> cronologia = new LinkedList<Backup>();

    // costruttore
    public UndoBuffer(int n) {
        super(n);
    }

    // metodi
    // override di set
    public void set(int i, char c) throws ArrayIndexOutOfBoundsException {
        // salva nella cronologia il valore nell'indice prima della modifica
        // (in testa, così da poterlo rimuovere senza conoscere la lunghezza de
lla cronologia)
        cronologia.add(0, new Backup(i, get(i)));
        // aggiungi il nuovo valore nel buffer
        super.set(i, c);
    }

    public void undo() {
        // se non è mai stato effettuato un set non fare niente
        if (cronologia.size() <= 0)
            return;
        // tolgo l'ultimo backup dalla cronologia salvandolo in una variabile
        Backup backup = cronologia.remove(0);
        // ripristino l'ultimo elemento modificato con il valore di backup
        super.set(backup.i, backup.c);
    }

}
```

```java
import java.util.*;

public class MoreDataStat extends DataStat {
    // campi
    private List<Measurable> misurati = new ArrayList<Measurable>(); // Lista d
i tutti gli oggetti Measurable aggiunti
    private Measurable min; // l'oggetto Measurable più corto aggiunto finora

    // metodi
    // override di add, oltre a memorizzare il massimo (super.add()), memorizzo
 il minimo
    // inoltre aggiungo l'oggetto in misurati
    public void add(Measurable x) {
        // verifico prima che il contatore venga incrementato
        if (count() == 0 || min.measure() > x.measure())
            min = x;
        // non incremento count perché viene già incrementato in DataStat
        misurati.add(x);
        super.add(x);
    }

    /**
     * @pre true
     * @post @nochange
     * @result = l'ampiezza dell'intervallo delle misure considerate,
     * ovvero la differenza tra le misure degli elementi con misura
     * massima e minima.
     */
    public double size() {
        return max().measure() - min.measure();
    }

    /**
     * @pre d >= 0
     * @post @nochange
     * @result = un iteratore che permette di ottenere in sequenza gli
     * elementi la cui misura e' minore del valore d
     */
    public Iterator<Measurable> lessThan(double d) {
        // creo un nuovo ArrayList
        List<Measurable> selezionati = new ArrayList<Measurable>();
        // scorro gli elementi misurati aggiungendoli a selected solo se inferi
ori a d
        for(Measurable m : misurati)
            if(m.measure() < d)
                selezionati.add(m);
        // restituisco l'iteratore di selezionati
        return selezionati.iterator();
    }

}
```

```java
import java.util.*;
/*
 * la classe memorizza in un ArrayList le operazioni eseguite
 * in caso di undo, elimina tutti gli elementi dalla coda
 * ripete le operazioni della coda ad eccezione dell'ultima
 * che viene cancellata dalla cronologia
 */

// è importante che il parametro T venga specificato per entrambe le classi
class UQueue<T> extends Queue<T> {
    // classe interna che identifica un'operazione
    class Operation {
        // attributi
        // isEnqueue true se è un'operazione di enqueue, false se è di dequeue
        boolean isEnqueue;
        // value = valore T utilizzato dall'operazione
        T value;
        // costruttore
        Operation(boolean _isEnqueue, T _value) {
            isEnqueue = _isEnqueue;
            value = _value;
        }
    }

    // attributi
    // cronologia mantiene la sequenza di operazioni in una collezione
    List<Operation> cronologia = new ArrayList<Operation>();

    // costruttore ereditato da Queue

    // sovrascrittura dei metodi enqueue e dequeue di Queue
    // esegui i metodi e salva l'operazione eseguita in cronologia
    public void enqueue(T elem) {
        cronologia.add(new Operation(true, elem));
        super.enqueue(elem);
    }

    public T dequeue() {
        // eseguo il metodo delete di Queue salvandomi il valore dell'elemento
cancellato da memorizzare in cronologia
        T mem = super.dequeue();
        cronologia.add(new Operation(false, null));
        // torno l'elemento come previsto dal metodo di Queue
        return mem;
    }

    public void undo() {
        // indice dell'ultima operazione
        int i = cronologia.size() - 1;
        // se la cronologia è vuota esco
        if(i < 0) return;
        // azzera la coda con il dequeue di Queue, perché non devo salvare in c
ronologia queste operazioni
        int cont;
        int lQueue = size(); // lunghezza lista data da size(), metodo ereditat
o
        for (cont = 0; cont < lQueue; cont++)
            super.dequeue();
        // rimuovi l'ultimo elemento dalla cronologia
        cronologia.remove(i);
        // ripeti ogni operazione eseguita distinguendo il tipo di operazione
        for (Operation op : cronologia) {
            if (op.isEnqueue)
                super.enqueue(op.value);
            else
                super.dequeue();
        }
```

```
class Const implements BoolExp {
    // campo con un boolean che indica se vero o falso
    boolean bool;

    // costruttore
    Const(boolean _bool) {
        bool = _bool;
    }

    // metodi
    public boolean eval() {
        return bool;
    }

    public String toString() {
        if(bool)
            return "TRUE";
        return "FALSE";
    }
}

class Cond implements BoolExp {
    // campi con tre espressioni booleane
    BoolExp ifb;
    BoolExp thenb;
    BoolExp elseb;

    // costruttore
    Cond(BoolExp _ifb, BoolExp _thenb, BoolExp _elseb) {
        ifb = _ifb;
        thenb = _thenb;
        elseb = _elseb;
    }

    // metodi
    public boolean eval() {
        // se è vera l'espressione booleana del if, torna quella di then, altri
menti quella di else
        if(ifb.eval())
            return thenb.eval();
        return elseb.eval();
    }

    public String toString() {
        if(ifb.eval())
            return thenb.toString();
        return elseb.toString();
    }
}

class TestBool {
    // definisco due valori V F
    public static final BoolExp TRUE = new Const(true);
    public static final BoolExp FALSE = new Const(false);

    public static void main(String[] argv) {
        // if a then (if b then FALSE else TRUE) else FALSE
        // definisco a e b, a caso
        BoolExp a = TRUE;
        BoolExp b = FALSE;
        // definisco il primo if, (if b then FALSE else TRUE)
        BoolExp if1 = new Cond(b, FALSE, TRUE);
        // definisco il secondo if, if a then (primo if) else FALSE
        BoolExp if2 = new Cond(a, if1, FALSE);
        // primo if: b è false, entra in else, torna true
        // secondo if: a è true, entra in then, torna primo if cioè true
        System.out.println(if2.toString());
```

```java
import java.util.*;

interface M {
    M m();
}

interface K {
    void k();
}

class A implements M, K {
    public M m() {
        System.out.print(" A "); return this;
    }
    public void k() {
    }
}

class B extends A {
    public M m() {
        System.out.print(" B ");
        return super.m();
    }
}

class Ese3 {
    public static void main(String[] argv) {
        /*
        M a = new B(); // tipo statico = M, tipo dinamico = B
        B b = ((B)a).m();
        // tipo statico = B
        // tipo dinamico M, perché il metodo m torna un M
        // non è possibile assegnare a b di tipo B un oggetto di tipo M (che no
n sappiamo se è B)
        // non compila

        M a = new B(); // tipo statico = M, tipo dinamico = B
        K b = (K)(a.m());
        // tipo statico = K, tipo dinamico K
        // il metodo m è definito in M e torna a sua volta M
        // il casting da M a K è accettato perché M è un interfaccia (il castin
g da interfaccia a classe compila sempre)
        // compila
        // in esecuzione si usa il metodo del tipo dinamico B
        // B richiama a sua volta il metodo m di A, che torna se stesso quindi
un B, che è anche A perché B estende A
        // il casting da tipo A all'interfaccia K viene eseguito, si "maschera"
 l'oggetto con la sua interfaccia
        // esegue stampa "B A"

        M a = new A(); // tipo statico = M, tipo dinamico = A
        B b = (B)(a.m());
        // tipo statico = B, tipo dinamico = B
        // il metodo m è definito in M e torna a sua volta M
        // il casting da M a B è accettato perché M è un interfaccia
        // compila
        // in esecuzione si usa il metodo m di A, che torna un oggetto di tipo
dinamico A
        // il casting da A (superclasse) a B (sottoclasse) non è accettato
        // non esegue
        */
    }
}
```

```java
import java.util.*;

public abstract class Resource {
    // campi
    private String name;
    private String owner;

    // costruttori
    Resource() {
        // default constructor setta i campi a stringa vuota
    }

    Resource(String _name, String _owner) {
        name = _name;
        owner = _owner;
    }

    // lascio i metodi protected, per default
    String name() {
        return name;
    }

    String owner() {
        return owner;
    }
    // metodo da implemetare in File e Directory
    abstract int size();
}

public class File extends Resource {
    // campi
    private String contents;

    // costruttore
    public File(String name, String owner, String _contents) {
        super(name, owner);
        contents = _contents;
    }
    // implementazione del metodo size
    // supponiamo siano tutti file di testo, quindi il peso è 1 byte x lettera
    public int size() {
        return contents.length();
    }
}

public class Directory extends Resource {
    // campi
    private List<Resource> contents;

    // costruttore
    public Directory(String name, String owner) {
        super(name, owner);
        contents = new LinkedList<Resource>();
    }

    // metodo per inserire un file o directory nella directory
    protected void add(Resource r) {
        contents.add(r);
    }
    // implementazione del metodo size
    public int size() {
        int dim = 0;
        for (Resource r : contents)
            // esegui il metodo size di ogni risorsa, nel caso di direttory il
metodo size scorre i file della directory
            dim += r.size();
        return dim;
```