LEZIONE 1

La classe String presenta i seguenti metodi:

String	**concat**(String str) Concatenates the specified string to the end of this string.
int	**length**() Returns the length of this string.
String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**trim**() Returns a copy of the string, with leading and trailing whitespace omitted.

String concat(String str)
Int length()
String substring(int beginIndex) oppure String substring(int beginIndex, int endIndex)
String toLowerCase() oppure String toUpperCase()
String trim() ritorna una copia della stringa, omettendo gli spazie.

char <-> int : codice ASCII del carattere
Esempio

```
char c; int d;
c = (char) 65;
d = (int) 'A';
```

&& and
|| or
! not
Espressione condizionale:

```
(espressione booleana)?<valore1>:<valore2>
```

Per ogni tipo primitivo esiste la corrispondente classe associata
(denominata classe wrapper)

Byte	short	int	long	float	double	char	boolean
Byte	Short	Integer	Long	Float	Double	Character	Boolean

Esempio

```
String s = JOptionPane.showInputDialog("Inserisci un numero");
int x = Integer.parseInt(s);
double x = Double.parseDouble(s);
```

Esempio

```
int mese=…
int giorniDelMese;
switch (mese) {
        case 4: case 6: case 9: case 11:
        giorniDelMese = 30; break;
        case 1: case 3: case 5: case 7: case 8:
        case 10: case 12:
        giorniDelMese = 31; break;
```

```
            case 2:
            giorniDelMese = 28;
            break;
}
```

_ _

LEZIONE 2

Valori costanti possono essere definiti all'interno di una classe (fuori dalle definizioni dei metodi) con la definizione di una variabile che assume un valore con non può essere modificato (final)

```
public class Utilita {
      public static final int ZERO=0;
….
}
```

Esempio ARRAY: Porzione di codice che calcola la somma dei quadrati dei numeri interi da 0 a 99;

```
int[] v = new int[100];
for (int i=0; i < v.length; i++)
    v[i] = i*i;
int somma = 0;
for (int i=0; i < v.length; i++)
somma += v[i];
```

Matrici:

```
tipo[][] nomeMatrice;
nomeMatrice.length // numero di righe
nomeMatrice[0].length // numero di colonne
```

Somma di due matrici

```
public static double[][] sommaMatrici(double[][] A, double[][] B) {
    double[][] C = new double [A.length][A[0].length];
    for (int i=0; i<A.length; i++)
       for (int j=0; j<A[0].length; j++)
          C[i][j] = A[i][j] + B[i][j];
    return C;
}
```

Inizializzazione di array e matrici: Array e matrici sono inizializzati automaticamente con valori di default associati al tipo di dato della struttura.

valori numerici	0
char	'\0'
boolean	false
oggetti	null

InputStream: E' il capostipite degli stream di input per i byte
OutputStream: E' il capostipite degli stream di output per i byte
Reader: E' il capostipite degli stream di input per i caratteri
Writer: E' il capostipite degli stream di output per i caratteri

Gestione tastiera: soluzione completa

• Per gestire correttamente la tastiera useremo quindi una sequenza di istruzioni di questo tipo:

BufferedReader kbd = new BufferedReader(new InputStreamReader(System.in));

Gestione del video: soluzione completa
• *PrintWriter video = new PrintWriter(System.out);*

File di testo
• Possiamo leggere e scrivere file di testo utilizzando stream di caratteri
• In particolare:
– La classe FileReader (derivata da Reader) ci permette di leggere un file di testo
– La classe FileWriter (derivata da Writer) ci permette di scrivere in un file di testo

FileInputStream/FileOutputStream
• Sono sottoclassi di InputStream e OutputStream
• Aprono stream di byte da/verso file
• Hanno gli stessi metodi di InputStream e OutputStream
• Si possono applicare i filtri (ad esempio DataInputStream e DataOutputStream)
• Costruttori:
– public FileInputStream(File file) throws FileNotFoundException
– public FileInputStream(String name) throws FileNotFoundException

FileReader/FileWriter
• Sono sottoclassi di Reader e Writer
• Aprono stream di caratteri da/verso file
• Hanno gli stessi metodi di Reader e Writer
• Si possono applicare BufferedReader e BufferedWriter
• Costruttori:
– *public FileReader(File file) throws FileNotFoundException*
– *public FileReader(String name) throws FileNotFoundException*
– *public FileWriter(File file) throws FileNotFoundException*
– *public FileWriter(String name) throws FileNotFoundException*
– *public FileWriter(String name, boolean append) throws FileNotFoundException*

La classe File
• Rappresentazione astratta di file e directory
• Metodi per manipolare file e directory (creare, distruggere ecc.)
• Non contiene metodi per leggere/scrivere
• Costruttori:
– *public File(String pathname) throws NullPointerException*
– *public File(File parent, String child) throws NullPointerException*
– *public File(String parent, String child) throws NullPointerException*

• Campi più importanti:
– *public static final char separatorChar*: restituisce il carattere (dipendente dal sistema operativo) che separa i nomi nelle directory
• Metodi più importanti:
– *public boolean delete() throws SecurityException*
– *public boolean exists() throws SecurityException*
– *public String getAbsolutePath()*
– *public String getName()*
– *public boolean isDirectory()*
– *public boolean isFile()*
– *public long length()*

– *public String[] list()*
– *public File[] listFiles()*
– *public static File[] listRoots()*
– *public boolean mkdir() throws SecurityException*
– *public boolean mkdirs() throws SecurityException*
• *SecurityException è sottoclasse di RuntimeException*

I/O generico su stream di caratteri

• L'organizzazione gerarchica degli stream consente la definizione di metodi di I/O generici rispetto al particolare dispositivo di I/O usato
• Ad esempio, il seguente metodo

```
public static void scrivi(PrintWriter o, String s) {
        o.println(s);
}
```

può essere usato per scrivere una stringa su un qualunque stream di output o.

• Diverse invocazioni effettueranno diverse operazioni di I/O

```
scrivi(new PrintWriter(System.out), "Ciao");                // stampa su schermo
scrivi(new PrintWriter(new FileWriter(f)), "Ciao");         // scrive su file
```

Accesso e lettura del file

• Per accedere al file possiamo utilizzare la classe FileReader
• Il costruttore di questa classe prende come parametro il nome del file da leggere e lo apre in lettura
• FileReader è però uno stream di dati e offre solo le funzionalità base: lettura a caratteri singoli
• Sappiamo però come risolvere questo problema: ricorriamo allo stream di manipolazione BufferedReader e lo agganciamo al FileReader:

```
FileReader fr = new FileReader(“inventory.dat”);
BufferedReader inFile = new BufferedReader(fr);
```

• Possiamo quindi utilizzare inFile.readLine() per leggere le righe del file
• Quando il file termina readLine() restituisce null

StringTokenizer

• Ci rimane però un problema: all’interno di ogni riga abbiamo più informazioni separate da spazi e quindi dobbiamo scomporla
• La classe StringTokenizer, inclusa nel package java.util svolge proprio questo compito
• Il costruttore prende come parametro la stringa da scomporre e con il metodo nextToken() possiamo estrarre le singole sottostringhe e convertirle:

```
...
tokenizer = new StringTokenizer (line);
name = tokenizer.nextToken();
units = Integer.parseInt (tokenizer.nextToken());
price = Float.parseFloat (tokenizer.nextToken());
```

Scrittura su un file di testo:

```
import java.io.*;
public class Tabelline{
        public static void main (String[] args) throws IOException {
                FileWriter fw = new FileWriter("tabelline.txt");
                PrintWriter outFile = new PrintWriter(fw);
                for (int i=1; i<=10; i++){
                        for (int j=1; j<=10; j++){
                                outFile.print((i*j)+" ");
                        }
```

```
                outFile.println();
                }
        outFile.close();
        }
}
```

_ _

LEZIONE 3 -CLASSI

I modificatori di accesso consentono di definire la visibilità (accesso) dei campi (dati e operazioni) definiti all'interno di una classe.
In Java esistono quattro tipi di modificatori di accesso
• public – accesso da qualsiasi altra classe
• modificatore di default – accesso da tutte le classi del package
• protected – accesso da tutte le classi derivate
• private – accesso solo dalla classe in cui il campo è definito

Regola generale	
Campi dati (variabili di istanza)	non public/tipicamente private
Campi operazioni (metodi) di servizio	public
Campi operazioni (metodi) di supporto	private

Costruttori
Metodi che creano (allocano memoria per) gli oggetti della classe e inizializzano le variabili di istanza.
• Sono metodi (non static) di una classe
• Hanno lo stesso nome della classe
• Non hanno un tipo di ritorno esplicito (neanche void).
Note:
• una classe può avere più metodi costruttori
• se non viene specificato nessun costruttore, viene assegnato un costruttore di default senza argomenti

In caso di omonimia delle variabili di istanza e delle variabili o dei parametri formali dichiarati nel metodo, **this** consente di discriminare una variabile di istanza da una variabile locale.

Il tentativo di applicare metodi di istanza ad una variabile che non si riferisce ad un oggetto (quindi che contiene il valore null) genera un errore a run-time di tipo NullPointerException.

ERRORI

• La superclasse Throwable ha due sottoclassi dirette, sempre in java.lang
– Error
• Errori fatali, dovuti a condizioni incontrollabili
– Esaurimento delle risorse di sistema necessarie alla JVM, incompatibilità di versioni
• In genere i programmi non gestiscono questi errori
– Exception
• Tutti gli errori che non rientrano in Error
• I programmi possono gestire o no questi errori a seconda dei casi

Una eccezione è un evento che interrompe la normale esecuzione del programma
• Se si verifica un'eccezione il metodo trasferisce il controllo ad un gestore delle eccezioni
– Il suo compito è quello di uscire dal frammento di codice che ha generato l'eccezione e decidere cosa fare
• Java mette a disposizione varie classi per gestire le eccezioni, in vari package, ad es., java.lang, java.io

• Tutte le classi che gestiscono le eccezioni sono ereditate dalla classe Exception

• Esempio di eccezione controllata
– EOFException: terminazione inaspettata del flusso di dati in ingresso
– Può essere provocata da eventi esterni
• errore del disco
• interruzione del collegamento di rete
– Il gestore dell'eccezione si occupa del problema
• Esempi di eccezione non controllata
– NullPointerException: uso di un riferimento null
–IndexOutofBoundException: accesso ad elementi esterni ai limiti di un array
• Non bisogna installare un gestore per questotipo di eccezione
– Il programmatore può prevenire queste anomalie, correggendo il codice

Eccezioni controllate
• Tutte le sottoclassi di IOException
– EOFException
– FileNotFoundException
– MalformedURLException
– UnknownHostException
– ClassNotFoundException
– CloneNotSupportedException

Eccezioni non controllate
• Tutte le sottoclassi di RunTimeException
– ArithmeticException
– ClassCastException
– IllegalArgumentException
– IllegalStateException
– IndexOutOfBoundsException
– NoSuchElementException
– NullPointerException

Catturare eccezioni
• Vengono eseguite le istruzioni all'interno del blocco **try**
• Se nessuna eccezione viene lanciata, le clausole **catch** sono ignorate
• Se viene lanciata una eccezione viene eseguita la corrispondente clausola catch

Dettagli su Exception
• Costruttori di oggetti della classe Exception:
– Exception() costruisce un'eccezione senza uno specifico messaggio
– Exception(String msg) costruisce un'eccezione con il messaggio msg
• Metodo ereditati dalla classe Throwable:
– String getMessage() restituisce come stringa il messaggio contenuto nell'eccezione
• Quando l'eccezione viene lanciata:
– può essere già stata creata o venir creata contestualmente
• Garbage collector: distrugge solo le eccezioni che sono state catturate

Per lanciare un'eccezione, usiamo la parola chiave throw (lancia), seguita da un oggetto di tipo eccezione
throw exceptionObject;
• Il metodo termina immediatamente e passa il controllo al gestore delle eccezioni
– Le istruzioni successive non vengono eseguite

Per segnalare le eccezioni controllate che il metodo può lanciare usiamo la parola chiave throws

```
public void read(BufferedReader in) throws IOException
```

• Un metodo può lanciare più eccezioni controllate, di tipo diverso

```
public void read(BufferedReader in)  throws IOException, ClassNotFoundException
```

Per avere un messaggio di errore che stampa lo stack delle chiamate ai metodi in cui si è verificata l'eccezione usiamo il metodo printStackTrace()

```
catch(DivisionePerZeroException exception) {
        exception.printStackTrace();
}
```

A volte vogliamo eseguire altre istruzioni prima dell'arresto
• La clausola finally viene usata per indicare un'istruzione/blocco che va eseguita/o sempre
– Ad, esempio, se stiamo leggendo un file e si verifica un'eccezione, vogliamo comunque chiudere il file

_ _

LEZIONE 4 EREDITARIETA' E POLIMORFISMO

Definizione di una classe che è una specializzazione di una classe già esistente e ha nuove funzionalità (comportamenti) e/o nuove informazioni.
Invece di modificare la classe già definita si crea una nuova classe derivata da essa.

```
public class Studente extends Persona{
…..
}
```

• Una sottoclasse eredita tutti i metodi e le variabili di istanza della superclasse, ed in più può avere metodi e variabili di istanza specifici.
• Metodi e variabili delle classi base seguono le regole di visibilità anche per le classi derivate. Campi privati di una classe base non sono visibili in una classe derivata.

• Tutte le proprietà (variabili di istanza e metodi) definite per la classe base vengono implicitamente definite anche nella classe derivata, cioè vengono ereditate da quest'ultima.
• La classe derivata può avere ulteriori proprietà rispetto a quelle ereditate dalla classe base.
• Ogni oggetto della classe derivata è anche un oggetto della classe base, e ciò implica che è possibile usare un oggetto della classe derivata in ogni situazione o contesto in cui si può usare un oggetto della classe base.

super(...) invoca il costruttore della classe base, deve essere la prima istruzione del metodo, se non presente si assume la chiamata super(), cioè al costruttore di default (se non esiste il compilatore segnala errore.

ESEMPIO

```
public class Studente extends Persona {
        // campi dati aggiuntivi
        private String matricola;
        // campi operazione aggiuntivi
        public Studente (String n, int e, String m) {
```

```
        // inizializzazione delle variabili della classe base
        super(n,e);
        // inizializzazione delle variabili della classe derivata
        matricola = m;
}
```

• Ogni oggetto della classe derivata è anche un oggetto della classe base, quindi è possibile usare un oggetto della classe derivata in ogni situazione o contesto in cui si può usare un oggetto della classe base.
• Gli oggetti della classe derivata sono compatibili con gli oggetti della classe base. Ma non è vero il viceversa!

```
Persona p1 = new Persona("Mario", 33);
Studente s1 = new Studente("Maria", 24, "12345678");
Persona p2 = s1; // OK
Studente s2 = p1; // Errore
```

• La compatibilità tra oggetti delle classi base con quelle delle classi derivate vale anche nel passaggio dei parametri

```
public static void stampaPersona (Persona p) { … }
public static void stampaStudente (Studente s) { … }
public static void main(String args[]) {
        Persona p1 = new Persona("Mario", 33);
        Studente s1 = new Studente("Maria", 24, "12345678");
        stampaPersona (s1); // OK
        stampaStudente(p1); // Errore
}
```

Campi dati o operazione dichiarati con il modificatore di accesso protected sono visibili solo alle classi derivate. Uso di protected solo in casi particolari (diminuisce la robustezza).

Polimorfismo: presenza di metodi con la stessa segnatura che si comportano in modo diverso all'interno di una gerarchia di classi

In Java, per i metodi statici si usa binding statico, mentre per i metodi di istanza si usa binding dinamico.

```
public static void main(String[] args) {
        Studente s1 = new Studente("Maria Verdi",24,"1234");
        Persona p2 = s1;
        p2.stampa(); // binding dinamico
}

public static void stampa(Persona p) { … }
public static void stampa(Studente s) { … }
public static void main(String[] args) {
        Studente s1 = new Studente("Maria Verdi",24,"1234");
        Persona p2 = s1;
        stampa(p2); // binding statico
}
```

UGUAGLIANZA E COPIA DI OGGETTI

Tutte le classi Java sono sottoclassi della classe Object, anche se non viene indicato esplicitamente.
Classe Object
• Classe Class
• Uguaglianza
– Uguaglianza superficiale: operatore ==
– Uguaglianza profonda: funzione equals
• Copia di oggetti
– Copia superficiale: operatore =
– Copia profonda: funzione clone
protected Object clone()
// Crea e restituisce una copia di questo oggetto.
public boolean equals(Object obj)
// Verifica se il contenuto di questo oggetto è "uguale" a obj.
public Class getClass()
// Restituisce la classe a runtime a cui appartiene l'oggetto.
public String toString()
// Restituisce una stringa che rappresenta l'oggetto

toString() viene invocato automaticamente nelle operazioni di stampa

Tutti gli oggetti sono compatibili con la classe Object.

```
Object o = new Persona("Mario", 32);
```

Attenzione:

```
System.out.println(o); // OK (polimorfismo)
System.out.println(o.getEta()); // Errore
```

Esempio:

```
Object o = new Persona("Mario", 32);
Persona p = (Persona) o;
System.out.println(p); // OK
System.out.println(p.getEta()); // OK
```

• Class è una classe predefinita in Java i cui oggetti corrispondono alle classi definite in un programma
• Esiste implicitamente un oggetto di classe Class per ogni classe A (di libreria o definita da utente)
• Questo oggetto può essere denotato in due modi:
– tramite la classe A con l'espressione: A.class di tipo Class
– tramite oggetti della classe A usando la funzione getClass() di Object,
ad esempio:
a1=new A();
a1.getClass() // espressione di tipo Class

Gli oggetti della classe Class consentono di stabilire a run-time la classe a cui appartiene un oggetto (introspezione)
Esempio

```
Object x = …
Object y = …
boolean r = (x.getClass().equals(y.getClass()); // verifica se i due oggetti x e y appartengono alla stessa classe
```

La classe Class definisce la funzione

boolean isInstance(Object)
che restituisce true se e solo se il suo parametro attuale è un riferimento ad oggetto di una classe compatibile per l'assegnazione con la stessa classe dell'oggetto di invocazione.
Esempio

```
A a1=new A();
A.class.isInstance(a1) // vale true
```

Un oggetto di una classe B derivata da A è compatibile con la classe A (non è vero il viceversa)
Esempio

```
class B extends A …
A a1=new A(); B b1=new B();
A.class.isInstance(b1) // vale true
B.class.isInstance(a1) // vale false
```

Se vogliamo mettere a confronto due valori di un tipo base, usiamo l'operatore di uguaglianza ==

• Se usiamo == per confrontare due oggetti, stiamo verificandone l'uguaglianza superficiale (cioè il confronto tra i valori dei riferimenti agli oggetti).

• La funzione equals può essere usata per confrontare il contenuto di oggetti. La funzione equals di Object si comporta come l'operatore ==

Se desideriamo che per una classe C si possa verificare l'uguaglianza profonda fra oggetti,
– il progettista di C deve fornire l'overriding della funzione equals()
– il cliente di C deve effettuare il confronto fra oggetti usando equals() per l'uguaglianza profonda e == per quella superficiale.

La funzione clone() di Object effettua la copia superficiale (mediante =) di tutti i campi dell'oggetto su cui è invocata

• Se desideriamo che per una classe C si possa effettuare la copia profonda fra oggetti
– il progettista di C deve effettuare l'overriding della funzione clone()
– il cliente di C deve effettuare la copia fra oggetti usando clone() per la copia profonda e = per quella superficiale.

Nella classe String la funzione clone() non è disponibile, ma la copia profonda si può ottenere mediante l'uso del costruttore String(String o)

Esempio

```
C c1;
…
C c2 = new C(c1); // COPIA PROFONDA
```

FUNZIONE	SVILUPPATORE DELLA CLASSE	CLIENT
Uguaglianza superficiale		==
Uguaglianza profonda	Ridefinizione di equals()	equals()
Copia superficiale		=
Copia Profonda	Ridefinizione di clone() Costruttore di copia	clone() Costruttore di copia

CLASSI ASTRATTE E INTERFACCE

Le classi astratte sono classi particolari, nelle quali una o più funzioni possono essere solo dichiarate (cioè si descrive la segnatura), ma non definite (cioè non si specificano le istruzioni).
Esempio: vogliamo associare alla classe Persona una funzione aliquota() che calcola la sua aliquota fiscale, ma il vero e proprio calcolo per una istanza della classe Persona dipende dalla sottoclasse di Persona (ad esempio: pensionato, studente, impiegato, ecc.) a cui l'istanza appartiene.

```
public abstract class Persona {
        // campi dati
        …
        // campi operazione
        …
        public abstract double aliquota();
}
```

Una classe astratta può contenere uno o più metodi astratti (non definiti).
Non è consentito creare oggetti di una classe astratta. Non si possono invocare i metodi astratti, poiché manca il codice.

```
public class Professore extends Persona {
        // campi dati
        …
        // campi operazione
        …
        public double aliquota() {
        return …;
        }
}
```

Sottoclassi (non astratte) di una classe astratta devono implementare tutti i metodi astratti.

Una classe A si dovrà definire astratta quando:
• non ha senso pensare a oggetti che siano istanze di A senza essere istanze di una sottoclasse di A
• esiste una funzione che ha senso associare ad essa, ma il cui codice non può essere specificato a livello di A, mentre può essere specificato a livello delle sottoclassi di A.
Una classe astratta A non ha istanze dirette, ma ha come istanze tutti gli oggetti che sono istanze di sottoclassi di A non astratte.

Se A è una classe astratta:

• Si possono definire e usare variabili si tipo A
• NON si possono creare oggetti di tipo A

Esempio:

```
Persona p = new Persona(…); // ERRORE
Persona p = new Professore(…); // OK
p.aliquota(); // OK grazie al polimorfismo
```

INTERFACCE

Una interfaccia è un'astrazione per un insieme di funzioni pubbliche delle quali si definisce solo la segnatura, e non le istruzioni.
Un'interfaccia viene poi implementata da una o più classi (anche astratte). Una classe che implementa un'interfaccia deve definire o dichiarare tutte le funzioni della interfaccia.
Si dichiarano solo funzioni pubbliche (non definite)
Non si possono dichiarare variabili (a meno che non siano final).

Se I è un'interfaccia, allora possiamo:
• definire una o più classi che implementano I, cioè che definiscono tutte le funzioni dichiarate in I
• definire variabili di tipo I (conterranno riferimenti ad oggetti di classi che implementano I),
• usare i riferimenti di tipo I invocando le funzioni di I su tali riferimenti, sapendo che in esecuzione essi saranno riferimenti ad oggetti che implementano le funzioni in I.
Non possiamo
• creare oggetti di tipo I
• invocare altri metodi degli oggetti riferiti da I diversi da quelli presenti in I

Esempio:

```
Esempio
interface Tassabile {
  double aliquota (); // dichiarazione
}
abstract class Persona implements Tassabile {
  ….
  double aliquota (); // dichiarazione
}
class Professore extends Persona implements Tassabile { double aliquota () { // definizione
    return …;
  }
}
```

L'ereditarietà si può stabilire anche tra interfacce, nel senso che una interfaccia si può definire come derivata da un'altra.
Se una interfaccia J è derivata da una interfaccia I, allora tutte le funzioni dichiarate in I sono implicitamente dichiarate anche in J.
Una classe che implementa J deve anche definire tutte le funzioni di I.

```
interface I { void
  g();
}
interface J { void
  h();
}
```

```
interface M extends I, J { // vale solo per le interfacce
void k();
 }
 class C implements M {
   void g() { ... }
   void h() { ... }
   void k() { ... }
}
```

Interfacce e classi astratte hanno qualche similarità:
• entrambe hanno funzioni dichiarate e non definite;
• non esistono istanze di interfacce,
• non esistono istanze dirette di classi astratte.
Si tenga però presente che:
• una classe astratta è comunque una classe, ed è quindi un'astrazione di un insieme di oggetti (le sue istanze)
• una interfaccia è un'astrazione di un insieme di funzioni.

* Solo campi dati final

	class	abstract	interface
Riferimenti	SI	SI	SI
Oggetti	SI	(indirettamente)	NO
Campi dati	SI	SI	NO*
Funzioni solo dichiarate	NO	SI	SI
Funzioni definite	SI	SI	NO
extends (abstract) class	1	1	NO
implements interface	>= 0	>= 0	NO
extends interface	NO	NO	>= 0

Interfacce Listener in una GUI

```
public interface MouseListener {
// Invocato quando si clicca sul componente associato
void mouseClicked(MouseEvent e);
// Invocato quando il puntatore del mouse entra nell'area del componente
void mouseEntered(MouseEvent e);
// Invocato quando il puntatore del mouse esce dall'area del componente
void mouseExited (MouseEvent e);
// Invocato quando un pulsante del mouse viene premuto sul componente
void mousePressed(MouseEvent e);
// Invocato quando un pulsante del mouse viene rilasciato sul componente
void mouseReleased(MouseEvent e);
 }
```
