ESERCITAZIONI SU JAVA

PER IL CORSO DI SISTEMI PER L'ELABORAZIONE DELL'INFORMAZIONE

Claudia Picardi
Dipartimento di Informatica - C.so Svizzera 185 -10149 - Torino
Tel. 0116706818 - e-mail: picardi@di.unito.it

PROGRAMMA RIASSUNTIVO DELLE ESERCITAZIONI

1. Iniziare a programmare in java

- il compilatore `javac` e l'interprete `java`
- il metodo `main` e gli argomenti del programma
- creare ed utilizzare oggetti di classi esistenti:
  - `String`
  - `StringBuffer`
  - importare un package

2. Creare una classe

- membri dati, metodi e costruttori
- ereditarietà
  - overriding
  - overloading

3. Utilizzo dei componenti grafici

- Aprire e chiudere la finestra principale del programma
- Inserire componenti predefiniti nella finestra principale
- Disporre i componenti: i Layout Manager
  - `FlowLayout`
  - `GridLayout`
  - `BorderLayout`
  - `BoxLayout`
- Utilizzare i pannelli `JPanel` per disporre i componenti nel modo desiderato
- Componenti predefiniti:
  - `JButton` (pulsante)
  - `JTextField` (casella di testo)
  - `JLabel` (etichetta)
  - `JCheckBox` (casella di scelta, tipo non esclusivo)
  - `JRadioButton` (casella di scelta, tipo esclusivo)

4. Interazione con l'utente: eventi

- Implementare un'interfaccia
- Eventi specifici e loro interfacce:
  - `ActionEvent` e `ActionListener` (per pulsanti e caselle di testo)
  - `WindowEvent` e `WindowListener` (per uscire dal programma alla chiusura della finestra)
  - `ItemEvent` e `ItemListener` (per caselle di scelta)
  - `MouseEvent` e `MouseListener` (per gestire il mouse)
- Le classi `Adapter`:
  - `WindowAdapter`
  - `MouseAdapter`

5. Creazione di un nuovo componente

- Estendere la classe `JComponent`
- Disegnare il componente:
  - il metodo `paint`
  - gli oggetti `Graphics` e `Graphics2D`
  - il colore (`Color`) e il tratto (`BasicStroke`)
  - le forme (`Rectangle2D`, `RoundRectangle2D`, `Ellipse2D`, `Arc2D`, `Line2D`)
- Rendere il componente attivo ricevendo i `MouseEvent`.

ESERCITAZIONI

1. Compilare ed eseguire un programma Java

Un programma Java è costituito dalla descrizione di una o più classi. **Ciascuna classe deve essere definita in un file differente.** I file che costituiscono il programma devono avere estensione `<.java>`; **il nome del file deve essere uguale al nome della classe descritta al suo** interno.
Così ad esempio la classe `Farfalla` sarà definita in un file chiamato `Farfalla.java`.
Una delle classi che compongono il programma è la **classe principale**. Questa è la classe in cui è implementato il metodo `main`, ossia il "punto di partenza" del programma.
Il compilatore di Java viene eseguito dalla console dei comandi, che in Windows si chiama anche "Prompt di DOS".
Per compilare i file Java **bisogna trovarsi nella cartella in cui si trovano i file da compilare**. Per spostarsi da una cartella ad un'altra si usa il comando `cd` seguito dal nome della cartella in cui ci si vuole spostare.
Per compilare i file Java si utilizza il comando `javac`. Quindi scriveremo ad esempio:

```
javac Farfalla.java
```

Questo va fatto per ogni file che compone il programma (ovviamente è necessario ri-compilare un file solo se è stato modificato). Se desideriamo compilare tutti i file contemporaneamente possiamo scrivere:

```
javac *.java
```

Nota:

quest'ultimo comando compila tutti i file Java contenuti nella cartella corrente. Quindi se nella stessa cartella ci sono file di due programmi diversi, mescolati tra loro, vengono compilati tutti. **Per i programmi Java è buona norma tenere una cartella separata per ciascun programma.**

Il comando `javac` produce, per ciascun file Java, un file con lo stesso nome ma con l'estensione `<.class>` (ad esempio `Farfalla.class`); questi file sono scritti in un linguaggio chiamato **Java Bytecode**.
Una volta che i file sono stati compilati, il programma può essere **eseguito** utilizzando il comando `java`. **Tale comando deve ricevere come parametro il nome della classe principale**; scriveremo perciò ad esempio:

```
java Farfalla
```

Attenzione che il nome della classe deve essere scritto come all'interno del file, rispettando le maiuscole e minuscole. **Per convenzione in Java i nomi delle classi si scrivono con la prima lettera maiuscola**.

Esempio

Supponiamo di avere scritto un programma Java che serve a catalogare i libri di una biblioteca; il programma è composto dai seguenti file: `Catalogo.java`, `Biblioteca.java`, `Scaffale.java`, `Libro.java`. La classe principale è `Catalogo`. Supponiamo inoltre che il programma si trovi sull'hard disk nella cartella `C:\Java\Biblio`. Ecco la sequenza di comandi per compilare ed eseguire il programma:

```
cd \Java\Biblio
javac *.java
java Catalogo
```

Se a questo punto decidiamo di modificare il file `Catalogo.java` possiamo ricompilare soltanto lui:

```
javac Catalogo.java
java Catalogo
```

2. Un primo semplice programma Java

Il programma che segue si limita a stampare una stringa di testo.

```
class Programma1
{
  public static void main(String[] args)
  {
     System.out.println(“Qualcosa”);
  }
}
```

Note:

a. la classe `Programma1` non ha nessun membro dati, mentre ha un metodo: `main`.

b. il metodo `main` deve per forza essere dichiarato **pubblico** (parola chiave `public`) e **statico** (parola chiave `static`). Deve essere **pubblico** perché deve poter essere chiamato dall'esterno della classe (esso, infatti, sarà chiamato dalla Java Virtual Machine, quel software che lanciamo con il comando `java` e che ha il compito di eseguire i programmi Java). Deve essere **statico** perché i metodi statici possono essere chiamati senza creare nessun oggetto della classe in cui il metodo è definito. Quando eseguiamo un programma gli unici oggetti che esistono sin dall'inizio sono quelli di classi predefinite, ma ovviamente non esiste alcun oggetto della classe definita da noi. Pertanto se `main` non fosse statico non potrebbe essere chiamato.

Quesiti:

Attenzione, per alcuni quesiti potrebbe non essere possibile dare una risposta sulla base del piccolo programma sopra descritto! Nel caso questo succeda, sapreste spiegare perché?
(Suggerimento: abbiamo detto che di solito in Java i nomi delle classi iniziano con lettere maiuscole; i nomi di oggetti e metodi invece iniziano con lettere minuscole.)

Q1. `System` cosa indica? Una classe, un oggetto o un metodo?

Q2. `out` invece? È un oggetto o un metodo? Di quale classe? È pubblico? È statico?

Q3. `println` è evidentemente un metodo. Quanti parametri ha in ingresso? Di che tipo sono?

3. Argomenti del programma

Come visto nell'esempio precedente, il metodo `main` ha come parametro d'ingresso un array di oggetti di tipo `String`. Ciascun elemento dell'array corrisponde ad un argomento passato in input al programma quando viene eseguito. Se ad esempio nell'eseguire `Programma1` scriviamo:

```
java Programma1 qui quo qua
```

l'array `args` conterrà tre elementi, e rispettivamente `args[0]` conterrà la stringa `“qui”`, `args[1]` conterrà la stringa `“quo”`, `args[2]` conterrà la stringa `“qua”`.
Per verificare questo, scriviamo un secondo programma che stampa gli argomenti ricevuti:

```
class Programma2
{
  public static void main(String[] args)
  {
     for(int i=0; i < args.length; i++)
     {
        System.out.println(args[i]);
     }
  }
}
```

Note:

Tutti gli oggetti di tipo array hanno un membro dati chiamato `length` che contiene la lunghezza dell'array stesso. A differenza di quanto succede in altri linguaggi di programmazione, esistono anche gli array vuoti, ossia gli array con `length` pari a zero.

4. Manipolazione di stringhe – parte I

Vogliamo ora scrivere un metodo che rispetti le seguenti caratteristiche:

- prendere in input una stringa (oggetto di tipo `String`);
- restituire una stringa che sia ottenuta dalla prima eliminando le vocali ('a', 'e', 'i', 'o', 'u').

Aggiungeremo dunque alla nostra classe un metodo con la seguente dichiarazione:

```
String eliminaVocali(String s)
```

Il codice finale è molto semplice:

```
class Programma2
{
  public static String eliminaVocali(String s)
  {
     String risultato = new String();
     for(int i=0; i < s.length(); i++)
     {
        char c = s.charAt(i);
        if ((c != 'a') && (c != 'e') && (c != 'i') &&
              (c != 'o') && (c != 'u'))
           risultato = risultato + c;
     }
     return risultato;
  }

  public static void main(String[] args)
  {
     for(int i=0; i < args.length; i++)
     {
        System.out.println(eliminaVocali(args[i]));
     }
  }
}
```

5. Manipolazione di stringhe – parte II

Questo programma tuttavia non è molto efficiente (in altri termini "perde tempo a fare azioni inutili"): vediamo infatti cosa succede quando viene eseguita l'istruzione

```
risultato = risultato + c;
```

Ogni volta che questa istruzione viene eseguita, ossia per ogni carattere che non è una vocale, Java **crea una nuova stringa** formata dalla concatenazione di `risultato` e `c`. Dopodiché **cambia il riferimento** memorizzato in `risultato` in modo che esso "punti" alla nuova stringa invece che a quella vecchia. Insomma: vengono create una marea di stringhe per niente! Sarebbe molto più semplice creare una sola stringa vuota, in cui mano a mano vengono aggiunti i caratteri.
Il motivo per cui Java fa questo è che **un oggetto di tipo `String` non ha nessun metodo che permetta di modificarlo, per cui una volta creato resta così com'è**.
Esiste un'altra classe, chiamata `StringBuffer`, che è in grado di contenere una stringa e permette di modificarla.
Quello che noi vogliamo fare è usare gli `StringBuffer` direttamente. L'idea è di copiare la stringa iniziale in uno `StringBuffer` carattere per carattere, saltando però quei caratteri che corrispondono a lettere vocali.

```
class Programma2
{
  public static String eliminaVocali(String s)
  {
     StringBuffer buf = new StringBuffer();
     for(int i=0; i < s.length(); i++)
     {
        char c = s.charAt(i);
        if ((c != 'a') && (c != 'e') && (c != 'i') &&
              (c != 'o') && (c != 'u'))
           buf.append(c);
     }
     return buf.toString();
  }

  public static void main(String[] args)
  {
     for(int i=0; i < args.length; i++)
     {
        System.out.println(eliminaVocali(args[i]));
     }
  }
}
```

Note:

i) Abbiamo usato due metodi della classe `String`: il metodo `length()` che restituisce la lunghezza della stringa, e il metodo `charAt(int n)` che restituisce l'n-esimo carattere della stringa. Mentre negli array `length` era un membro dati, nelle stringhe è un metodo e dunque deve essere chiamato con le parentesi tonde.
ii) Abbiamo anche usato due metodi della classe `StringBuffer`: si tratta di `append(char c)`, che aggiunge un carattere in fondo al buffer, e `toString()`, che converte il buffer in un oggetto di tipo `String`.

Quesito:

Perché il metodo `eliminaVocali` deve essere dichiarato `static`?

6. Costruttori - parte I

Quesito:

Si consideri la seguente classe:

```
class Lampadina
{
  private int accesa; // 0 significa spenta, 1 significa accesa
  public Lampadina()
  {
     accesa = 0;
     System.out.println(“Costruita!”);
  }
  public void accendi() // accende la lampadina
  {
     accesa = 1;
  }
  public void spegni() // spegne la lampadina
  {
     accesa = 0;
  }
  public boolean seiAccesa()
  {
     return (accesa == 1);
  }
}
```

Come si può notare all’interno del costruttore viene stampata una scritta. Per verificare quando il costruttore viene chiamato si scriva un programma che prende come argomento un numero intero *n* e quindi costruisce *n* lampadine.

Note:

Il programma riceve gli argomenti come oggetti `String`, quindi è necessario convertire una stringa in un numero intero. A questo scopo si scriva un metodo `stringToInt` che prende in input una `String` e restituisce un `int`.

Suggerimento: in Java non è possibile (come invece in C) stabilire il numero rappresentato da un carattere utilizzando il suo codice ASCII, perché in Java i caratteri non sono rappresentati tramite codici ASCII. Quindi ad esempio non è possibile stabilire che `‘9’` rappresenta il numero nove con l’istruzione:

```
char carattere = ‘9’;
int cifra = carattere – ‘0’;
```

Si dovrà quindi adottare un approccio più “stupido”, del tipo:

```
int cifra;
if (carattere == ‘0’) cifra = 0;
if (carattere == 1) cifra = 1;
…
```

Si consiglia pertanto di scrivere anche un metodo `charToInt` che si occupi di questo compito, ossia che prenda in input un carattere e restituisca la cifra da esso rappresentata.

Soluzione

```
class GeneraLampadine
{
  public static int charToInt(char c)
  {
     if (c == '0') return 0;
     if (c == '1') return 1;
     if (c == '2') return 2;
     if (c == '3') return 3;
     if (c == '4') return 4;
     if (c == '5') return 5;
     if (c == '6') return 6;
     if (c == '7') return 7;
     if (c == '8') return 8;
     if (c == '9') return 9;
     return –1; // indica che c non è una cifra
  }
  public static int stringToInt(String s)
  {
     int n = 0;
     for (int i=0; i < s.length(); i++)
     {
        int cifra = charToInt(s.charAt(i));
        if (cifra < 0) // il carattere non è una cifra!
           break;      // esce dal ciclo for
        n = n*10 + cifra;
     }
     return n;
  }
  public static void main(String[] args)
  {
     if (args.length < 1)
        System.out.println("Errore!");
     int n = stringToInt(args[0]);
     System.out.println("Costruzione dell'array.");
     Lampadina[] tutte = new Lamapadina[n];
     System.out.println("Costruzione delle lampadine.");
     for (int i=0; i < tutte.length; i++)
        tutte[i] = new Lampadina();
  }
}
```

7. Costruttori – parte II

Breve riepilogo sui costruttori:

(1) Il costruttore di una classe è un metodo particolare che ha lo stesso nome della classe e che non può restituire alcun valore. Può però avere degli argomenti.
(2) Il costruttore di una classe viene chiamato ogniqualvolta viene creato un oggetto di quella classe con l'istruzione `new`.
(3) Una classe ha sempre almeno un costruttore. Se chi scrive la classe non mette nessun costruttore, Java inserisce automaticamente un costruttore che non prende nessun argomento e non contiene nessuna istruzione.
(4) Un costruttore che non prende argomenti è detto **costruttore di default.**
(5) Una classe può avere più di un costruttore, a patto che costruttori diversi prendano argomenti diversi. Ad esempio la classe `String` ha un costruttore di default, che viene chiamato quando viene eseguita l'istruzione `new String()`, ed ha anche un costruttore che prende come argomento un'altra stringa, che viene chiamato quando viene eseguita un'istruzione tipo `new String("Bla")`.
(6) Se nella classe c'è almeno un costruttore, Java non inserisce nessun costruttore automatico.
(7) All'interno di un costruttore è possibile richiamarne un altro usando la parola chiave **this**. Ad esempio, il costruttore di default di `String` potrebbe essere fatto semplicemente così:

```
public String()
{
  this("");
}
```

Esso rihiama il secondo costruttore (che prende come argomento una stringa) e gli passa una stringa vuota.

8. Costruttori e Sottoclassi

Si considerino le seguenti due classi:

```
class Animale
{
  Animale()
  {
     System.out.println("Costruttore di Animale");
  }
}
class Topo extends Animale
{
  Topo()
  {
     System.out.println("Costruttore di Topo");
  }
}
```

Ora si provi a scrivere un programma che si limita a creare un oggetto di tipo `Topo`.

Quesiti:

Q1. Si spieghi l'output del programma.
Q2. Si modifichi la classe `Animale` in modo che essa abbia un membro dati privato di tipo `String` che si chiama `nome` (deve infatti rappresentare il nome dell'animale). Si modifichi inoltre il costruttore in modo che prenda come argomento una stringa da assegnare a `nome`. Sono necessarie altre modifiche perché il programma funzioni? Se sì, quali?

9. Ereditarietà I – Overriding

L'**overriding** consiste nel **ridefinire** un metodo esistente in una classe C all'interno di una sottoclasse di C. Gli oggetti della sottoclasse useranno la versione ridefinita del metodo, mentre gli oggetti della classe base useranno la versione iniziale.
Modifichiamo la classe `Animale` nel modo seguente:

```
class Animale
{
  private String nome;
  public Animale(String s)
  {
     nome = s;
  }
  public String comeTiChiami()
  {
     return nome;
  }
  public void parla()
  {
  }
}
```

Il metodo `parla` non fa niente perché non è possibile sapere che suono emetta un generico animale.

Quesiti:

Q1. Si costruiscano due classi `Topo` e `Gatto` che estendono la classe `Animale`. In particolare le due classi devono **ridefinire** il metodo `parla`. **Attenzione:** per ridefinire un metodo è necessario che la sua dichiarazione sia **identica** a quella della versione base (in questo caso `"public void parla()"`). Il `Topo` deve ridefinire il metodo in modo che esso stampi sullo schermo "Squit" mentre il `Gatto` deve stampare sullo schermo "Miao".

Q2. Si scriva il seguente programma e se ne osservino i risultati:

```
class AnimaliParlanti
{
  public static void main(String[] args)
  {
     if (args.length == 0)
     {
        System.out.println("Errore");
        System.exit(1);
     }
     Animale a = null;
     if (args[0].equals("Topo"))
        a = new Topo("Jerry");
     else if (args[0].equals("Gatto"))
        a = new Gatto("Tom");
     if (a != null)
        a.parla();
  }
}
```

Q3. Dal momento che la versione base del metodo `parla` (quella definita nella classe `Animale`) non fa nulla, si sarebbe potuto pensare di ometterla. Tuttavia in questo modo il programma non avrebbe funzionato. Che tipo di errore avrebbe dato? Perché?

Q4. Cosa sarebbe cambiato nel programma principale se invece di scrivere `(args[0].equals("Topo"))` avessimo scritto `(args[0] == "Topo")`?

10. Ereditarietà II – Overloading

L'**overloading** consiste nel dare una **nuova versione** di un metodo che ha lo stesso nome ma prende argomenti diversi. La nuova versione non sovrascrive la versione precedente, ma si aggiunge ad essa, e viene trattata a tutti gli effetti come un nuovo metodo. La nuova versione può essere aggiunta sia nella stessa classe della versione preesistente, o in una sottoclasse.
Si aggiunga alla classe `Animale` il seguente metodo:

```
public void incontra(Animale a)
{
  System.out.println(nome + ":<Ciao, " + a.nome + ">");
  parla();
}
```

Il metodo `incontra` rappresenta ciò che un animale dice quando ne incontra un altro. Si scriva ora il seguente programma, e se ne osservi il risultato:

```
class Incontro
{
  public static void main(String[] args)
  {
     Gatto g = new Gatto("Tom");
     Topo t = new Topo("Jerry");
     System.out.println("Il gatto incontra il topo:");
     g.incontra(t);
     System.out.println("Il topo incontra il gatto:");
     t.incontra(g);
  }
}
```

Quesiti:

Q1. Si aggiunga ora nella classe `Topo` una nuova versione del metodo `incontra` che prende come argomento un oggetto di tipo `Gatto`. Questo metodo rappresenta ciò che succede quando un topo incontra un gatto, e potrebbe ad esempio essere fatto così:

```
public void incontra(Gatto g)
{
  System.out.println(nome + ":<Aiutoooooo!!!>");
  parla();
}
```

Analogamente si aggiunga alla classe `Gatto` una terza versione di `incontra` che prende come argomento un oggetto di tipo `Topo`, e descrive ciò che succede quando un gatto incontra un topo. Si osservi come cambia il funzionamento del programma precedente.

Q2. Si modifichi il `main` descritto sopra nel modo seguente:

```
public static void main(String[] args)
{
    Animale g = new Gatto("Tom");
    Animale t = new Topo("Jerry");
    System.out.println("Il gatto incontra il topo:");
    g.incontra(t);
    System.out.println("Il topo incontra il gatto:");
    t.incontra(g);
}
```

Si osservi l'output del programma e si spieghi cosa è successo.

11. Interfaccia grafica

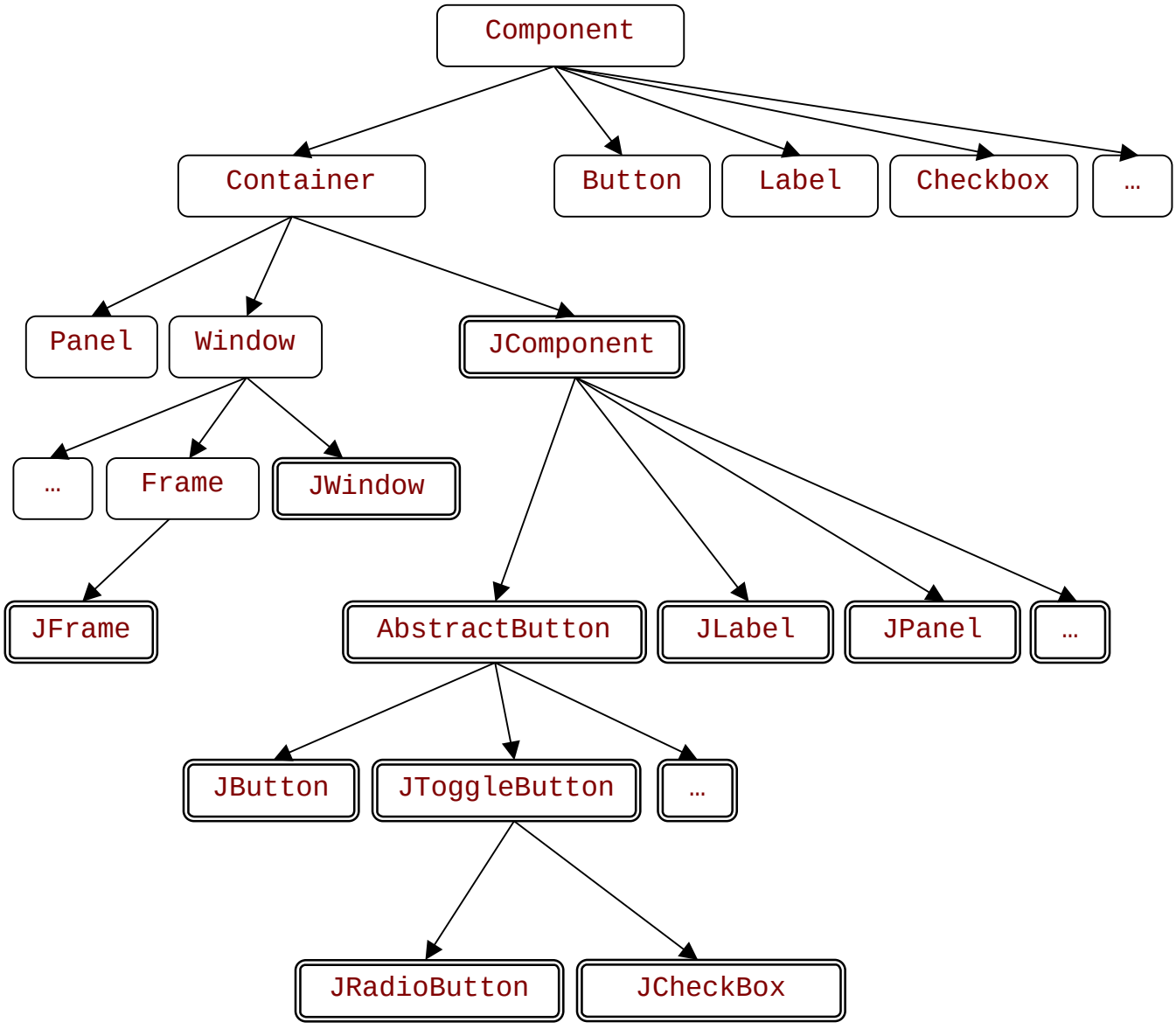
Ecco una parte della gerarchia delle classi predefinite in Java che costituiscono i componenti principali dell'interfaccia grafica.
Queste classi sono definite in libreria, altrimenti dette **package**. Per poterle usare in un file è necessario includere il package. Questo deve essere fatto all'inizio del file. L'istruzione per includere un package è:

```
include nomepackage.nomeclassedaincludere;
```

oppure

```
include nomepackage.*;
```

se si vogliono includere tutte le classi del package.
Per quanto riguarda il disegno, le classi con il bordo semplice sono contenute nel package `java.awt` mentre le classi con il bordo doppio sono contenute nel package `javax.swing`



12. Visualizzare una finestra `JFrame`.

La classe `JFrame` descrive una finestra di quelle che vengono normalmente usate come **finestre principali** delle applicazioni. Si tratta cioè di una finestra che ha un **bordo** che permette di ridimensionarla, e una **barra del titolo** con i pulsanti per chiuderla, minimizzarla e ripristinarla.
Tramite la classe `JFrame` possiamo **creare** una finestra e **visualizzarla** sullo schermo. Vediamo come.

```
import javax.swing.*;

class ProgrammaX1
{
  public static void main(String[] args)
  {
     JFrame mainFrame = new JFrame("La mia finestra");
     mainFrame.setSize(300,300);
     mainFrame.show();
  }
}
```

Note:

i) Innanzitutto ricordiamo che, come abbiamo detto nella sezione precedente, la classe `JFrame` è contenuta nel package `javax.swing` e dunque è necessario importare o la classe oppure tutto il package. In questo esempio abbiamo deciso per comodità di importare tutto il package con l'istruzione

```
import javax.swing.*;
```

ii) Innanzitutto dobbiamo **creare** un oggetto `JFrame` attraverso il suo costruttore. La classe `JFrame` ha molti costruttori; in particolare uno dei suoi costruttori prende come argomento una **stringa di testo** che comparirà come **titolo** della finestra. Ad ogni modo c'è anche un **costruttore di default** (senza argomenti) che costruisce una finestra senza titolo. In questo esempio abbiamo scelto di usare il costruttore che permette di specificare il titolo:

```
JFrame mainFrame = new JFrame("La mia finestra");
```

iii) Creare una finestra non è sufficiente a vederla sullo schermo; infatti quando una finestra viene creata inizialmente è **invisibile**. Per visualizzarla è necessario chiamare un metodo specifico che tutti gli oggetti di tipo `Window` (e quindi anche gli oggetti di tipo `JFrame`, che è sottoclasse di `Window`) possiedono: il metodo `show`. Prima di fare ciò però vogliamo assegnare una dimensione alla finestra. Possiamo farlo con il metodo `setSize` che prende come argomenti la **larghezza** e l'**altezza** che la finestra dovrà assumere:

```
mainFrame.setSize(300,300);
```

Infine, chiamiamo il metodo `show` per visualizzare la finestra sullo schermo:

```
mainFrame.show();
```

iv) Osserviamo che quando "chiudiamo" la finestra con il pulsante apposito, essa scompare dallo schermo; ciononostante il programma sembra stare ancora girando, perché nella finestra da cui abbiamo lanciato il comando `java` non ci viene restituito il prompt dei comandi. Infatti il comportamento standard di un oggetto `JFrame` è il seguente: alla pressione del pulsante di chiusura la finestra viene nascosta, ma non eliminata. Pertanto il programma non termina. Dovremo essere noi – vedremo nell'esercizio successivo come – ad accorgerci che la finestra viene chiusa e a dare le istruzioni per fare terminare definitivamente il programma. Per ora possiamo "uccidere" brutalmente il programma premendo i tasti **CTRL+C**.

13. Uscire dal programma alla chiusura della finestra (solo JDK 1.3)

Gli oggetti `JFrame` hanno un metodo che permette di scegliere (all'interno di un insieme predefinito) quale operazione eseguire quando la finestra viene chiusa con l'apposito pulsante. Si tratta del metodo `setDefaultCloseOperation`. Le possibilità tra cui è possibile scegliere sono:

(1) non fare nulla
(2) nascondere la finestra (questa è l'operazione predefinita, che viene fatta se noi non scegliamo nulla)
(3) nascondere e distruggere la finestra
(4) nascondere e distruggere la finestra, ed uscire dal programma

A ciascuna di queste possibilità è associato un codice (un numero intero); il metodo `setDefaultCloseOperation` prende come argomento tale codice. Fortunatamente non abbiamo bisogno di ricordarci e nemmeno conoscere i codici associati alle diverse operazioni: essi sono infatti memorizzati in costanti predefinite.
Tali costanti sono:

1. `WindowConstants.DO_NOTHING_ON_CLOSE` per non fare nulla.
2. `WindowConstants.HIDE_ON_CLOSE` per nascondere la finestra.
3. `WindowConstants.DISPOSE_ON_CLOSE` per nascondere e distruggere la finestra.
4. `JFrame.EXIT_ON_CLOSE` per nascondere e distruggere la finestra, ed uscire dal programma.

Si noti che i nomi delle costanti sono preceduti dal nome della classe in cui sono definite.
L'ultima possibilità è quella che interessa a noi. Per ottenere l'effetto desiderato sarà dunque sufficiente aggiungere al programma precedente la seguente istruzione:

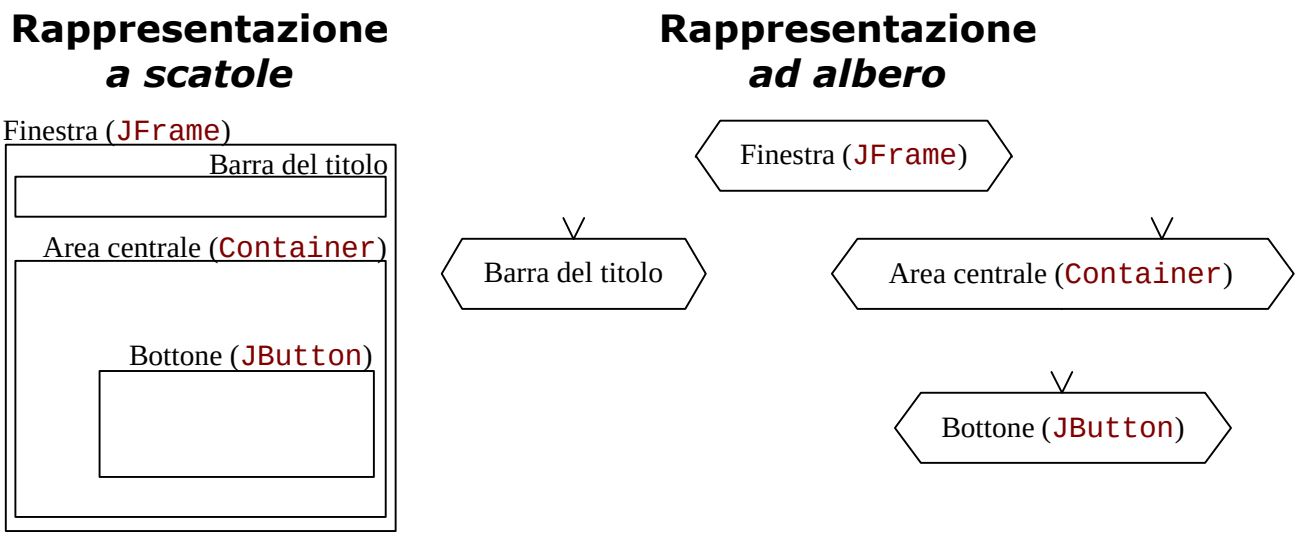
```
mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

14. Aggiungere un pulsante `JButton` ad una finestra

Vogliamo adesso aggiungere un bottone alla nostra finestra. Un bottone è un **componente grafico**; sono detti componenti grafici tutti quegli oggetti che appartengono alla classe `Component` (e sue sottoclassi).
In generale gli unici componenti grafici che sono in grado di contenerne altri sono i **contenitori**, ossia quelli che appartengono alla classe `Container` (e sue sottoclassi). La classe `Container` ha infatti il metodo `add` che permette di aggiungere all'interno del contenitore un altro componente grafico. Quando il contenitore viene visualizzato, vengono visualizzati anche tutti gli oggetti al suo interno.
Poiché un contenitore è anche un componente grafico (leggi: la classe `Container` è sottoclasse della classe `Component`), è possibile aggiungere in un contenitore un altro contenitore.
La nostra finestra `JFrame` è un contenitore; tuttavia non possiamo aggiungere componenti grafici direttamente ad essa. Infatti una finestra `JFrame` contiene già altri componenti; fra questi:

(i) la **barra del titolo**
(ii) l'**area centrale** che per il momento è vuota.

Se vogliamo aggiungere alla finestra un bottone, non dobbiamo aggiungerlo alla finestra stessa, bensì alla sua **area centrale** che è a sua volta un contenitore. Possiamo rappresentare questa situazione graficamente:



Quindi per aggiungere un bottone dobbiamo innanzitutto **ottenere dall'oggetto** `JFrame` **un riferimento alla sua area centrale** (un oggetto `Container`); dopodiché potremo **creare il bottone** e **aggiungere all'area centrale il bottone creato**.
Per fare tutte queste cose useremo un approccio leggermente diverso da quello usato nell'esercizio precedente. Poiché stiamo cercando di **personalizzare** la finestra, creeremo una classe apposita che rappresenti proprio la **nostra** finestra.
Naturalmente tale classe sarà una sottoclasse di `JFrame` in modo da ereditare il comportamento e le capacità di quest'ultima, e da permetterci di aggiungere ciò che desideriamo (in questo caso, un bottone).
Ecco il codice della nostra finestra, che chiameremo `MyFrame`:

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame
{
  public MyFrame()
  {
     super("La mia finestra");
     setSize(300,300);
     setDefaultCloseOperation(EXIT_ON_CLOSE);
     Container areaCentrale = getContentPane();
     JButton pulsante = new JButton("Pulsante");
     areaCentrale.add(bottone);
  }
}
```

Note:

a. Per cominciare, possiamo inserire le operazioni per l'aggiunta del bottone nel costruttore. Inoltre possiamo spostare nel costruttore alcune delle operazioni che prima avevamo inserito nel metodo `main`. Come prima cosa invochiamo il costruttore della superclasse (ossia `JFrame`) tramite la parola chiave `super`, e gli passiamo il titolo da dare alla finestra. In secondo luogo impostiamo le dimensioni con il metodo `setSize`. Infine impostiamo l'operazione di uscita.

```
super("La mia finestra");
setSize(300,300);
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Si osservi qui la differenza tra la chiamata di un metodo dall'**esterno** di un oggetto (bisogna far precedere il nome del metodo dal nome dell'oggetto seguito dal punto) e la chiamata all'**interno** dell'oggetto stesso, in cui non è necessario specificare il nome dell'oggetto.

b. Ora è necessario ottenere un riferimento all'area centrale della finestra `JFrame`. In Java l'area centrale dell'oggetto si chiama **content pane**; per ottenere il riferimento si può chiamare il metodo `getContentPane` definito in `JFrame`. Il metodo restituisce un riferimento ad un oggetto `Container` che memorizziamo in una variabile appropriata.

```
Container areaCentrale = getContentPane();
```

Attenzione: la classe `Container` è definita nel package `java.awt` quindi è necessario importarlo!

c. A questo punto possiamo creare un pulsante. La classe che descrive i pulsanti è la classe `JButton`.. Anche essa ha molti costruttori; quello che interessa a noi in questo momento è il costruttore che prende come argomento una stringa di testo da visualizzare sul pulsante.

```
JButton pulsante = new JButton("Pulsante");
```

d. Infine, aggiungiamo il pulsante all'area centrale attraverso il metodo `add`. Il metodo `add` prende come argomento qualunque oggetto di tipo `Component`, quindi anche un oggetto `JButton`.

```
areaCentrale.add(bottone);
```

Adesso che abbiamo definito la classe che rappresenta la nostra finestra, dobbiamo creare un oggetto di tipo `MyFrame` nel metodo `main`, e visualizzarlo sullo schermo. Il `main` sarà più corto di quello degli esercizi precedenti, perché abbiamo spostato alcune istruzioni nel costruttore di `MyFrame`.

```
class ProgrammaX2
{
  public static void main(String[] args)
  {
     MyFrame mainFrame = new MyFrame();
     mainFrame.show();
  }
}
```

Ed ecco il risultato:

15. Rilevare la pressione del pulsante

Java gestisce l'interazione tra un utente e l'interfaccia grafica del programma con un meccanismo **ad eventi**. Quando un utente utilizza un programma da noi creato, solitamente interagisce con l'interfaccia grafica con il mouse e la tastiera; tramite di essi può cliccare su pulsanti, selezionare elementi in un elenco, scrivere in una casella di testo, etc. **Ciascuna di queste interazioni viene chiamata "evento" (c' è un tipo di evento diverso per ogni tipo di interazione).** Quello che noi dobbiamo fare è semplicemente **catturare l'evento**, ossia fare in modo di accorgerci che esso è accaduto.
In Java un evento è rappresentato da un oggetto della classe `EventObject`. Tale classe ha svariate sottoclassi; ciascuna descrive un tipo specifico di evento. Il tipo di evento associato alla pressione di un pulsante è chiamato **action event**; la classe che lo descrive è `ActionEvent` (che naturalmente è una sottoclasse di `EventObject`).
Quando l'evento si verifica il pulsante che è stato premuto crea automaticamente un oggetto di tipo `ActionEvent` che contiene le informazioni sull'evento in questione (ad esempio, **quale** pulsante è stato premuto). Da un punto di vista pratico, pertanto, dobbiamo riuscire a farci "consegnare" l'oggetto evento appena costruito.
Perché la nostra classe possa ricevere l'oggetto evento generato dal pulsante essa deve:

1. essere in grado di ricevere un oggetto `ActionEvent`;
2. dichiarare esplicitamente al pulsante che vuole ricevere gli oggetti `ActionEvent` da esso generati.

Perché una classe sia in grado di ricevere degli oggetti evento essa deve definire al suo interno determinati metodi. Questi metodi sono quelli il creatore dell'evento (ad esempio il pulsante) chiamerà per "passare" l'oggetto evento a chi desidera riceverlo (ad esempio la nostra finestra).
I metodi da definire cambiano a seconda del tipo di evento; l'elenco dei metodi per un certo tipo di evento si trova nella corrispondente **interfaccia**. Ad esempio i metodi necessari per l'evento di tipo `ActionEvent` si trovano nell'interfaccia `ActionListener`. Si tratta in realtà di un metodo solo:

```
public void actionPerformed(ActionEvent e)
```

Dunque la nostra classe deve definire questo metodo. Quando il pulsante viene premuto esso innanzitutto crea un oggetto di tipo `ActionEvent`; dopodiché **chiama** il metodo `actionPerformed` che noi abbiamo definito e gli passa come argomento l'oggetto che ha appena costruito. Pertanto le istruzioni che mettiamo nel metodo costituiscono la nostra **risposta** alla pressione del bottone.
Definire il metodo non è sufficiente: dobbiamo dichiarare **nell'intestazione della nostra classe** che vogliamo implementare l'interfaccia corrispondente (ossia in questo caso `ActionListener`). Questo si fa con la seguente dichiarazione:

```
public class MyFrame extends JFrame implements ActionListener
```

Questo serve per **garantire** che nella nostra classe ci sia il metodo `actionPerformed` (altrimenti noi potremmo decidere che vogliamo ricevere l'evento dal pulsante e poi quando il pulsante cerca di chiamare il metodo non lo trova!). Perché il fatto di scrivere `implements ActionListener` è una garanzia? Perché se noi dichiariamo nell'intestazione di una classe che vogliamo **implementare un'interfaccia** il compilatore ci compilerà il file **soltanto se nella classe sono definiti TUTTI i metodi elencati nell'interfaccia**.
A questo punto abbiamo soddisfatto il punto 1, ossia abbiamo messo la nostra classe in grado di ricevere un oggetto `ActionEvent` (implementa il metodo che serve **e** dichiara ufficialmente che lo fa). Resta da comunicare al pulsante che la classe vuole ricevere i suoi eventi. Questo si fa chiamando un metodo del pulsante (classe `JButton`) che è in grado di "iscrivere" un oggetto nella lista di coloro che ricevono l'evento. Il metodo si chiama `addActionListener` ed accetta soltanto oggetti che implementano l'interfaccia `ActionListener`:

```
public void addActionListener(ActionListener l)
```

Attenzione: questo metodo non dobbiamo definirlo noi, è già definito nella classe `JButton`. Noi dobbiamo solo chiamarlo e passargli come argomento il nostro `ActionListener`, ossia la finestra stessa.
Vediamo ora come esempio un programma che "riceve" l'evento corrispondente alla pressione del bottone, e quando lo riceve stampa a video la scritta "Click!".

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyFrame extends JFrame implements ActionListener
{
  public MyFrame()
  {
     super("La mia finestra");
     JButton pulsante = new JButton("Pulsante");
     Container areaCentrale = getContentPane();
     areaCentrale.add(pulsante);
     pulsante.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e)
  {
     System.out.println("Click!");
  }
}
```

Note:

a. Le parti sottolineate sono quelle aggiunte rispetto alla versione precedente. Si osservi innanzitutto che abbiamo importato un nuovo **package**: si tratta di **java.awt.event** che contiene tutte le classi e interfacce che hanno a che vedere con gli eventi (in particolare, a noi servono `ActionEvent` e `ActionListener`).

b. Il metodo `actionPerformed` contiene una sola istruzione per stampare sul video una scritta. Durante l'esecuzione, cliccando sul pulsante, si può verificare che il metodo viene chiamato **ogni volta** che il pulsante viene premuto.

c. Si osservi l'istruzione:

```
pulsante.addActionListener(this)
```

Essa chiama il metodo `addActionListener` dell'oggetto `pulsante`, passandogli come parametro `this` ossia l'oggetto corrente. Questo perché è l'oggetto `MyFrame` stesso che stiamo definendo che riceverà l'evento creato dal pulsante.

Quesiti

Q1. Nell' esercizio precedente abbiamo fatto sì che fosse la finestra stessa a ricevere l'evento della pressione del pulsante. Si provi ora a costruire una **classe separata** `AscoltaPulsante` che implementa `ActionListener`; quindi si crei un oggetto di tale classe che riceve l'evento in questione. Si ricordi di importare nella nuova classe i package necessari!
Perché la classe `AscoltaPulsante` implementi `ActionListener` essa deve avere la dichiarazione **implements** e inoltre definire il metodo `actionPerformed`. D'altro canto non sarà più necessario che `MyFrame` implementi `ActionListener`, pertanto da essa si possono rimuovere la dichiarazione **implements** e il metodo `actionPerformed`.
Nel costruttore di `MyFrame` si dovrà innanzitutto costruire un oggetto di tipo `AscoltaPulsante` (sarà lui a dover ricevere l'evento):

```
AscoltaPulsante ap = new AscoltaPulsante();
```

Quindi si dovrà registrare presso il pulsante non più la finestra stessa (`this`), ma l'oggetto appena creato:

```
pulsante.addActionListener(ap);
```

Q2. Si modifichi la classe `AscoltaPulsante` in modo che invece di stampare la scritta "Click!" stampi **il numero di volte che il pulsante è stato premuto sino a quel momento**.

16. Uscire dal programma alla chiusura della finestra (JDK 1.1 e 1.2)

Nell'esercizio precedente abbiamo visto come ricevere eventi di tipo `ActionEvent`, che vengono creati da un pulsante quando viene premuto. In questo esercizio vedremo invece come ricevere gli eventi di tipo `WindowEvent`, che vengono creati da una finestra quando le accade qualcosa. In particolare vogliamo accorgerci di quando la finestra viene chiusa utilizzando il pulsante con la X in alto a destra, in modo da poter uscire dal programma.
Mentre c'è un solo evento di tipo `ActionEvent` (la pressione del pulsante: è l'unico caso in cui viene creato un oggetto `ActionEvent`), ci sono molti eventi di tipo `WindowEvent`, e sono tutti quegli eventi che hanno a che vedere con una finestra. Ogni volta che uno di questi eventi accade la finestra crea un oggetto di tipo `WindowEvent`; quindi chiama **un metodo diverso** a seconda dell'evento che si è verificato. Dunque l'interfaccia corrispondente all'evento della finestra, ossia `WindowListener`, contiene una lista di più metodi, uno per ogni diversa circostanza in cui l'evento `WindowEvent` viene creato.
Qui di seguito è riportato l'elenco dei metodi, e per ciascuno la circostanza in cui viene chiamato:

- `public void windowOpened(WindowEvent we)`: finestra aperta per la prima volta.
- `public void windowDeactivated(WindowEvent we)`: finestra disattivata (l'utente passa ad un'altra finestra).
- `public void windowActivated(WindowEvent we)`: finestra riattivata.
- `public void windowIconified(WindowEvent we)`: finestra minimizzata.
- `public void windowDeiconified(WindowEvent we)`: finestra ripristinata (dopo essere stata minimizzata).
- `public void windowClosing(WindowEvent we)`: l'utente manifesta l'intenzione di chiudere la finestra. **Questa è la circostanza particolare che ci interessa**.
- `public void windowClosed(WindowEvent we)`: finestra effettivamente chiusa.

Creiamo ora una classe `AscoltaFinestra` analoga alla precedente `AscoltaPulsante`, ma che implementa `WindowListener`:

```
import java.awt.event.*;

class AscoltaFinestra implements WindowListener
{
  public void windowOpened(WindowEvent we)
  {
  }
  public void windowDeactivated(WindowEvent we)
  {
  }
  public void windowActivated(WindowEvent we)
  {
  }
  public void windowIconified(WindowEvent we)
  {
  }
  public void windowDeiconified(WindowEvent we)
  {
  }
  public void windowClosing(WindowEvent we)
  {
     System.exit(1);
  }
  public void windowClosed(WindowEvent we)
  {
  }
}
```

Note

a. Come si può osservare, la classe `AscoltaFinestra` definisce tutti i metodi elencati in `WindowListener`: questo è **obbligatorio** se si dichiara di implementare l'interfaccia `WindowListener`. A noi interessa solo compiere un'azione (in particolare, uscire dal programma) quando l'utente dichiara di voler chiudere la finestra. Poiché in tale circostanza viene chiamato il metodo `windowClosing`, mettiamo l'istruzione per uscire dal programma dentro di esso. Per quanto riguarda gli altri metodi, essi vengono chiamati in circostanze in cui non ci interessa fare niente. Pertanto li lasciamo senza istruzioni. (Si osservi che c'è una differenza fra un metodo **definito esplicitamente senza istruzioni**, come in questo caso, e un metodo **dichiarato ma non definito** come succede nelle interfacce. Nel primo caso esiste un'area in cui vengono conservate le istruzioni del metodo, solo che è vuota. Nel secondo caso tale area non esiste.)

Per fare sì che alla chiusura della finestra il programma esca, ci resta solo da **registrare presso la finestra** il fatto che c'è un oggetto (di tipo `AscoltaFinestra`) interessato a ricevere i suoi eventi. Riprendiamo pertanto la nostra finestra `MyFrame` e **aggiungiamo** le nuove istruzioni (lasciando anche quelle per ascoltare il pulsante!).

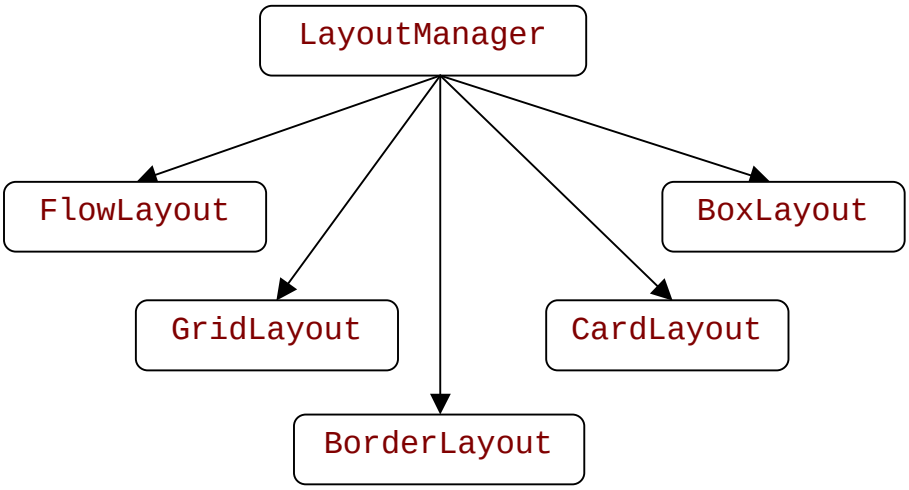
```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame
{
  public MyFrame()
  {
     super("La mia finestra");
     JButton pulsante = new JButton("Pulsante");
     Container areaCentrale = getContentPane();
     areaCentrale.add(pulsante);
     AscoltaPulsante ap = new AscoltaPulsante();
     pulsante.addActionListener(ap);
     AscoltaFinestra af = new AscoltaFinestra();
     addWindowListener(af);
  }
}
```

Note

b. Le istruzioni aggiunte sono quelle riportate sottolineate. Osserviamo che la chiamata al metodo `addWindowListener` non è preceduta dal nome di alcun oggetto: questo perché il metodo viene chiamato sulla finestra stessa (`this`).

17. Posizionamento dei componenti: i Layout Manager

Supponiamo di aggiungere più bottoni ad una stessa finestra: in tal caso vorremmo anche decidere come posizionarli uno rispetto all'altro.
La decisione su come disporre i componenti grafici in una finestra viene presa da un oggetto chiamato **Layout Manager**. Ci sono diversi tipi di Layout Manager: ciascuno dispone i componenti in un modo diverso. Ciascun tipo di Layout Manager è descritto da una classe diversa; tutte però derivano da `LayoutManager`. Nel seguente schema sono riportati i principali Layout Manager:



Vediamo adesso un esempio di come si può scegliere un particolare Layout Manager. Modifichiamo il programma dell'esercizio 16 in modo da inserire non uno, ma sei bottoni. Quindi scegliamo un Layout Manager e vediamo come dispone i componenti nella finestra.

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame
{
  public MyFrame()
  {
     super("La mia finestra");
     setSize(200,200);
     JButton b1 = new JButton("1");
     JButton b2 = new JButton("2");
     …
     JButton b6 = new JButton("6");
     Container areaCentrale = getContentPane();
     areaCentrale.setLayout(new FlowLayout());
     areaCentrale.add(b1);
     areaCentrale.add(b2);
     …
     areaCentrale.add(b6);
     AscoltaFinestra af = new AscoltaFinestra();
     addWindowListener(af);
  }
}
```

Note

a. L'istruzione chiave è quella sottolineata. Innanzitutto essa corrisponde in realtà a due istruzioni, in quanto è una forma abbreviata per

```
LayoutManager lm = new FlowLayout();
areaCentrale.setLayout(lm);
```

Con l'istruzione `new FlowLayout()` si crea un nuovo oggetto Layout Manager del tipo desiderato, in questo caso appunto `FlowLayout`. Bisogna poi assegnare il Layout Manager alla finestra che dovrà utilizzarlo, ossia nell'esempio `areaCentrale`. Questo viene fatto chiamando il metodo `setLayout` a cui si passa il Layout Manager appena creato.
Il costruttore di `FlowLayout` non prende nessun argomento. Vedremo che i costruttori di alcuni Layout Manager prendono degli argomenti.

Caratteristiche del Flow Layout

- Il costruttore è senza argomenti;
- i componenti vengono disposti in fila, da sinistra a destra;
- se non ci stanno in una riga, vengono disposti su due, e così via;
- i componenti su ciascuna riga sono centrati rispetto alla finestra;
- ciascun componente è della dimensione minima necessaria per visualizzarlo (nel caso di un bottone, tale dimensione minima dipende dalla scritta che esso contiene).

Vediamo ora le caratteristiche degli altri Layout Manager. Per ciascuno diamo anche un esempio di utilizzo.

Caratteristiche del Grid Layout

- Il `GridLayout` crea una griglia le cui dimensioni (righe ? colonne) devono essere specificate nel costruttore;
- viene inserito un componente in ciascun riquadro della griglia; se ci sono più componenti che riquadri (ad esempio 7 componenti in una griglia 2?3) vengono aggiunte delle colonne. Se ci sono meno componenti che riquadri, il numero di colonne viene aggiustato in modo che non ci siano colonne completamente vuote;
- i riquadri hanno tutti la stessa dimensione;
- i componenti vengono allargati ad occupare l'intero riquadro.

Esempio:

```
areaCentrale.setLayout(new GridLayout(3,2));
```

Caratteristiche del Border Layout

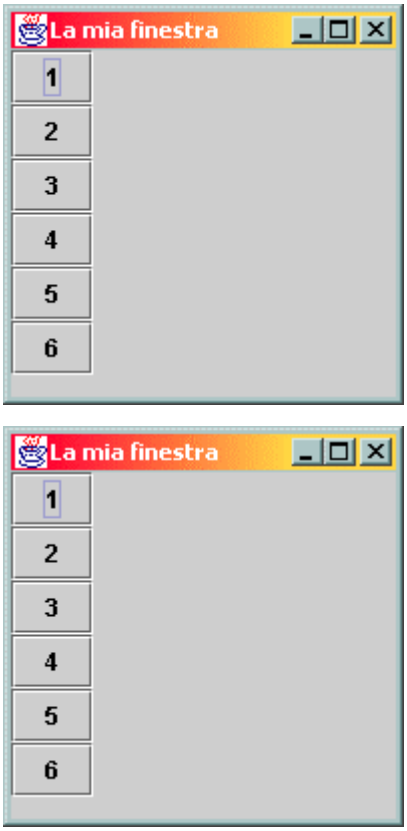
- Il `BorderLayout` divide la finestra in 5 parti, chiamate NORTH, SOUTH, WEST, EAST e CENTER;
- in ciascuna parte viene inserito un componente; poiché ci sono 5 sezioni possono essere inseriti solo cinque componenti;
- le aree NORTH e SOUTH hanno larghezza massima e altezza minima, mentre le aree WEST ed EAST hanno altezza massima e larghezza minima. L'area CENTER occupa lo spazio restante;
- se una delle aree ai bordi è vuota il suo spazio viene occupato dall'area CENTER;
- quando si aggiunge un componente bisogna anche specificare in **quale** area lo si vuole inserire.

Esempio:

```
areaCentrale.setLayout(new BorderLayout());
areaCentrale.add(b1, BorderLayout.NORTH);
areaCentrale.add(b2, BorderLayout.SOUTH);
…
```

Si noti che le costanti numeriche NORTH, SOUTH, etc. sono definite nella classe `BorderLayout` e pertanto è necessario far precedere i loro nomi dal nome della classe stessa.

Caratteristiche del Box Layout



- Il Box Layout dispone i componenti in verticale **oppure** in orizzontale, a seconda di quanto specificato nel costruttore attraverso le costanti Y_AXIS (verticale) e X_AXIS (orizzontale);
- il costruttore vuole anche come parametro un riferimento alla finestra di cui sarà il Layout Manager;
- i componenti vengono disposti in colonna (o in riga) anche qualora non stessero nella finestra;
- la dimensione dei componenti è la minima possibile.

Esempio:

```
areaCentrale.setLayout(new BoxLayout(areaCentrale, BoxLayout.Y_AXIS));
```

Le costanti X_AXIS e Y_AXIS sono definite nella classe `BoxLayout`.

Quesito

Si provino ad usare i Layout Manager illustrati in questo esercizio e si verifichi il risultato.

18. Posizionamento dei componenti: i Componenti Invisibili

I componenti invisibili sono dei componenti che occupano spazio ma non si vedono, pertanto possono essere utilizzati per introdurre degli spazi tra altri componenti. Ci sono due tipi di componenti invisibili: Area Rigida e Colla. Il Layout Manager che permette di sfruttare meglio i componenti invisibili è il Box Layout; ad ogni modo essi possono essere inseriti anche con gli altri Layout Manager, sebbene i risultati non siano sempre quelli più ovvii.

Area Rigida (`RigidArea`)

Un'Area Rigida è un rettangolo di dimensione fissata che occupa lo spazio corrispondente. Per creare un'Area Rigida è necessario chiamare il metodo statico `createRigidArea` della classe `Box`. I metodi che servono per creare degli oggetti che non possono essere creati direttamente tramite il costruttore sono detti

metodi *factory* ossia **metodi-fabbrica**. Il metodo `createRigidArea` è per l'appunto un metodo-fabbrica. esso prende come argomento un oggetto di tipo `Dimension`. La classe `Dimension` è una classe molto semplice che si limita a raggruppare in un unico oggetto due interi rappresentati rispettivamente una larghezza ed una altezza.
Ecco l'istruzione necessaria per creare, ad esempio, un'Area Rigida di dimensioni 20?20:

```
Component areaRigida = Box.createRigidArea(new Dimension(20,20));
```

Ovviamente perché l'Area Rigida prenda il suo posto nella finestra è necessario inserirvela come qualunque altro componente. È necessario ricordare che i componenti vengono visualizzati nella finestra nell'ordine in cui vengono inseriti, pertanto ad esempio per inserire un'Area Rigida fra due bottoni è necessario inserire nell'ordine il primo bottone, l'Area Rigida e quindi il secondo bottone. Un'Area Rigida può anche avere una delle due dimensioni pari a 0; in tal caso più che di un'area si tratterà di un segmento, e occuperà spazio solo in una dimensione.

Esempio con Box Layout orizzontale, due bottoni separati da un'Area Rigida 20? 0.

```
 … vengono creati due pulsanti, p1 e p2 …
areaCentrale.setLayout(new BoxLayout(areaCentrale,BoxLayout.X_AXIS));
areaCentrale.add(p1);
areaCentrale.add(Box.createRigidArea(new Dimension(20,0)));
areaCentrale.add(p2);
 … etc etc …
```

È importante tenere presente che la dimensione impostata per l'Area Rigida al momento della costruzione è una **dimensione minima**; pertanto quei Layout Manager che massimizzano la dimensione di un componente non ne tengono conto (si provi ad esempio ad inserire un'Area Rigida in un Grid Layout: essa occuperà un posto, ma la dimensione sarà comunque quella del riquadro della griglia).

Colla (`Glue`)

Il secondo tipo di componente invisibile è la Colla. La Colla non ha una dimensione fissa, ma tende ad espandersi il più possibile (se il Layout Manager glielo consente). Potremmo dire che la Colla ha dimensione minima nulla, e dimensione massima infinita. Essa viene usata per distanziare due componenti il più possibile. La Colla può essere orizzontale o verticale. Una Colla orizzontale ha altezza 0, mentre la sua larghezza è espandibile. Una Colla verticale ha larghezza 0, ed è la sua altezza ad essere espandibile. I due tipi di Colla si possono creare con due metodi-fabbrica della classe `Box`, rispettivamente `createHorizontalGlue` e `createVerticalGlue`:

```
Component collaOrizzontale = Box.createHorizontalGlue();
Component collaVerticale = Box.createVerticalGlue();
```

Esempio con Box Layout orizzontale, due bottoni separati da una Colla orizzontale.

```
 … vengono creati due pulsanti, p1 e p2 …
areaCentrale.setLayout(new
BoxLayout(areaCentrale,BoxLayout.X_AXIS));
areaCentrale.add(p1);
areaCentrale.add(Box.createHorizontalGlue());
areaCentrale.add(p2);
 … etc etc …
```

Poiché la dimensione **minima** della Colla è nulla, con alcuni Layout Manager essa non funziona. Ad esempio il Flow Layout visualizza tutti i componenti alla loro dimensione minima, pertanto in esso la Colla non si vede. Se ci sono più oggetti Colla le loro dimensioni vengono i l più possibile uguagliate.

Esempio con Box Layout orizzontale, tre pulsanti separati da Colle orizzontali.

```
 … vengono creati tre pulsanti, p1, p2  e p3 …
areaCentrale.setLayout(new BoxLayout(areaCentrale,BoxLayout.X_AXIS));
areaCentrale.add(p1);
areaCentrale.add(Box.createHorizontalGlue());
areaCentrale.add(p2);
areaCentrale.add(Box.createHorizontalGlue());
areaCentrale.add(p3);
 … etc etc …
```

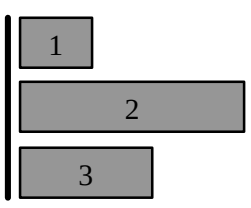
Quesiti

Q1. Si provi a costruire una finestra con i bottoni disposti come nella seguente figura (tra i bottoni 2 e 3 ci sono 10 pixel di distanza):

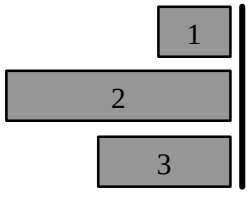
Q2. Si provi ora a costruire una finestra con i bottoni disposti come nella seguente figura:

19. Posizionamento dei componenti: allineamento in un Box Layout

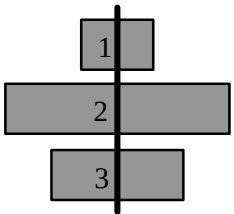
Quando si utilizza un Box Layout è possibile specificare anche l'**allineamento** dei componenti uno rispetto all'altro. L'allineamento **orizzontale** decide se un componente si trova sulla destra o sulla sinistra rispetto ad un altro, mentre l'allineamento **verticale** decide se un componente si trova in alto o in basso rispetto ad un altro. Qui sotto sono riportati alcuni esempi; come si può vedere dalle figure l'allineamento orizzontale influisce solo su componenti disposti verticalmente (Box Layout con Y_AXIS), mentre l'allineamento



orizzontale a sinistra

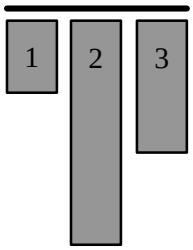


orizzontale a destra

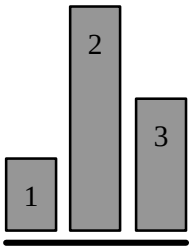


orizzontale al centro

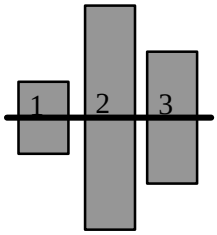
verticale influisce solo sui componenti disposti orizzontalmente (Box Layout con X_AXIS).



verticale in alto



verticale in basso

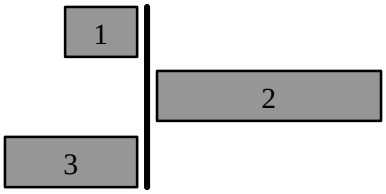


verticale al centro

L'allineamento di un componente è stabilito da due parametri decimali di tipo `float` chiamati AlignmentX (allineamento orizzontale) e AlignmentY (allineamento verticale). Per modificarne il valore si possono usare i metodi del componente `setAlignmentX` e `setAlignmentY` rispettivamente.
I parametri passati a questi metodi devono essere valori `float` compresi fra 0 e 1. In particolare:

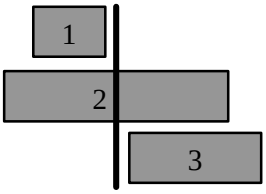
- il valore 0 significa allineamento a sinistra e in alto, rispettivamente;
- il valore 1 significa allineamento a destra e in basso, rispettivamente;
- il valore 0.5 significa allineamento centrale;
- i valori intermedi significano posizioni intermedie fra quelle elencate, ma sono molto poco usati.

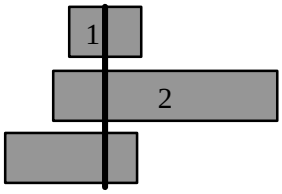
Per ottenere gli allineamenti raffigurati è necessario assegnare a tutti i componenti uno stesso valore; è tuttavia anche possibile ottenere effetti diversi impostando valori diversi per ciascun componente. Ecco alcuni esempi:



In questo caso il primo e terzo componente hanno allineamento orizzontale pari a 1 (destra), mentre il secondo componente ha allineamento pari a 0 (sinistrA). Si immagini una linea verticale, come quella in figura, che deve coincidere con i lati destri o sinistri a seconda dell'allineamento impostato.

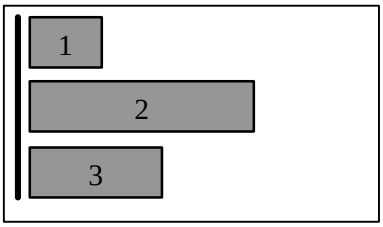
In questo caso il primo componente ha allineamento pari a 1 (destra), il secondo pari a 0.5 (centro) e il terzo pari a 0 (sinistra).

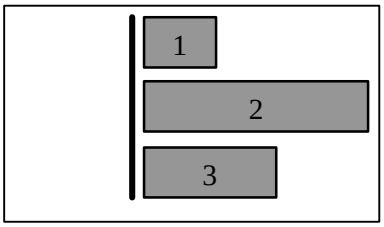




In questo caso il primo componente ha allineamento pari a 0.5 (centro), il secondo pari a 0.25 (un quarto sulla sinistra) e il terzo pari a 0.75 (un quarto sulla destra).

Quanto detto spiega come i componenti vengono disposti **uno rispetto all'altro**, ma non come vengono disposti all'interno della finestra. Ad esempio nella figura sottostante si vede come dei componenti allineati a sinistra possano essere disposti diversamente all'interno della finestra:





Ebbene: la disposizione dei componenti rispetto alla finestra viene **calcolata automaticamente** a partire dall'allineamento reciproco, e pertanto non è possibile stabilirla indipendentemente dal resto. In generale possiamo dire che:

- se i componenti sono tutti allineati a sinistra, saranno a sinistra anche nella finestra;
- se sono tutti allineati a destra, saranno a destra anche nella finestra;
- se sono tutti al centro, saranno al centro anche nella finestra;

- altrimenti saranno in qualche posizione intermedia.

Andiamo ora a sperimentare quanto detto fino a qui; modifichiamo ossia il programma degli ultimi esercizi in modo da vedere i diversi allineamenti.

```
 … siamo nel costruttore della finestra …
JButton p1 = new JButton("1");
JButton p2 = new JButton("222222"); //più lungo
JButton p3 = new JButton("333"); //lunghezza intermedia
areaCentrale.setLayout(new BoxLayout(areaCentrale,BoxLayout.Y_AXIS));
p1.setAlignmentX(1);
p2.setAlignmentX(1);
p3.setAlignmentX(1);
areaCentrale.add(p1);
areaCentrale.add(p2);
areaCentrale.add(p3);
 … etc etc …
```

Quesito

Si provi ad eseguire il programma con diversi valori per l'allineamento; in particolare si provi a disporre i componenti come nelle figure precedenti.

20. Posizionamento dei componenti: Pannelli annidati

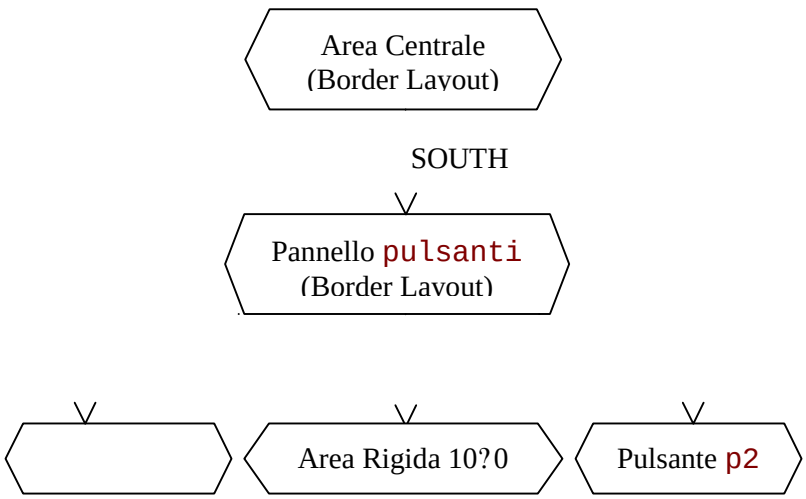
I Layout Manager che abbiamo visto permettono un numero limitato di disposizioni dei componenti. Alcuni poi (ad es. il Border Layout) permettono addirittura un numero limitato di componenti!
Ad esempio, con nessuno dei Layout Manager visti sinora è possibile ottenere la seguente finestra:

Il segreto nell'ottenere le più svariate combinazioni di pulsanti consiste nel **usare delle finestre intermedie**, dette **Pannelli**, che vengono inserite una dentro l'altra. Ciascun Pannello è un'area rettangolare all'interno della quale è possibile disporre altri componenti, utilizzando tutto ciò che abbiamo visto sinora (Layout Manager, Componenti Invisibili, Allineamento, etc). A sua volta, tuttavia, un pannello è un componente, e pertanto può essere inserito all'interno di una finestra.
Un pannello è descritto dalla classe `JPanel` e può essere costruito con `new JPanel()`.

Consideriamo ad esempio la finestra qui sopra. Essa può essere realizzata nel modo seguente:

- Si crea un Pannello (chiamiamolo `pulsanti`) in cui vengono inseriti i due pulsanti. Il Pannello usa un Flow Layout (in modo che i pulsanti stiano al centro), e tra i due pulsanti viene inserita un'Area Rigida per spearli un po'.
- Si assegna all'area centrale della finestra un Border Layout.
- Si inserisce il Pannello `pulsanti` nella zona SOUTH dell'Area Centrale.

La figura a fianco schematizza graficamente quanto detto.



Ecco come deve venire modificato il costruttore della finestra per ottenere il risultato desiderato:

```
 … vengono creati due pulsanti, p1 e p2 …
JPanel pulsanti = new JPanel();
pulsanti.setLayout(new FlowLayout());
pulsanti.add(p1);
pulsanti.add(Box.createRigidArea(new Dimension(10,0)));
pulsanti.add(p2);
areaCentrale.setLayout(new BorderLayout());
areaCentrale.add(pulsanti, BorderLayout.SOUTH);
 … etc etc …
```

Quesito

Si provi a costruire la seguente finestra:

21. Componenti: la casella di testo (`JTextField`)

La casella di testo è un componente grafico in cui l'utente può scrivere. Per sperimentarne l'utilizzo svolgiamo il seguente esercizio:
Si costruisca un programma che contiene due caselle di testo. L'utente può scrivere nella prima; quando egli preme il tasto INVIO il contenuto viene trascritto capovolto nella seconda.

Creazione e visualizzazione di una casella di testo

Per prima cosa cerchiamo di realizzare la finestra che serve per il programma, senza preoccuparci di capovolgere la scritta.
La classe che descrive una casella di testo è `JTextField`. Il costruttore prende come argomento la larghezza della casella di testo, espressa in **numero di caratteri** che la casella deve poter contenere. **Attenzione**: tale numero non limita il numero di caratteri che l'utente può scrivere nella casella, ma solo quelli visibili simultaneamente sullo schermo. Se l'utente inserisce più caratteri, il testo scorre in modo da permettere l'inserimento dei nuovi.
Per il nostro programma creiamo due caselle di testo da 15 caratteri ciascuna:

```
JTextField prima = new JTextField(15);
JTextField seconda = new JTextField(15);
```

La finestra risultante dovrà essere la seguente:

Possiamo realizzarla assegnando all'area centrale un Flow Layout, quindi inserendo (nell'ordine) la prima casella di testo, un'Area Rigida 15?0, la seconda casella di testo.
Trucco: per ottenere che la finestra principale (ossia quella derivata da `JFrame`) abbia le dimensioni giuste, invece di impostare la dimensione manualmente con il metodo `setSize`, si può utilizzare il metodo `pack()` che sceglie le dimensioni adatte per visualizzare il contenuto della finestra. Perché il metodo funzioni correttamente, deve essere chiamato **dopo aver già aggiunto tutti i componenti alla finestra**, ossia ad esempio al termine del costruttore, o nel metodo `main` dopo aver costruito la finestra.
Attenzione: una casella di testo può essere aggiunta ad una finestra **esattamente come si fa con i pulsanti**, ossia con il metodo `add`.

Eventi generati da una casella di testo

Una casella di testo genera un evento di tipo `ActionEvent` quando l'utente preme il tasto INVIO dopo avere scritto nella casella. Pertanto se vogliamo capovolgere la scritta quando l'evento occorre dobbiamo fare in modo di riceverlo.
Come già visto per i pulsanti, per ricevere un evento di tipo `ActionEvent` bisogna disporre di un oggetto di tipo `ActionListener`. Questo oggetto può essere o la finestra stessa, o un oggetto di una nuova classe creata per l'occasione. Ricapitoliamo vantaggi e svantaggi dei due approcci:

- Utilizzare la finestra stessa è comodo perché non è necessario definire una nuova classe né creare nuovi oggetti. Tuttavia poiché la classe per la finestra estende la classe `JFrame`, se usiamo la finestra siamo costretti ad implementare l'interfaccia da zero, senza poter usare le classi `Adapter`, che abbiamo visto per i `WindowEvent`. Queste classi sono utili quando l'interfaccia contiene più metodi e non si desidera implementarli tutti, ma trattandosi di classi devono essere **estese** (mentre le interfacce vengono **implementate**) Poiché una sottoclasse può estendere una e una sola classe, se la finestra estende `JFrame` non può estendere la classe `Adapter`.
- Utilizzare una nuova classe permette di estendere una classe `Adapter`, ma comporta per l'appunto il definire una classe aggiuntiva e creare un oggetto separato che riceva l'evento.

In questo caso possiamo osservare che non c'è bisogno di utilizzare classi `Adapter`, in quanto l'interfaccia che ci interessa, `ActionListener`, ha un solo metodo e dunque un `Adapter` non ci porterebbe nessun vantaggio. Pertanto possiamo usare la finestra stessa.
Rivediamo brevemente quali sono i passi necessari per ricevere un `ActionEvent`:

1. Dichiarare che si desidera implementare l'interfaccia `ActionListener` attraverso la dichiarazione `implements`. Questa dichiarazione deve essere fatta dalla classe che in seguito verrà usata per ricevere l'evento. In questo caso si tratta della finestra.
2. Implementare effettivamente l'interfaccia `ActionListener` aggiungendo alla classe interessata il metodo `actionPerformed`, nel quale devono essere inserite le istruzioni che si vogliono eseguire quando l'evento avviene.
3. Costruire se necessario un oggetto della classe che implementa l'interfaccia (nel nostro caso non è necessario perché l'oggetto esiste già: è `this`). Chiamare il metodo `addActionListener` dell'oggetto che genera l'evento (ossia la casella di testo) passandogli come argomento l'`ActionListener` che abbiamo creato.

Ecco il codice provvisorio della finestra :

```
public class MyFrame extends JFrame implements ActionListener
{
  public MyFrame()
  {
     super("La mia finestra");
     addWindowListener(new MyWindowListener());
     setSize(400,60);
     Container areaCentrale = getContentPane();
     areaCentrale.setLayout(new FlowLayout());
     JTextField prima = new JTextField(15);
     JTextField seconda = new JTextField(15);
     text1.addActionListener(this);
     areaCentrale.add(text1);
     areaCentrale.add(Box.createRigidArea(new Dimension(15,0)));
     areaCentrale.add(text2);
  }

  public void actionPerformed(ActionEvent e)
  {
     //qui bisogna mettere le istruzioni per capovolgere
     //la scritta!
  }
}
```

Capovolgere la scritta

Il codice qui sopra è provvisorio perché come si vede nel metodo `actionPerformed` non c'è nessuna istruzione: è necessario aggiungere le istruzioni per capovolgere la scritta.
Per fare questo dobbiamo:

1. Ottenere una stringa con il contenuto della prima casella di testo.
2. Capovolgere la stringa.
3. Scrivere nella seconda casella di testo la stringa capovolta.

Se assumiamo di avere scritto un metodo `capovolgi` che prende una stringa e la restituisce capovolta, il codice di `actionPerformed` potrebbe risultare come segue:

```
public void actionPerformed(ActionEvent e)
{
  String iniziale = prima.getText();
  String finale = capovolgi(iniziale);
  seconda.setText(finale);
}
```

Come si vede la classe `JTextField` ha due metodi che permettono rispettivamente di ottenere la stringa scritta nella casella e di stampare una stringa nella casella: si tratta si `getText()` e `setText(String s)`. Sfortunatamente il codice che abbiamo scritto ha un problema: le variabili `prima` e `seconda` sono state **dichiarate** all'interno del costruttore, e sono pertanto visibili solo all'interno di esso. Per fare sì che

`prima` e `seconda` siano visibili in tutti i metodi della classe `MyFrame` esse devono essere **membri dati** della classe. Pertanto dobbiamo spostare la **dichiarazione** delle due variabili al di fuori del costruttore:

```
public class MyFrame extends JFrame implements ActionListener
{
  JTextField prima;
  JTextField seconda;
  public MyFrame()
  {
     …
```

Quesito

Si definisca nella classe `MyFrame` il metodo:

```
public String capovolgi(String s)
```

che capovolge la stringa `s`.

Ultimi ritocchi

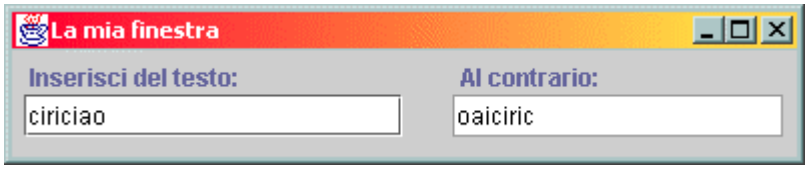
C'è un ultimo aspetto da perfezionare: per come è adesso il programma, l'utente può scrivere in entrambe le caselle di testo, mentre avrebbe senso che scrivesse solo nella seconda. La casella di testo ha un parametro booleano, `editable` (tradotto: modificabile), che specifica se la casella è modificabile o meno. Quando la casella viene costruita il parametro ha valore `true` (dunque la casella è modificabile); per cambiarne il valore si può usare il metodo `setEditable`, come ad esempio in:

```
seconda.setEditable(false);
```

Si faccia attenzione al fatto che se si desidera che la casella di testo sia non-modificabile sin dall'inizio, tale istruzione deve essere chiamata subito dopo la creazione della casella.

22. Componenti: l'etichetta (`JLabel`)

I componenti che abbiamo visto sinora (il pulsante e la casella di testo) sono **attivi**, ossia interagiscono con l'utente. Ci sono anche componenti **passivi**, che non interagiscono con l'utente ma servono semplicemente a visualizzare delle informazioni. Un esempio è la cosiddetta **etichetta**, che visualizza sullo schermo una scritta (ad esempio a scopo esplicativo). Un esempio estremo di componenti passivi sono i componenti invisibili: non solo non interagiscono con l'utente, ma non si vedono nemmeno. Servono soltanto ad aiutare nella visualizzazione di altri componenti.
Torniamo all'etichetta: per provarne l'utilizzo modifichiamo il programma visto nell'esempio precedente in modo che il suo aspetto grafico sia il seguente:



Le scritte "Inserisci del testo:" e "Al contrario:" sono visualizzate come etichette. Le etichette sono rappresentate dalla classe `JLabel`, il cui costruttore prende come argomento una stringa. Per creare le etichette in figura pertanto si possono usare le seguenti istruzioni:

```
JLabel l1 = new JLabel("Inserisci del testo:");
JLabel l2 = new JLabel("Al contrario:");
```

Dopodiché, le etichette possono essere visualizzate aggiungendole in un pannello esattamente come per i pulsanti e le caselle di testo.

Quesito

Si modifichi il programma dell'esercizio precedente in modo che appaia come nella figura qui sopra.

23. Componenti: le caselle di scelta (`JCheckBox` e `JRadioButton`)

Quando si vuole permettere all'utente di scegliere delle opzioni che possono essere vere o false, si utilizzano come componenti le cosiddette "caselle di scelta". Dal punto di vista dell'aspetto grafico, ci sono due tipi di caselle di scelta.
Il primo tipo, descritto dalla classe `JCheckBox`, rappresenta una scelta in cui le diverse opzioni sono indipendenti, e l'utente può sceglierne quante ne vuole. Ecco un esempio di `JCheckBox` (si tratta di tre caselle allineate orizzontalmente):

Il secondo tipo, descritto dalla classe `JRadioButton`, rappresenta invece una scelta **esclusiva**, in cui una e soltanto una opzione può essere selezionata in uno stesso istante. Ecco un esempio di `JRadioButton` (di nuovo, tre caselle orientate orizzontalmente):

La distinzione tra i due tipi di casella di scelta è soltanto grafica.
Poiché però gli utenti sono abituati che le caselle con il primo aspetto grafico funzionano in un certo modo, mentre quelle con il secondo aspetto grafico funzionano in un altro, sarà compito del programmatore fare sì che il funzionamento corrisponda a ciò che gli utenti si aspettano.
Cominciamo con il vedere un esempio in cui si utilizzano solo le `JCheckBox`, ossia in cui le scelte sono indipendenti.

ESERCIZIO A

Si costruisca un programma che permette all'utente di ottenere un preventivo per un viaggio a Parigi a seconda delle opzioni che sceglie. In particolare l'utente deve poter scegliere:

a) se includere o meno il viaggio A/R (L. 300000)
b) se includere o meno il pernottamento per 7 notti (L. 840000)
c) se includere o meno una gita a Eurodisney (L. 120000)

Il preventivo deve comparire in una casella di testo ed essere modificato qualora l'utente cambi le sue scelte.
Cerchiamo di affrontare il problema in modo sistematico.

Punto primo: scoprire le informazioni che ci servono sul nuovo componente

Quando decidiamo di usare un nuovo componente (in questo caso la `JCheckBox`), dobbiamo porci alcune domande e cercare le risposte.

1) Che argomenti prende il costruttore del componente?
Il costruttore di `JCheckBox` prende come argomento una stringa, che comparirà accanto alla casella.

2) Il componente interagisce con l'utente? Se sì, come?
L'utente può interagire con una `JCheckBox` cliccandoci sopra. Se la casella è selezionata, il click la deseleziona. Viceversa se non è selezionata, il click la seleziona. Quindi il click da parte dell'utente ha un effetto diverso a seconda dello stato attuale della casella.

3) Il componente genera degli eventi? Quando? Quali eventi genera?
L'oggetto `JCheckBox` genera un `ActionEvent` ogni volta che viene cliccato. Tuttavia l'`ActionEvent` non permette di distinguere se il click ha selezionato o deselezionato la casella. La `JCheckBox` genera anche un altro tipo di evento, detto `ItemEvent`, che viene generato ogni volta che la casella viene selezionata o deselezionata. Questo secondo tipo di evento permette di distinguere cosa è successo alla casella.

4) Come si ricevono gli eventi generati dal componente (se ce ne sono)?
Abbiamo già visto per i pulsanti e le caselle di testo come si riceve un `ActionEvent`. Per ricevere un `ItemEvent` bisogna invece implementare l'interfaccia `ItemListener`, che contiene un solo metodo con la seguente dichiarazione:

```
public void itemStateChanged(ItemEvent e)
```

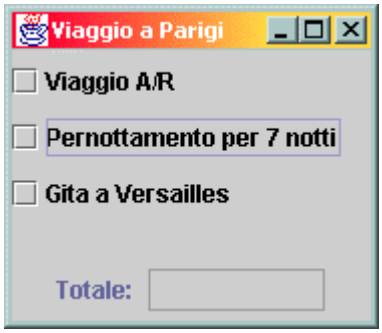
Per segnalare alla `JCheckBox` che si desidera ricevere l'evento, bisogna usare il metodo `addItemListener` su tutte le `JCheckBox` che ci interessano.

Punto secondo: costruire l'interfaccia grafica

Per prima cosa progettiamo l'interfaccia grafica del programma, senza pensare per ora a ricevere gli eventi. Se leggiamo le specifiche, vediamo che il nostro programma dovrà contenere tre oggetti `JCheckBox` (uno per il viaggio, uno per il pernottamento ed uno per la gita). Inoltre dovrà contenere una casella di testo per mostrare il preventivo, magari accompagnata da un'etichetta esplicativa. La casella di testo **non** deve poter essere modificata dall'utente.

Quesito

Costruire i componenti necessari e aggiungerli alla finestra principale `MyFrame` in modo che essa venga visualizzata come nella figura qui accanto (per il momento non ci si preoccupi di ricevere gli eventi e calcolare il preventivo). Si diano i seguenti nomi ai componenti creati:



`checkViaggio`
`checkNotti` → le caselle di scelta
`checkGita`

`textTotale` → la casella di testo

Punto terzo: aggiungere la funzionalità al programma

Per prima cosa è necessario ricevere gli eventi generati dalle `JCheckBox`. Abbiamo già detto cosa è necessario fare: implementare l'interfaccia `ItemListener`, aggiungendo quindi il metodo `itemStateChanged`, e segnalare alle `JCheckBox` che si vuole ricevere l'evento tramite il metodo `addItemListener`.
La funzionalità del programma è interamente gestita dal metodo `itemStateChanged` che viene chiamato quando lo stato di una delle `JCheckBox` cambia. È questo il punto in cui si deve aggiornare la cifra da visualizzare nella casella di testo. Per calcolare tale cifra abbiamo tuttavia bisogno di:
1 – delle costanti che memorizzino i costi delle singole opzioni
2 – una variabile di tipo `int` che memorizzi il totale attuale a cui apportare eventuali modifiche.
Poiché queste informazioni devono essere visibili in tutti i metodi, le aggiungiamo come membri dati della classe `MyFrame`:

```
class MyFrame extends JFrame implements ItemListener
{
  int COSTOVIAGGIO = 300000;
  int COSTONOTTI = 840000;
  int COSTOGITA = 120000;
  int totale;
  …
```

Si noti che convenzionalmente le costanti sono indicate in lettere maiuscole.
Ora vediamo come deve essere costruito il metodo `itemStateChanged`:

```
public void itemStateChanged(ItemEvent e)
{
  int cifra = 0;
  Object o = e.getSource();
  if (o == checkViaggio) cifra = COSTOVIAGGIO;
  else if (o == checkNotti) cifra = COSTONOTTI;
  else if (o == checkGita) cifra = COSTOGITA;
  int x = e.getStateChange();
  if (x == ItemEvent.SELECTED) totale += cifra;
  else if (x == ItemEvent.DESELECTED) totale -= cifra;
  textTotale.setText(“”+totale);
}
```

Note

a. Innanzitutto ricordiamo che il metodo viene chiamato quando **una qualunque** delle `JCheckBox` cambia stato. Pertanto la prima cosa da fare è capire **quale** delle caselle ha cambiato stato, e determinare la cifra corrispondente, che verrà memorizzata nella variabile `cifra`. A questo scopo si può utilizzare il metodo `getSource()` che **è presente nella classe `ItemEvent` così come in tutte le classi che rappresentano degli eventi (`ActionEvent`, `WindowEvent`, …)**. Esso resituisce un riferimento all'oggetto che ha causato l'evento (nel nostro caso, una delle caselle di scelta). Il metodo `getSource()` restituisce un oggetto di tipo `Object`, ossia un oggetto generico (ricordiamo che la classe `Object` è la classe a cui qualsiasi oggetto appartiene, la **radice** dell'albero delle classi). Quello che noi possiamo fare con l'oggetto è **confrontarlo** con le nostre caselle di scelta per verificare quale di esse si tratta.
Quando scriviamo un confronto come `(o == checkViaggio)` il risultato è `true` se e solo se `o` e `checkViaggio` sono due riferimenti **allo stesso oggetto in memoria**.

b. Una volta stabilita la casella di scelta che ha causato l'evento, e memorizzata la cifra corrispondente nella variabile `cifra`, dobbiamo stabilire se la casella è stata selezionata o deselezionata. Nel primo caso, infatti, dovremo **aggiungere** la cifra al totale, mentre nel secondo dovremo **sottrarla**. Per sapere cosa è successo dobbiamo chiamare il metodo `getStateChange` della classe `ItemEvent`. Esso resituisce un numero intero che è di fatto un codice per identificare l'accaduto. In particolare tale intero è uguale alla costante `ItemEvent.SELECTED` se la casella è stata selezionata, ed è uguale a `ItemEvent.DESELECTED` in caso contrario.

c. Quando il totale è stato aggiornato, dobbiamo ricopiarlo nella casella di testo. Questo viene fatto con l'istruzione `textTotale.setText(""+totale)`. Ricordiamo che `setText` si aspetta una stringa, mentre `totale` è un numero intero. Per ottenere una stringa con il numero contenuto in `totale` utilizziamo l'espressione `""+totale`, che appende in fondo alla stringa tra virgolette il numero contenuto nella variabile. Poiché la stringa tra virgolette è vuota, il risultato sarà una stringa con il numero soltanto.

d. **Attenzione**: il metodo `itemStateChanged` fa riferimento più volte alle variabili relative ai componenti grafici (`checkViaggio`, `textTotale`, …). Perché esse siano visibili all'interno del metodo ricordiamo che è necessario dichiararle come membri dati della classe `MyFrame`.

e. C'è un ultimo particolare da sistemare: è necessario decidere le impostazioni iniziali del programma, quelle che devono comparire quando esso viene avviato. Se stabiliamo che all'inizio nessuna casella è selezionata, allora la casella di testo dovrebbe visualizzare la cifra 0 (e la variabile `totale` dovrà essere inizializzata a zero). Diversamente, potremmo anche decidere che all'inizio alcune voci sono selezionate e quindi stabilire una cifra iniziale differente. Per fare sì che una casella di scelta sia selezionata si può usare il metodo `setSelected` che prende come argomento un booleano: `true` fa sì che la casella sia selezionata, `false` fa sì che non lo sia. Ad esempio, se vogliamo che all'inizio siano selezionate "Viaggio" e "Pernottamento" dovremmo fare:

```
checkViaggio.setSelected(true);
checkNotti.setSelected(true);
```

Attenzione: se il nostro `ItemListener` sta già ascoltando le caselle quando queste istruzioni vengono eseguite, il metodo `itemStateChanged` viene chiamato (perché le istruzioni modificano lo stato delle caselle) e dunque il totale aggiornato automaticamente.

ESERCIZIO B

Si modifichi il programma precedente in modo che l'utente possa, nel caso egli decida di acquistare il viaggio, scegliere fra il treno (L. 300000) e l'aereo (L. 450000).
La scelta fra aereo e treno è esclusiva: o si va in treno, oppure in aereo. Invece di usare la classe `JCheckBox` utilizzeremo pertanto la classe `JRadioButton`. Ricordiamo che le due classi sono assolutamente identiche dal punto di vista del funzionamento; cambia soltanto il nome del costruttore da chiamare!

Punto primo: scoprire le informazioni che ci servono sul nuovo componente

Abbiamo detto che `JCheckBox` e `JRadioButton` funzionano esattamente allo stesso modo (se ci rifacciamo le domande che ci siamo fatti per le `JCheckBox` le risposte sono esattamente le stesse).

Abbiamo anche detto che tipicamente le `JRadioButton` vengono usate quando la scelta è esclusiva, ossia una e soltanto una è l'opzione selezionata. Questo significa che quando l'utente seleziona una casella, quella precedentemente selezionata deve essere automaticamente deselezionata. Per ottenere questo effetto dobbiamo creare un **gruppo di caselle**, e aggiungervi quelle caselle che vogliamo siano esclusive. Questo è sufficiente ad ottenere l'effetto desiderato.
Un **gruppo di caselle** è descritto dalla classe `ButtonGroup` e può essere costruito come segue:

```
ButtonGroup gruppo = new ButtonGroup();
```

Per aggiungere una casella al gruppo è sufficiente la seguente istruzione:

```
gruppo.add(nomecasella);
```

Attenzione: il gruppo di caselle **non** è un componente grafico e pertanto non può essere inserito in quanto gruppo nella finestra. Le caselle devono venire inserite nella finestra una per una. Il gruppo serve solo a far sì che la scelta sia esclusiva.
Aggiungendo le caselle al gruppo otteniamo di **modificare** il modo in cui esse interagiscono con l'utente. Pertanto abbiamo anche risposte diverse ad alcune delle domande chiave:

1) Che argomenti prende il costruttore del componente?

Questo resta inalterato. Il costruttore di `JRadioButton` prende come argomento una stringa, che comparirà accanto alla casella.

2) Il componente interagisce con l'utente? Se sì, come?

L'interazione con l'utente viene modificata dall'appartenenza al gruppo. L'utente può interagire con una `JRadioButton` cliccandoci sopra. Se il click avviene su una casella selezionata, questa resta selezionata. Se il click avviene su una casella non selezionata, essa viene selezionata e quella precedentemente selezionata viene deselezionata automaticamente.

3) Il componente genera degli eventi? Quando? Quali eventi genera?

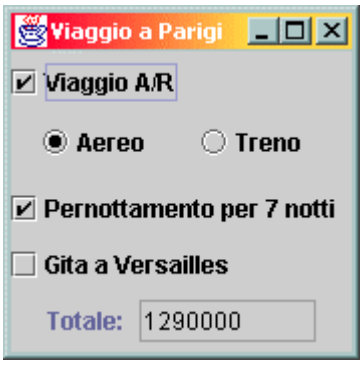
Il meccanismo degli eventi coincide con quello delle `JCheckBox`. L'oggetto `JRadioButton` genera un `ActionEvent` ogni volta che viene cliccato. Inoltre genera un `ItemEvent` ogni volta che la casella viene selezionata o deselezionata. Poiché un singolo click seleziona una casella e automaticamente ne deseleziona un'altra, ad ogni click corrispondono **due** `ItemEvent`. L'unica eccezione – talmente singolare da sembrare un errore! – è quando un utente clicca di nuovo su una casella già selezionata. In questo caso viene generato un solo `ItemEvent` corrispondente alla nuova selezione della casella.

4) Come si ricevono gli eventi generati dal componente (se ce ne sono)?

Esattamente come per le `JCheckBox`: con un `ItemListener`.

Punto secondo: modificare l'interfaccia grafica

La nuova interfaccia grafica dovrebbe apparire come nella figura qui accanto. Abbiamo aggiunto due oggetti `JRadioButton` (`radioAereo` e `radioTreno`), e abbiamo stabilito che all'inizio è selezionata la voce "Aereo". Inoltre abbiamo raggruppato le due `JRadioButton` in modo da rendere la scelta esclusiva. Per il momento la funzionalità non è stata modificata, per cui la cifra nella casella di testo non cambia se si sceglie "Treno" invece di "Aereo".



Quesito

Modificare il costruttore di `MyFrame` in modo che la finestra appaia come in figura.

Punto terzo: modificare la funzionalità

Se vogliamo far funzionare le due nuove caselle, dobbiamo fare sì che anche esse vengano ascoltate. Possiamo usare sempre il nostro `ItemListener`. Ora ci sono due aspetti da considerare: innanzitutto quando l'utente clicca su "Viaggio" (`checkViaggio`) la cifra da aggiungere o togliere dipenderà da quale tra "aereo" e "treno" è al momento selezionata. In secondo luogo, se l'utente dopo aver selezionato "Viaggio" cambia la sua scelta fra "Aereo" e "Treno" cliccando su `radioAereo` o `radioTreno`, la cifra deve cambiare. Quest'ultimo compito è reso più difficile dal fatto che l'utente potrebbe ciccare più volte sulla stessa casella generando molteplici eventi, e a causa del comportamento particolare delle caselle

esclusive noi non saremmo in grado di accorgerci se è la prima volta che ci clicca sopra o meno. Per semplificare le cose, pertanto, cambiamo radicalmente il metodo `itemStateChanged`, e invece di aggiungere o togliere dal totale a seconda di quanto è accaduto, ricalcoliamo ogni volta da zero il preventivo. Per cui il metodo diventa come segue:

```
public void itemStateChanged(ItemEvent e)
{
  totale = 0;
  if (checkViaggio.isSelected())
  {
     if (radioAereo.isSelected())
        totale = COSTOAEREO;
     else if (radioTreno.isSelected())
        totale = COSTOTRENO;
  }
  if (checkNotti.isSelected())
     totale += COSTONOTTI;
  if (checkGita.isSelected())
     totale += COSTOGITA;
  textTotale.setText(“”+totale);
}
```

Note

a. In questa versione, invece di controllare tramite l'oggetto `ItemEvent` cosa è successo, andiamo a chiedere a ciascuna delle caselle se al momento attuale è selezionata o no. Possiamo fare questo tramite il metodo `isSelected()`, che è definito sia per `JCheckBox` che per `JRadioButton`, e restituisce un valore di tipo `boolean`.

b. In realtà nel caso in cui l'utente non scelga di acquistare il viaggio, non ha molto senso che egli possa manipolare le caselle "Aereo" e "Treno". Un modo per impedirgli di farlo è quello di **disabilitarle** quando "Viaggio" non è selezionata. In generale qualunque componente può venire disabilitato: quando è disabilitato è visualizzato "sbiadito" e l'utente non può interagire con esso. Per disabilitare un qualunque componente si può chiamare su di esso il metodo `setEnabled` che prende come argomento un `boolean`: `true` abilita il componente, mentre `false` lo disabilita.
Nel nostro caso perciò aggiungiamo al metodo `itemStateChanged` le seguenti istruzioni:

```
if (checkViaggio.isSelected())
{
   radioAereo.setEnabled(true);
   radioTreno.setEnabled(true);
}
else
{
   radioAereo.setEnabled(false);
   radioTreno.setEnabled(false);
}
```

24. Creare un nuovo componente: un componente passivo

In questo esercizio vedremo come si crea un nuovo componente Java, invece che usarne uno predefinito. Per semplicità vedremo un componente "passivo" ossia un componente che non interagisce con l'utente. Ecco il testo dell'esercizio:
Progettare un componente Java che visualizzi un rettangolo con il bordo nero.
Come i componenti predefiniti, anche il nostro sarà descritto da una classe, che chiameremo `Rettangolo`. La prima regola da seguire è: **la classe che descrive il nostro componente deve estendere la classe `JComponent`, che è la classe base per tutti i componenti grafici.** La classe `JComponent` appartiene al package `javax.swing`, che pertanto bisogna ricordarsi di includere.
Avremo perciò in prima approssimazione:

```
import javax.swing.*;
public class Rettangolo extends JComponent
{
  …
}
```

Il secondo passo è stabilire quali sono i parametri che possono variare all'interno del componente, e stabilire come possono venire modificati (modalità di accesso). I parametri che sembra più logico inserire sono il colore del rettangolo, l'altezza e la larghezza. Questi devono diventare **membri dati** della classe. La larghezza e l'altezza possono essere facilmente memorizzate in variabili intere. Per quanto riguarda il colore, c'è una classe apposita per descriverlo: la classe `Color`, definita nel package `java.awt`. Tramite il costruttore di `Color` possiamo costruire un oggetto che rappresenta un colore ben preciso. Un colore è individuato dai tre parametri R (red), G (green) e B (blue), che rappresentano rispettivamente la quantità di rosso, verde e blu presente nel colore. I tre parametri possono assumere un valore intero fra 0 e 255, estremi inclusi.
Il costruttore della classe `Color` prende come argomenti i valori R,G e B; per costruire il colore rosso dovremo ad esempio scrivere `new Color(255,0,0)`.
Ad ogni modo, aggiungiamo i membri dati alla nostra classe:

```
import javax.swing.*;
import java.awt.*;
public class Rettangolo extends JComponent
{
  int larghezza;
  int altezza;
  Color colore;
  …
}
```

Adesso si tratta di decidere quando vogliamo che questi parametri possano essere impostati e modificati. Le possibilità sono principalmente due: o decidiamo che i parametri possono venire impostati una volta per tutte quando l'oggetto viene costruito, e in seguito mai modificati, oppure decidiamo che possono venire modificati in qualsiasi momento. Per ora consideriamo la prima possibilità.
Se vogliamo che i valori dei parametri possano essere decisi al momento della costruzione dell'oggetto, dobbiamo far sì che essi vengano passati al costruttore.
Il costruttore di `Rettangolo` avrà pertanto questa forma:

```
public Rettangolo(int l, int a, Color c)
{
  larghezza = l;
  altezza = a;
  colore = c;
  …
```

Ci sono però altri parametri, non stabiliti da noi ma esistenti nella classe `JComponent`, che dobbiamo impostare. In particolare dobbiamo impostare le dimensioni del componente. **Attenzione:** per il momento abbiamo impostato la larghezza e l'altezza del rettangolo che disegneremo. Le dimensioni del componente sono invece la larghezza e l'altezza che il componente va ad assumere all'interno della finestra in cui viene visualizzato, e possono essere differenti. In realtà noi non possiamo decidere le dimensioni esatte che il componente assumerà: queste infatti vengono stabilite dal Layout Manager. Quelle che possiamo impostare sono le dimensioni minime, le dimensioni massime, e le dimensioni preferite del nostro componente. Dobbiamo anche tenere presente che queste impostazioni vengono considerate semplicemente indicative dal Layout Manager, che potrebbe anche decidere di ignorarle.
Se non impostiamo alcuna dimensione minima e preferita potremmo avere delle conseguenze spiacevoli: infatti le dimensioni minime e preferite predefinite sono… 0! Il nostro componente potrebbe risultare pertanto invisibile. Nel nostro caso sarebbe opportuno che le dimensioni minime e preferite siano leggermente maggiori della larghezza e dell'altezza del rettangolo (diciamo 10 pixel in più). Pertanto aggiungeremo al costruttore le seguenti istruzioni:

```
    …
    Dimension dim = new Dimension(larghezza+10,altezza+10);
    setMinimumSize(dim);
    setPreferredSize(dim);
}
```

Per impostare le dimensioni dobbiamo innanzitutto creare un oggetto `Dimension` che memorizzi la larghezza (`width`) e l'altezza (`height`) da noi desiderate. La classe `Dimension` è definita nel package `java.awt` che abbiamo già incluso.
Dopodiché possiamo impostare le dimensioni minime e preferite con i due metodi `setMinimumSize` e `setPreferredSize`, definiti nella classe `JComponent`, che prendo come argomento proprio un oggetto `Dimension`. Se volessimo impostare anche le dimensioni massime potremmo usare il metodo `setMaximumSize`.

Adesso che abbiamo impostato i parametri del nostro componente, dobbiamo fare in modo che il rettangolo venga effettivamente disegnato sullo schermo. La classe `JComponent` ha un metodo `paint` che si occupa di disegnare il componente sullo schermo. Poiché un generico `JComponent` è completamente trasparente, il metodo `paint` in `JComponent` non fa nulla. **Per fare sì che il nostro componente venga disegnato sullo schermo come vogliamo noi dobbiamo ridefinire il metodo `paint`, la cui sintassi completa è la seguente:**

```
public void paint(Graphics g)
```

Il metodo `paint` di un componente viene chiamato dal "sistema" quando c'è da disegnare il componente sullo schermo. Come argomento viene passato a `paint` un oggetto `Graphics`. Possiamo immaginare un oggetto `Graphics` come il *pennello* con cui disegnare sullo schermo: esso infatti fornisce tutti i metodi necessari per farlo, e il modo di impostare il colore e altri parametri.
Proprio a proposito dell'oggetto `Graphics` c'è una particolarità da segnalare. A partire da Java 1.2 è stata introdotta una **sottoclasse** della classe `Graphics`, chiamata `Graphics2D`, che contiene metodi più avanzati per disegnare. Poiché si voleva mantenere la compatibilità con le vecchie versioni di Java, la sintassi del metodo `paint` non è cambiata, e l'argomento è sempre di tipo `Graphics`. L'oggetto che viene passato a `paint` è tuttavia un oggetto `Graphics2D` (il passaggio funziona proprio perché `Graphics2D` estende `Graphics`). Se vogliamo però usare le caratteristiche avanzate presenti nella classe `Graphics2D` dobbiamo però fare un'operazione di *cast*, ossia memorizzare l'oggetto in una variabile della classe che vogliamo usare (`Graphics2D`, appunto). Il metodo `paint` pertanto inizierà così:

```
public void paint(Graphics g)
{
  Graphics2D g2 = (Graphics2D)g;
  …
```

Attenzione: il mancato cast renderà impossibile utilizzare i metodi che vedremo in seguito, che sono specifici della classe `Graphics2D`.

Vediamo adesso come funziona il nostro *pennello*, `g2`.
Un oggetto `Graphics2D` è in grado di disegnare delle **forme** (vedremo dopo come specificare quale forma si vuole disegnare). In particolare data una forma un oggetto `Graphics2D` può fare due cose: disegnarne i contorni (operazione `draw`) oppure colorarne l'interno (operazione `fill`).
Per fare ciò necessita principalmente di due informazioni: il **colore** e il **tratto** da utilizzare.
Per quanto riguarda il colore, abbiamo già visto che esso è rappresentato dalla classe `Color`. Una volta in possesso di un oggetto `Color` possiamo impostare il colore del pennello chiamando su di esso il metodo `setPaint`, a cui passeremo l'oggetto `Color`.
Per quanto riguarda il tratto, esso specifica il tipo di linea che il pennello disegna. L'informazione più importante in questo caso è lo **spessore** della linea. Diversi tipi di linea possono essere rappresentati da diverse classi; ci concentriamo ora sul tipo più semplice, una linea continua di cui possiamo scegliere lo spessore. La classe che la rappresenta è la classe `BasicStroke`, il cui costruttore prende come argomento un numero che rappresenta lo spessore. Il tratto del pennello può poi essere impostato con il metodo

`setStroke` a cui possiamo passare l'oggetto `BasicStroke` che abbiamo costruito con lo spessore desiderato.
È importante tenere presente che il **tratto** è importante solo qualora si stia facendo un'operazione di disegno (`draw`), in quanto in un'operazione di riempimento (`fill`) tale informazione non viene utilizzata.
Per concludere, è necessario specificare come specificare al pennello quale forma deve disegnare o riempire. Una forma generica è rappresentata dalla classe `Shape` (definita nel package `java.awt.geom`, che va dunque incluso). Poiché ci sono molte forme diverse, la classe `Shape` ha molte sottoclassi, che rappresentano le forme più comuni. Vediamo quali sono queste sottoclassi e quali argomenti prendono i loro costruttori:

Forme	Classi	Argomenti del Costruttore
Rettangolo (è un quadrato se ha larghezza e altezza uguali)	`Rectangle2D.Float` `Rectangle2D.Double`	`x,y`: le coordinate dell'angolo superiore sinistro del rettangolo (float o double, a seconda della classe). `width,height`: la larghezza e l'altezza del rettangolo (float o double, a seconda della classe).
Rettangolo smussato	`RoundRectangle2D.Float` `RoundRectangle2D.Double`	`x,y`: le coordinate dell'angolo superiore sinistro del rettangolo (float o double, a seconda della classe). `width,height`: la larghezza e l'altezza del rettangolo (float o double, a seconda della classe). `sl,sa`: larghezza e altezza dello smusso del rettangolo (float o double, a seconda della classe).
Ellisse (è un cerchio se ha larghezza e altezza uguali)	`Ellipse2D.Float` `Ellipse2D.Double`	L'ellisse viene specificata tramite il rettangolo ad essa circoscritto; pertanto gli argomenti sono: `x,y`: le coordinate dell'angolo superiore sinistro del rettangolo circoscritto (float o double, a seconda della classe). `width,height`: la larghezza e l'altezza del rettangolo circoscritto (float o double, a seconda della classe).
Arco di ellisse	`Arc2D.Float` `Arc2D.Double`	Per specificare l'arco bisogna innanzitutto specificare l'ellisse a cui esso appartiene: `x,y`: le coordinate dell'angolo superiore sinistro del rettangolo circoscritto (float o double, a seconda della classe). `width,height`: la larghezza e l'altezza del rettangolo circoscritto (float o double, a seconda della classe). Inoltre abbiamo i seguenti argomenti: `start`: il punto di partenza dell'arco, in gradi, misurati a partire dalle ore 3 di un'orologio, movendosi in senso antiorario (float o double, a seconda della classe). `extent`: l'estensione dell'arco, sempre in gradi (float o double, a seconda della classe). `int t`: il tipo di arco, può essere uno dei seguenti valori costanti: `Arc2D.OPEN` è un arco aperto, senza alcuna linea che lo chiuda dall'interno. `Arc2D.CHORD` è un arco chiuso da una corda. `Arc2D.PIE` è un arco a spicchio.
Segmento	`Line2D.Float` `Line2D.Double`	`x1, y1`: le coordinate di un estremo del segmento (float o double, a seconda della classe). `x2, y2`: le coordinate del secondo estremo del segmento (float o double, a seconda della classe).

Ricapitoliamo i passi necessari per disegnare o riempire una forma con un oggetto `Graphics2D`:

(1) Creare un oggetto `Shape` che rappresenti la forma desiderata
(2) Impostare nell'oggetto `Graphics2D` un colore con il metodo `setPaint` a cui si può passare come argomento un oggetto `Color`
(3) Impostare – se necessario – un tratto con il metodo `setStroke` a cui si può passare come argomento un oggetto `BasicStroke`
(4) Chiamare sull'oggetto `Graphics2D` il metodo corrispondente all'operazione che si desidera effettuare (`draw` per disegnare i contorni della forma, `fill` per riempirne l'interno), a cui si deve passare l'oggetto `Shape` della forma da utilizzare.

Vediamo l'applicazione di quanto detto al nostro caso specifico, in cui vogliamo disegnare un rettangolo di dimensioni prefissate, che abbia l'interno di un colore prefissato, e il bordo nero.

```
public void paint(Graphics g)
{
  Graphics2D g2 = (Graphics2D)g;
  Shape rettangolo = new Rectangle2D.Float(0,0,larghezza,altezza);
  g2.setColor(colore);
  g2.fill(rettangolo);
  g2.setStroke(new BasicStroke(3));
  g2.setColor(new Color(0,0,0));
  g2.draw(rettangolo);
}
```

Note

a. Innanzitutto abbiamo creato un rettangolo delle dimensioni stabilite nel costruttore, le cui coordinate (ricordiamo, dell'angolo superiore sinistro) sono 0,0.
b. In secondo luogo abbiamo impostato il colore stabilito nel costruttore e abbiamo chiamato il metodo `fill` per riempire l'interno del rettangolo di quel colore.
c. Poi abbiamo impostato un tratto dello spessore di 3 pixel, abbiamo impostato il colore nero (RGB 0,0,0) e abbiamo chiamato il metodo `draw` per disegnare il contorno del rettangolo.
d. Si noti che è opportuno prima colorare l'interno e poi disegnare il contorno, per evitare che il riempimento copra il bordo nero.

Esercizi

1) Si costruisca una finestra che visualizza il componente appena creato. Si provi a vedere cosa accade ingrandendo o rimpicciolendo manualmente la finestra.
2) Si modifichi la classe `Rettangolo` in modo che il rettangolo venga disegnato al centro del componente. A questo scopo è necessario conoscere le dimensioni del componente nel momento in cui viene disegnato: per fare questo si può usare il metodo `getSize()` della classe `JComponent` che restituisce un oggetto `Dimension` con le dimensioni attuali. La larghezza e l'altezza si trovano rispettivamente nei campi `width` e `height` dell'oggetto ottenuto.

25. Creare un nuovo componente: un componente modificabile

Vogliamo ora modificare leggermente il componente definito precedentemente in modo che il colore possa essere cambiato in qualsiasi momento, e non più stabilito una volta per tutte nel costruttore. In sostanza quello passato al costruttore sarà il **colore iniziale** del rettangolo, che potrà poi successivamente essere cambiato. Si noti che il componente continua ad essere passivo: infatti non interagisce con l'utente in alcun modo.
Per fare sì che un parametro del componente – in questo caso il colore – sia modificabile, dobbiamo aggiungere alla classe `Rettangolo` un metodo che effettui la modifica. La cosa non è ovvia come sembra: infatti non è sufficiente cambiare il valore del membro dati `colore`. Se a quel punto il rettangolo è già stato disegnato, cambiare il valore del membro dati non serve a nulla. Dobbiamo anche fare in modo che il rettangolo venga ridisegnato. Ecco come dovrebbe essere il metodo da aggiungere:

```
public void cambiaColore(Color nuovo)
{
  colore = nuovo;
  repaint();
}
```

Nota

Il metodo `repaint()` è definito per tutti i componenti (nella classe `Component`, che è superclasse di `JComponent`) e serve proprio a fare sì che il componente venga ridisegnato.

Esercizio

Costruire un programma che visualizzi il rettangolo, permettendo all'utente di cambiarne il colore scegliendo fra alcune opzioni. Per la scelta fra le opzioni si utilizzino delle `JRadioButton`.

26. Creare un nuovo componente: un componente attivo

Vogliamo ora provare a creare un componente attivo, con cui l'utente possa interagire, ad esempio cliccandoci sopra con il mouse. Il testo dell'esercizio è il seguente:
Costruire un componente `Tavolozza`*, che visualizzi sedici colori come quadratini disposti in una griglia 4x4, con sfondo bianco. L'utente può selezionare uno dei colori cliccando sul quadratino corrispondente. Il quadratino selezionato dovrà essere visualizzato con un rettangolo rosso intorno ad esso.*
Nello svolgere questo esercizio dobbiamo per prima cosa prendere alcune decisioni riguardo ad aspetti non specificati nel testo. Innanzitutto stabiliamo che i sedici colori sono fissi, e non possono essere modificati. Inoltre decidiamo che ciascun quadratino ha dimensioni 16x16, pertanto l'intera tavolozza avrà dimensioni 64x64. Come dimensioni minime e preferite per il componente decidiamo di scegliere 80x80, in modo da avere un po' di bordo intorno alla tavolozza.
Deciso questo, cominciamo con il creare il componente e disegnarlo sullo schermo. Dopodiché aggiungeremo l'interazione con l'utente.
Ecco il codice iniziale per la classe `Tavolozza`:

```
import java.awt.geom.*;
import java.awt.*;
import java.awt.event.*;
public class Tavolozza extends JComponent
{
  Color[] colori;
  public Tavolozza()
  {
     colori = new Color[16];
     colori[0] = new Color(255,255,255);
     colori[1] = new Color(255,0,0);
     colori[2] = new Color(0,255,0);
     colori[3] = new Color(0,0,255);
     colori[4] = new Color(255,255,0);
     colori[5] = new Color(255,0,255);
     colori[6] = new Color(0,255,255);
     colori[7] = new Color(127,0,0);
     colori[8] = new Color(0,127,0);
     colori[9] = new Color(0,0,127);
     colori[10] = new Color(127,127,0);
     colori[11] = new Color(127,0,127);
     colori[12] = new Color(0,127,127);
     colori[13] = new Color(193,193,193);
     colori[14] = new Color(127,127,127);
     colori[15] = new Color(0,0,0);
     setMinimumSize(new Dimension(80,80));
     setPreferredSize(new Dimension(80,80));
  }
```

```
   public void paint(Graphics g)
   {
      Graphics2D g2 = (Graphics2D)g;
      Dimension size = getSize();
      Shape sfondo = new
                     Rectangle2D.Float(0,0,size.width,size.height);
      g2.setPaint(new Color(255,255,255));
      g2.fill(sfondo);
      float x0 = size.width/2 - 32;
      float y0 = size.height/2 - 32;
      for (int i=0; i < 16; i++)
      {
         int riga = i/4;
         int col = i%4;
         int yi = riga*16;
         int xi = col*16;
         Shape quadratino = new Rectangle2D.Float(x0+xi,y0+yi,16,16);
         g2.setColor(colori[i]);
         g2.fill(quadratino);
      }
   }
}
```

Note

i) Per quanto riguarda il costruttore c'è poco da commentare. Guardiamo invece il metodo `paint`. Per prima cosa vogliamo colorare lo sfondo del componente di bianco. Per fare questo creiamo un rettangolo grande quanto tutto il componente (`sfondo`), impostiamo il colore bianco (RGB 255,255,255) e riempiamo il rettangolo (`fill`).

ii) Ora vogliamo disegnare i quadratini. Abbiamo detto che vogliamo disporli su 4 righe e 4 colonne. Per evitare di creare a mano i sedici quadratini, facciamo un ciclo **for** in cui calcoliamo automaticamente la riga e la colonna di un quadratino, e in base ad esse le sue coordinate. È facile controllare che il sistema adottato è corretto.

Quesito

Costruire una finestra che visualizzi il componente appena creato, per controllare che sia come dovuto.

A questo punto vogliamo aggiungere l'interazione con l'utente. La classe che abbiamo definito non ha nessun concetto di "quadratino selezionato": tutti i quadratini sono uguali. Dobbiamo quindi innanzitutto aggiungere un membro dati che memorizzi quale dei quadratini è selezionato in quel momento. È sufficiente ricordare l'indice del colore selezionato nell'array, quindi possiamo aggiungere un membro dati intero:

```
int selezionato;
```

Il membro dati deve poi essere inizializzato nel costruttore. Possiamo decidere che all'inizio, quando l'utente non ha ancora cliccato da nessuna parte, il colore selezionato è il primo, ossia quello di indice 0. Perciò aggiungiamo nel costruttore la seguente istruzione:

```
selezionato = 0;
```

Per quanto riguarda il metodo `paint`, esso deve essere fatto in modo da disegnare un bordo rosso al quadratino selezionato. Ecco come risulta:

```
public void paint(Graphics g)
{
  Graphics2D g2 = (Graphics2D)g;
  Dimension size = getSize();
  Shape sfondo = new
              Rectangle2D.Float(0,0,size.width,size.height);
  g2.setPaint(new Color(255,255,255));
  g2.fill(sfondo);
  Shape sel = null;
  float x0 = size.width/2 - 32;
  float y0 = size.height/2 - 32;
  for (int i=0; i < 16; i++)
  {
     int riga = i/4;
     int col = i%4;
     int yi = riga*16;
     int xi = col*16;
     Shape quadratino = new Rectangle2D.Float(x0+xi,y0+yi,16,16);
     g2.setColor(colori[i]);
     g2.fill(quadratino);
     if (i == selezionato)
        sel = quadratino;
  }
  g2.setPaint(new Color(255,0,0));
  g2.setStroke(new BasicStroke(3));
  g2.draw(sel);
}
```

Nota

Come al solito, conviene disegnare il bordo al quadratino selezionato dopo aver riempito tutti i quadratini, per evitare che il riempimento di un quadratino adiacente lo copra. Pertanto quando incontriamo l'indice corrispondente al quadratino selezionato (`i == selezionato`) memorizziamo il quadratino nella variabile `sel`. Al termine del ciclo potremo disegnare (`draw`) il bordo, dopo aver impostato il colore rosso (RGB 255,0,0) e un tratto di spessore 3.

A questo punto manca la parte più importante: l'interazione con l'utente, che cliccando con il mouse deve poter selezionare un quadratino diverso dal primo. A questo scopo dobbiamo accorgerci quando l'utente clicca sul mouse nella porzione di finestra occupata dalla `Tavolozza`. Fortunatamente parte del lavoro è già stato fatto nella classe `JComponent`. Infatti quando l'utente fa qualcosa con il mouse "sopra" un qualunque componente, esso genera un evento di tipo `MouseEvent`. A noi non resta altro da fare che catturare l'evento e rispondere in qualche modo (ad esempio, disegnando un quadrato rosso intorno al quadratino selezionato).
Per catturare l'evento, come al solito, è necessario **implementare un'interfaccia**. Sfortunatamente c'è una piccola complicazione: per catturare un `MouseEvent` ci sono due interfacce diverse. Per catturare i `MouseEvent` legati al movimento del mouse c'è l'interfaccia `MouseMotionListener`. Per catturare i `MouseEvent` legati alla pressione dei pulsanti c'è l'interfaccia `MouseListener`. Entrambe sono definite nel package `java.awt.event`. C'è poi un'interfaccia che le raggruppa, che si chiama `MouseInputListener` ed è definita nel package `java.swing.event`.
Vediamo i metodi delle due interfacce:

Interfaccia `MouseMotionListener`	
`public void mouseMoved(MouseEvent e)`	L'utente ha mosso il mouse.
`public void mouseDragged(MouseEvent e)`	L'utente ha mosso il mouse tenendo contemporaneamente premuto un pulsante.

Interfaccia `MouseListener`	
`public void mouseClicked(MouseEvent e)`	L'utente ha cliccato con il mouse sul componente.
`public void mousePressed(MouseEvent e)`	L'utente ha premuto un pulsante del mouse.
`public void mouseReleased(MouseEvent e)`	L'utente ha rilasciato un pulsante del mouse precedentemente premuto.
`public void mouseEntered(MouseEvent e)`	Il mouse è entrato nel componente.
`public void mouseExited(MouseEvent e)`	Il mouse è uscito dal componente.

Poiché noi siamo interessati ad accorgerci di quando l'utente **clicca con il mouse** l'interfaccia che a noi interessa implementare è `MouseListener`.
Poiché l'unico metodo che a noi interessa definire è `mouseClicked` sarebbe comodo poter usare una classe `Adapter` per evitarci di dover implementare anche tutti i metodi che non ci servono. La classe `Adapter` c'è, si tratta di `MouseAdapter`. Sfortunatamente però c'è un ostacolo all'utilizzarla: quando riceviamo l'evento vogliamo modificare delle informazioni relative alla `Tavolozza`, in particolare la variabile `selezionato` che memorizza l'indice del quadratino selezionato. Se creiamo una nuova classe per ricevere gli eventi essa non avrà accesso alle informazioni personali di `Tavolozza`, e pertanto non potrà modificare la variabile `selezionato`. D'altro canto, se facciamo implementare a `Tavolozza` stessa l'interfaccia, perdiamo la possibilità di estendere la classe `Adapter`.
La soluzione consiste nel definire una **classe interna**. Una classe interna è una classe che viene **definita dentro un'altra classe** (ossia all'interno delle parentesi graffe che la delimitano). Poiché è a tutti gli effetti una classe diversa, può estendere ciò che vuole (in particolare, la classe `MouseAdapter` che a noi interessa), ma può anche "vedere" i membri dati della classe in cui è contenuta.
Ecco una versione provvisoria di quanto dobbiamo fare per ricevere l'evento corrispondente ad un click del mouse:

```
public class Tavolozza extends JComponent
{
  class AscoltaMouse extends MouseAdapter
  {
     public void mouseClicked(MouseEvent e)
     {
        //qui dobbiamo aggiungere delle istruzioni!
     }
  }

  Color[] colori;
  public Tavolozza()
  {
     colori = new Color[16];
     colori[0] = new Color(255,255,255);
     …
     colori[15] = new Color(0,0,0);
     setMinimumSize(new Dimension(80,80));
     setPreferredSize(new Dimension(80,80));
     addMouseListener(new AscoltaMouse());
  }
  …
```

Note

a. Come si può vedere la classe `AscolteMouse` è stata definita **dentro** la classe `Tavolozza`. Si tratta di una versione provvisoria perché il metodo `mouseClicked` non contiene nessuna istruzione.
b. Per ricevere effettivamente gli eventi, nel costruttore dobbiamo creare un oggetto di tipo `AscoltaMouse` e registrarlo presso il componente stesso con il metodo `addMouseListener`.

Per concludere, vediamo cosa deve fare il metodo `mouseClicked`. Per prima cosa sarà necessario stabilire la posizione del mouse quando è stato cliccato il pulsante. Questa informazione è in possesso dell'oggetto `MouseEvent`, che ha due metodi, `getX()` e `getY()` che restituiscono rispettivamente l'ascissa e

l'ordinata del punto in questione. In secondo luogo a partire dalle coordinate dobbiamo risalire a quale quadratino si trova in quel punto, per impostare il valore giusto per la variabile `selezionato`. Infine dobbiamo chiedere di ridisegnare il componente, in modo che il bordo venga posto intorno al quadratino giusto.

```
public void mouseClicked(MouseEvent e)
{
  int x = e.getX();
  int y = e.getY();
  Dimension dim = getSize();
  int x0 = dim.width/2 - 32;
  int y0 = dim.height/2 - 32;
  int xRel = x - x0;
  int yRel = y - y0;
  int riga = yRel / 16;
  int col = xRel / 16;
  selezionato = riga*4 + colonna;
  repaint();
}
```

Nota

Si noti che prima di risalire all'indice a partire dalle coordinate, è opportuno determinare le coordinate relative all'angolo superiore sinistro della tavolozza (`x0,y0`), che deve essere calcolato.

Quesito

Si verifichi che l'interazione con l'utente funziona.

27. I bordi di un pannello

È possibile impostare un bordo ai pannelli, in modo da separarli visivamente l'uno dall'altro. La classe che rappresenta un bordo è la classe `Border`, definita nel package `javax.swing.border`. Ci sono alcuni tipi di bordi predefiniti che possiamo creare molto semplicemente utilizzando dei metodi statici definiti nella classe `BorderFactory`. Ciascuno di questi metodi – riportati nella tabella qui sotto insieme ad un esempio del bordo che essi creano – restituisce un oggetto di tipo `Border`. Per impostare il bordo ad un oggetto `JPanel` possiamo chiamare il metodo di quest'ultimo `setBorder`, che prende come argomento per l'appunto un oggetto `Border`.

Questa finestra non ha alcun bordo, contiene solo un bottone al centro.

Questo è un'esempio di **Bordo Vuoto**: un bordo invisibile che serve solo a lasciare uno spazio maggiore intorno al bottone. Per costruirlo:
`BorderFactory.createEmptyBorder(int t, int l, int b, int r)`
I parametri indicano lo spessore del bordo, rispettivamente in alto, a sinistra, in basso e a destra.

Questo è un esempio di **Bordo ad Incisione**. Per costruirlo:
`BorderFactory.createEtchedBorder()`

Questo è un esempio di Bordo Scolpito. In particolare il primo è un Bordo Scolpito in rilievo mentre il secondo è un Bordo Scolpito ad incasso. Per costruirlo:

```
BorderFactory.createBevelBorder(int tipo)
```

Il parametro può assumere uno dei due valori costanti BevelBorder.RAISED e BevelBorder.LOWERED. Nel primo caso il bordo risulterà in rilievo, nel secondo ad incasso.

Questo è un esempio di Bordo a Linea: una semplice linea intorno al pannello. Per costruirlo:

```
BorderFactory.createLineBorder(Color c)
```

Il parametro è il colore in cui disegnare la linea.

Questo è un esempio di Bordo Tinta Unita. Per costruirlo:

```
BorderFactory.createMatteBorder(int t, int l,
                                int b, int r, Color c)
```

I primi quattro parametri sono lo spessore del bordo rispettivamente in alto, a sinistra, in basso e a destra. Il quinto parametro è il colore del bordo.

Questo è un esempio di Bordo Composto, ossia un bordo ottenuto componendo insieme altri due bordi. Nell'esempio il bordo esterno è un bordo scolpito in rilievo, mentre quello più interno è un bordo scolpito ad incasso. Per costruirlo:

```
BorderFactory.createCompoundBorder(Border e, Border i)
```

I due parametri sono i due bordi da comporre, rispettivamente quello esterno e quello interno.

Questo è un esempio di Bordo con Titolo: un bordo a cui viene aggiunta una scritta. Per costruirlo:

```
BorderFactory.createTitledBorder(Border b, String s)
```

Il primo parametro è il bordo a cui aggiungere la scritta (nell'esempio è un bordo ad incisione), il secondo la scritta da aggiungere (nell'esempio, "Titolo").

Vediamo adesso come esempio il codice che crea la finestra dell'esempio, in particolare quella con il bordo composto:

```
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;

public class Finestrella extends JFrame
{
 public Finestrella()
 {
  super("Bordi");
  addWindowListener(new AscoltaFinestra());
  JPanel p = new JPanel();
  p.add(new JButton("X"));
  Border int = BorderFactory.createBevelBorder(BevelBorder.RAISED);
  Border est = BorderFactory.createBevelBorder(BevelBorder.LOWERED);
  Border comp = BorderFactor.createCompoundBorder(est,int);
  p.setBorder(comp);
  getContentPane().add(p);
 }
}
```

ESERCIZI DI RIEPILOGO

1. Multicolor

Parte 1 (FACILE)

Argomenti trattati: uso di `JTextField`, conversione di una stringa di testo in numero.
Si riprenda la classe che disegna un rettangolo.
Si costruisca una finestra in cui l'utente può specificare, tramite caselle di testo, i valori dei tre parametri RGB che compongono un colore.
Quando l'utente cambia uno di tali valori e preme INVIO, il rettangolo cambia colore.

Parte 3 (MEDI0)

Argomenti trattati: utilizzo di un componente predefinito che non è stato visto a lezione, ossia `JSlider`.
Si sostituiscano le tre caselle di testo per i valori RGB con altrettanti componenti di tipo `JSlider`.
Un componente `JSlider` è raffigurato come una levetta che è possibile spostare tra due estremi. A ciascuna posizione della leva corrisponde un valore.
Le informazioni necessarie per utilizzare questo nuovo tipo di componente sono le seguenti:

1. la classe `JSlider` è definita nel package `javax.swing`;
2. il costruttore di `JSlider` prende i seguenti argomenti:
   - un valore che stabilisce se la levetta deve spostarsi orizzontalmente o verticalmente; può essere uguale ad una delle due costanti `JSlider.HORIZONTAL` o `JSlider.VERTICAL`.
   - il valore intero corrispondente all'estremo inferiore (nel caso dei parametri RGB il minimo è 0)
   - il valore intero corrispondente all'estremo superiore (nel caso dei parametri RGB il massimo è 255)
   - il valore iniziale su cui deve essere posizionata la levetta.
3. Una volta costruito e inserito nella finestra, l'oggetto `JSlider` può essere utilizzato dall'utente. Per accorgersi di quando viene spostato però bisogna catturare gli eventi che esso genera.
4. È possibile impostare un valore iniziale per uno `JSlider` con il suo metodo `setValue`, a cui passare il valore (intero).
5. Un oggetto `JSlider` genera eventi di tipo `ChangeEvent`. Per riceverli è necessario implementare l'interfaccia `ChangeListener`, definita in `javax.swing.event`, che contiene un solo metodo: `public void stateChanged(ChangeEvent e)`.
6. La classe `ChangeEvent` fornisce un metodo per conoscere il valore associato alla nuova posizione della levetta: si tratta di `getValue()`, che restituisce un intero.

Parte 3 (DIFFICILE)

Argomenti trattati: gestione degli eventi legati al movimento del mouse.
Ora si faccia in modo che l'utente possa spostare il rettangolo "trascinandolo", ossia cliccandoci sopra e movendo il mouse.
Suggerimento: per ricevere gli eventi legati al movimento del mouse bisogna implementare l'interfaccia `MouseMotionListener`. In particolare serve il metodo `mouseDragged`, che viene chiamato quando l'utente muove il mouse sul componente tenendo premuto un pulsante. **Attenzione:** il metodo viene chiamato ad ogni spostamento del mouse, e l'evento che viene passato contiene le coordinate della **nuova** posizione del mouse. Per determinare di quanto si deve spostare il rettangolo, sono necessarie anche le coordinate precedenti. Sarà pertanto necessario memorizzarle di volta in volta per averle a disposizione.
Suggerimento 2: `MouseMotionListener` potrebbe non essere sufficiente: potrebbe infatti essere necessario sapere dove si trova il mouse quando il pulsante viene premuto per la prima volta dall'utente. Questo tipo di evento è ricevuto dall'interfaccia `MouseListener`. Se si vogliono implementare entrambe le interfacce, si può utilizzare la classe `MouseInputAdapter`, definita nel package `javax.swing.event`, che fa simultaneamente da adapter sia per `MouseListener` che per `MouseMotionListener`.

2. La griglia

Parte 1 (FACILE)

Si definisca un componente `Griglia` che raffigura una griglia (ossia una serie di linee verticali e orizzontali equidistanziate). La griglia deve occupare tutto il componente: per verificare che ciò succeda si crei una finestra contenente soltanto il componente griglia e la si allarghi a tutto schermo. La griglia dovrebbe occupare tutta la finestra.
Si costruisca ora una programma che oltre a visualizzare la griglia offre all'utente due caselle di testo in cui specificare la spaziatura orizzontale e verticale della griglia. Inizialmente le due caselle devono mostrare la spaziatura iniziale della griglia. Quando l'utente modifica i valori e preme INVIO, la griglia deve venire modificata in accordo a quanto specificato dall'utente.

Parte 2 (DIFFICILE)

La griglia dell'esercizio precedente divide il componente in tanti quadratini. Si faccia in modo che quando l'utente clicca su uno dei quadratini esso si colori di nero. Se l'utente ci clicca sopra una seconda volta, esso scompare (diventa trasparente – ossia: non viene più disegnato).
Se la spaziatura della griglia cambia, eventuali quadratini neri presenti si cancellano.
Si preveda la possibilità per l'utente di scegliere di nascondere la griglia, ossia non visualizzare le linee orizzontali e verticali che la compongono, ma solo i quadratini neri.

3. Grafico a torta

Parte 1 (FACILE/MEDIO)

Si crei un componente che permette di memorizzare fino a dieci valori, e visualizza un grafico a torta che rappresenta la percentuale di ciascun valore sul totale.
Se nessun valore è memorizzato, non viene visualizzato nulla.
Si usi un array per memorizzare i dieci valori.
Si crei quindi una finestra in cui l'utente può inserire i valori. L'utente scrive in una casella di testo il valore, e poi preme un pulsante "Aggiungi". Il nuovo valore viene aggiunto al grafico, che deve visualizzarlo.
Quando è stato raggiunto il limite di dieci valori il pulsante "Aggiungi" deve essere disabilitato.

4. La spezzata

Parte 1 (MEDIO)

Si definisca un componente che si comporta nel modo seguente: quando un utente ci clicca sopra, disegna un pallino rosso nel punto in cui l'utente ha cliccato, e lo unisce con un trattino al puntino precedente (se esiste)
Suggerimento: si fissi un numero massimo di puntini che l'utente può aggiungere, e si utilizzi un array per memorizzare la sequenza di puntini.
Suggerimento 2: nel package `java.awt.geom` sono definite due classi, `Point2D.Float` e `Point2D.Double` per memorizzare punti con coordinate rispettivamente float e double. I costruttori prendono come argomento le coordinate da assegnare al punto. Le coordinate sono memorizzate come membri dati pubblici perciò dato un oggetto `p` di tipo `Point2D` posso ottenerle scrivendo `p.x` e `p.y`.

Si costruisca una finestra in cui l'utente può utilizzare il componente, e in cui ci sono due pulsanti: il primo cancella un puntino per volta a partire dall'ultimo aggiunto, mentre il secondo cancella tutto. Si trovi inoltre un modo per segnalare all'utente quando ha raggiunto il numero massimo di puntini.

Parte 2 (DIFFICILE)

Invece che utilizzare un array si provi ad utilizzare un oggetto della classe `ArrayList`, definita nel package `java.util`. Il costruttore di tale classe non prende argomenti.
Un oggetto della classe `ArrayList` si comporta essenzialmente come un array, solo che si possono aggiungere quanti elementi si desidera (in questo modo si può togliere il limite sul numero di puntini).
I metodi principali della classe `ArrayList` sono:
`boolean add(Object o)` per aggiungere un oggetto in fondo alla lista
`Object remove(int i)` per rimuovere l'oggetto di indice `i` dalla lista (restituisce l'oggetto rimosso)
`int size()` per sapere quanti oggetti ci sono nella lista

`Object get(int i)` per farsi dare l'oggetto di indice `i` nella lista
`void clear()` per eliminare tutti gli oggetti nella lista
Lo scopo di questa parte dell'esercizio è quindi togliere il limite sul numero di puntini che l'utente può inserire utilizzando un `ArrayList` al posto di un normale array.

5. La retta

Parte 1 (FACILE)

Vogliamo costruire un programma che permetta all'utente di specificare una retta (tramite ad esempio il suo coefficiente angolare e il suo termine noto), e la visualizzi su un piano cartesiano.
Il piano cartesiano deve essere un nuovo componente che disegna gli assi cartesiani in una posizione prefissata (ad esempio al centro) ed è in grado di disegnare una retta dati il suo termine noto e il suo coefficiente angolare. Si definisca una scala (ad esempio 1 pixel = 1 unità, oppure 10 pixel = 1 unità, come si preferisce)
La finestra del programma deve fornire due caselle di testo in cui l'utente può specificare il termine noto e il coefficiente angolare. Ci deve essere poi un pulsante "Disegna": alla pressione del pulsante la retta viene disegnata.
Si evidenzino inoltre i punti di intersezione – se ci sono e sono visibili – della retta con gli assi, ad esempio disegnando un circoletto di colore diverso.

Parte 2 (FACILE/MEDIO)

Estendiamo il programma precedente in modo che l'utente possa scegliere se disegnare una retta o un'ellisse. Il componente che realizza il piano cartesiano deve essere modificato in modo da poter disegnare o una retta o un'ellisse.
La finestra del programma deve fornire, oltre a ciò che era descritto nel punto precedente, la possibilità di scegliere tra retta ed ellisse. Inoltre devono esserci delle caselle di testo per specificare la larghezza e l'altezza dell'ellisse.
Se l'utente decide di disegnare una retta, le caselle di testo relative all'ellisse devono essere disabilitate. Se l'utente decide di disegnare un'ellisse, le caselle di testo relative alla retta devono essere disabilitate.
Sarebbe carino che venissero anche evidenziati i fuochi dell'ellisse… ne siete capaci?
Suggerimento: la radice quadrata può essere calcolata utilizzando il metodo statico:
`double Math.sqrt(double d)`.

Parte 3 (DIFFICILE)

La cosa migliore in questo programma sarebbe che quando l'utente sceglie di disegnare una retta venissero visualizzate **solo** le caselle di testo relative alla retta, e analogamente, se l'utente sceglie di disegnare un'ellisse, dovrebbero venire disegnate **solo** le caselle di testo relative all'ellisse.
Questo si può ottenere utilizzando come Layout Manager il `CardLayout`. Esso infatti visualizza un componente alla volta (dove "un componente" può essere anche un intero pannello), e fornisce dei metodi per scegliere quale componente visualizzare.
Qui sotto sono riportate le informazioni necessarie per utilizzare un `CardLayout`:

1. Il costruttore non prende alcun argomento: `new CardLayout()`
2. Dopo aver assegnato il `LayoutManager` possiamo aggiungere i componenti usando il solito metodo `add` a cui però bisogna **passare due parametri**: il primo è il **componente da aggiungere**, il secondo è una **stringa che diventa il "nome" del componente. Tale nome verrà poi usato quando dovremo specificare quale dei componenti deve essere visualizzato**. Ad esempio, se la variabile `p` rappresenta un `JPanel` e la variabile `b` rappresenta un bottone, scriveremo `p.add(b, "Bottone")`.
3. Per scegliere quale componente visualizzare, dobbiamo chiamare il metodo `show` **che appartiene alla classe** `CardLayout`. Il metodo `show` prende come argomenti il pannello che possiede il card layout e il nome del componente da visualizzare. Ecco un esempio un po' più completo:

```
JPanel p = new JPanel();
CardLayout card = new CardLayout();
p.setLayout(card);
p.add(new JButton("Premi qui"), "Bottone");
p.add(new Jlabel("Ciao!"), "Etichetta");
```

Se ci fermiamo qui, alla visualizzazione della finestra il `CardLayout` mostrerà il bottone, ossia il primo componente che gli è stato aggiunto. Se invece vogliamo visualizzare l'etichetta dobbiamo aggiungere l'istruzione:

```
card.show(p,"Etichetta");
```

Naturalmente nell'esercizio proposto ciò che viene mostrato dovrà cambiare a seconda di ciò che l'utente sceglie di disegnare (retta o ellisse).

6. Triangoli!

Parte 1 (FACILE)

Tra le forme che abbiamo visto non esistono i triangoli. Esiste però una forma particolare che permette di definire qualsiasi poligono, semplicemente specificandone i punti. Si tratta della classe `Polygon`, sottoclasse di `Shape`. Per creare un poligono vuoto (ossia senza punti) posso semplicemente scrivere:

```
Polygon p = new Polygon();
```

A questo punto per ottenere un triangolo devo semplicemente aggiungere al poligono i tre punti che lo compongono, nell'ordine. Per aggiungere un punto si può usare il metodo `addPoint` che prende come argomento due interi, rispettivamente le coordinate `x` e `y`.
Ad esempio quindi scriveremo:

```
p.addPoint(0,40);
p.addPoint(40,40);
p.addPoint(20,0);
```

(queste istruzioni creano un triangolo isoscele con la base parallela all'asse X.)

Si crei un componente che, date le coordinate dei vertici del triangolo, è in grado di disegnarlo.
Deve essere possibile cambiare tali coordinate dall'esterno della classe (ossia con metodi appositi).
Si provi a fare un programma in cui l'utente può specificare le coordinate dei punti del triangolo (ad esempio scrivendole in caselle di testo), e il triangolo viene ridisegnato alle coordinate richieste dall'utente.

Parte 2 (MEDIA)

Si modifichi il programma precedente in modo che l'utente possa specificare le coordinate non scrivendole in caselle di testo, bensì cliccando nei punti in cui desidera siano i vertici del triangolo. È ovvio che fino a che l'utente non ha fatto almeno tre click non compare nessun triangolo.
Dopo il terzo click l'utente non può più modificare le coordinate. Si fornisca però un pulsante "Cancella" che cancella il triangolo e permette all'utente di ricominciare da capo.

Parte 3 (DIFFICILE)

Si prenda il programma della parte 2. Si aggiungano delle caselle di testo che, una volta che l'utente ha creato il triangolo, dovranno visualizzare **l'ampiezza degli angoli** del triangolo. Qui serve un po' di trigonometria.
Le funzioni trigonometriche sono implementate come **metodi statici** della classe `Math`. Pertanto per calcolare le funzioni trigonometriche si possono chiamare i seguenti metodi:
`double Math.sin(double d)`: seno
`double Math.cos(double d)`: coseno
`double Math.tan(double d)`: tangente
`double Math.asin(double d)`: arcoseno
`double Math.acos(double d)`: arcocoseno
`double Math.atan(double d)`: arcotangente
Tutti i metodi prendono un argomento double e restituiscono un double. Seno, coseno e tangente prendono un angolo **in radianti**. Analogamente, arcoseno, arcocoseno e arcotangente restituiscono un angolo **in radianti**.
Nella classe `Math` è anche definita la costante `PI`, come membro dati statico, che corrisponde al pi greco. Il suo nome completo è pertanto `Math.PI`.

Per capire quale ampiezza corrisponde a quale angolo, sarà meglio dare un nome ai tre angoli, ad esempio A, B e C, visualizzando le scritte sul disegno del triangolo. Quando si sta disegnando in un componente con il metodo `paint`, è possibile disegnare una scritta con il metodo `drawString` della classe `Graphics2D`. Tale metodo prende come argomenti:

- la stringa da disegnare
- le coordinate (float o int) del punto più a sinistra della linea su cui deve essere posizionato il testo. Tale linea è quella linea immaginaria che si trova alla base delle lettere (come le righe di un quaderno).

Parte 4 (DIFFICILISSIMO PRATICAMENTE IMPOSSIBILE :-)

L'utente deve poter ruotare il triangolo. In particolare, dopo che egli ha selezionato i tre punti del triangolo, una nuova pressione del mouse lo fa entrare in modalità "rotazione". Se lascia andare il pulsante del mouse, la modalità "rotazione" termina.
Mentre il triangolo è in modalità "rotazione" deve essere visualizzato il centro del componente, rispetto al quale il triangolo ruoterà. Se l'utente muove il mouse (sempre tenendo premuto il pulsante, se no la modalità "rotazione" termina) si deve calcolare l'angolo di rotazione determinato dalla nuova posizione del mouse e ridisegnare il triangolo ruotato di tale angolo.
Ecco come calcolare l'angolo di rotazione:

- si indichi con O il centro del componente e con P il punto in cui il mouse è stato inizialmente premuto;
- si indichi con Q la nuova posizione del mouse;
- l'angolo di rotazione è $P\hat{O}Q$.