Progettazione, Algoritmi e Computabilità
Interfacce Grafiche in Java

Michele Beretta

michele.beretta@unibg.it

Programmazione a eventi

Programmazione a eventi

Eventi:

- Rappresentano le “parole” del linguaggio compreso dalle GUI
- Sono solitamente “azioni fisiche” compiute dall’uomo per comunicare con il calcolatore
- Possono essere elementari o complessi

La *programmazione orientata agli eventi* ha una grande importanza nell’attuale produzione software.

La GUI è sempre “in ascolto”, pronta a catturare gli eventi generati dall’utente (muove il mouse, clicca bottoni e icone, interagisce tramite tastiera, etc).

La struttura di un codice in grado di *reagire* a tutti questi eventi dipende dal modello di eventi supportato dal linguaggio di programmazione usato.

Programmazione a eventi

Le componenti delle GUI (ma non solo) raccolgono gli eventi e li passano a speciali oggetti detti **listener** il cui compito è rispondere all'evento verificato.

I listener vanno precedentemente registrati al componente che genera gli eventi. Questo meccanismo si chiama **delega degli eventi**.

Il passaggio degli eventi agli ascoltatori è completamente controllato dal programmatore. Lo stesso approccio è implementato in .NET ed in altri linguaggi.

Eventi: scelta architetturale

Nella programmazione ad eventi, la scelta architetturale di fondo è quella di **mantenere separato il codice GUI dall'applicazione**. Questo può essere ottenuto in 4 modalità:

1. Scegliere come listener per un componente *l'oggetto che più agevolmente può gestire i suoi eventi*.
   Solitamente significa scegliere come listener il `Container` di un gruppo di componenti.

Esempio: Un gruppo di bottoni è raccolto in un pannello, e i bottoni fanno cambiare colore al pannello quando cliccati. È naturale abilitare il pannello stesso come listener dei bottoni che contiene.

Eventi: scelta architetturale

2. Abilitare la classe stessa come listener dei suoi stessi eventi.
   Si ottiene un'*implementazione compatta* in cui disegno e gestione dell'interfaccia sono nella stessa classe con relativi **pro** e **contro**.

```
MyFrame implements WindowListener
```

Verranno aggiunti i metodi imposti dall'interfaccia all'interno di `MyFrame`. La classe può auto-registrarsi come listener:

```
addWindowListener(this)
```

Similmente in cascata per tutti i suoi componenti che saranno listener dei loro stessi eventi.

Eventi: scelta architetturale

3. Utilizzare una classe esterna, spesso privata e nello stesso file.
   È una soluzione molto utilizzata, e **consigliata**, quando la gestione degli eventi diventa particolarmente complessa. Essa permette di ottenere una separazione più netta tra disegno dell'interfaccia e gestione degli eventi.

```
MyWindowListener implements WindowListener { ... }
addWindowListener(new WindowListener())
```

Possibile contro: il passaggio di parametri può essere scomodo ed essere origine di una maggiore difficoltà di gestione.

Eventi: scelta architetturale

4. Dichiarare la classe che implementa il listener direttamente quando si crea l'oggetto.
   È una soluzione *molto rapida* e spesso utilizzata per listener poco complessi.

```
addWindowListener(new WindowListener() {
  // metodi
})
```

Interfaccia `ActionListener`

Lo stesso panel non può ereditare sia da `JPanel` sia da `ActionListener`. Perciò `JPanel` è una classe e `ActionListener` un'interfaccia.

```
MyPanel extends JPanel implements ActionListener
```

`ActionListener` definisce un solo metodo che dev'essere implementato.

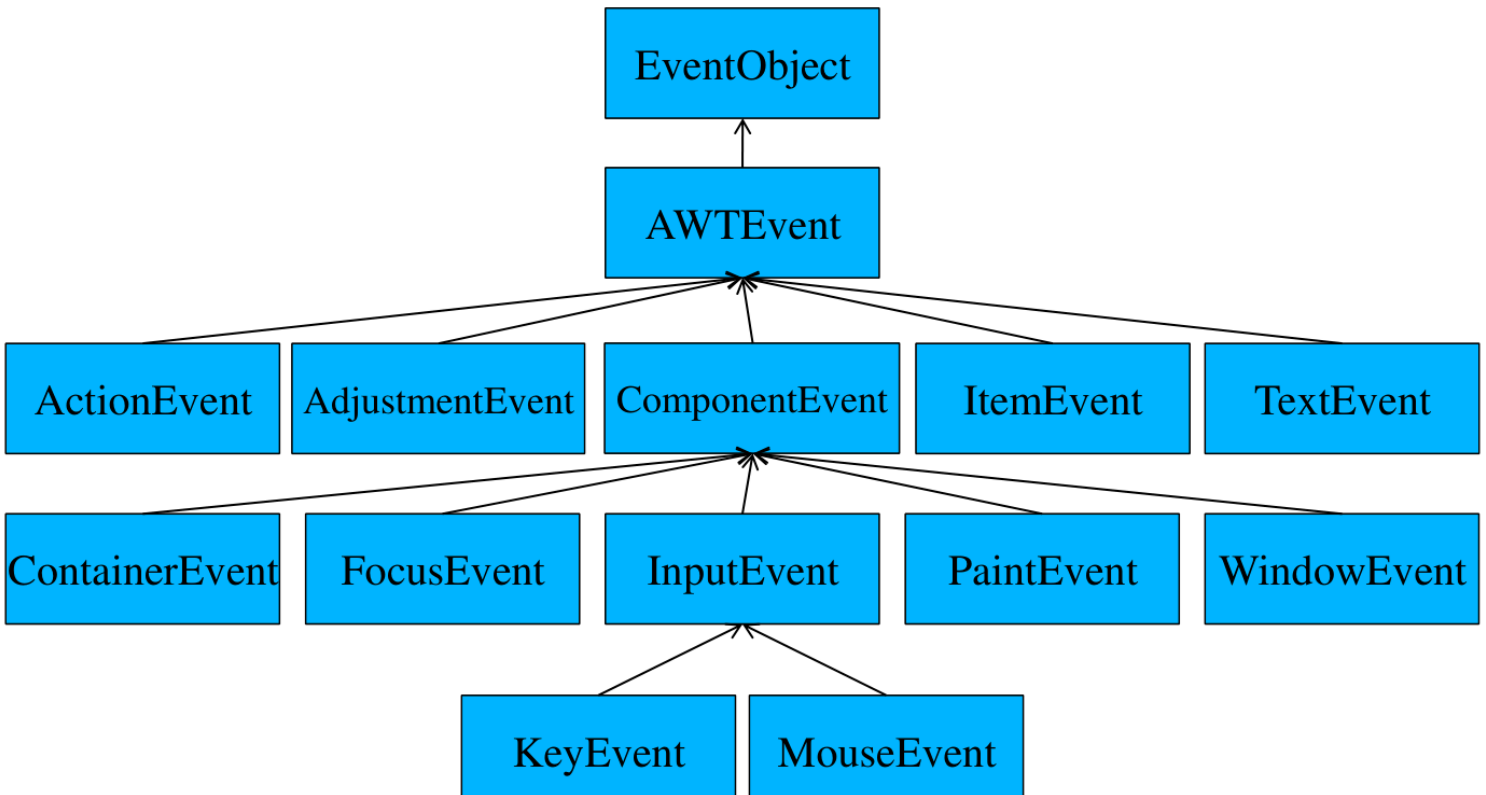
```
void actionPerformed(ActionEvent e)
```

Questo metodo racchiude tutte le istruzioni da eseguire quando il componente a cui il listener è registrato invia un evento di «azione» (e.g., bottone premuto, bottone rilasciato).

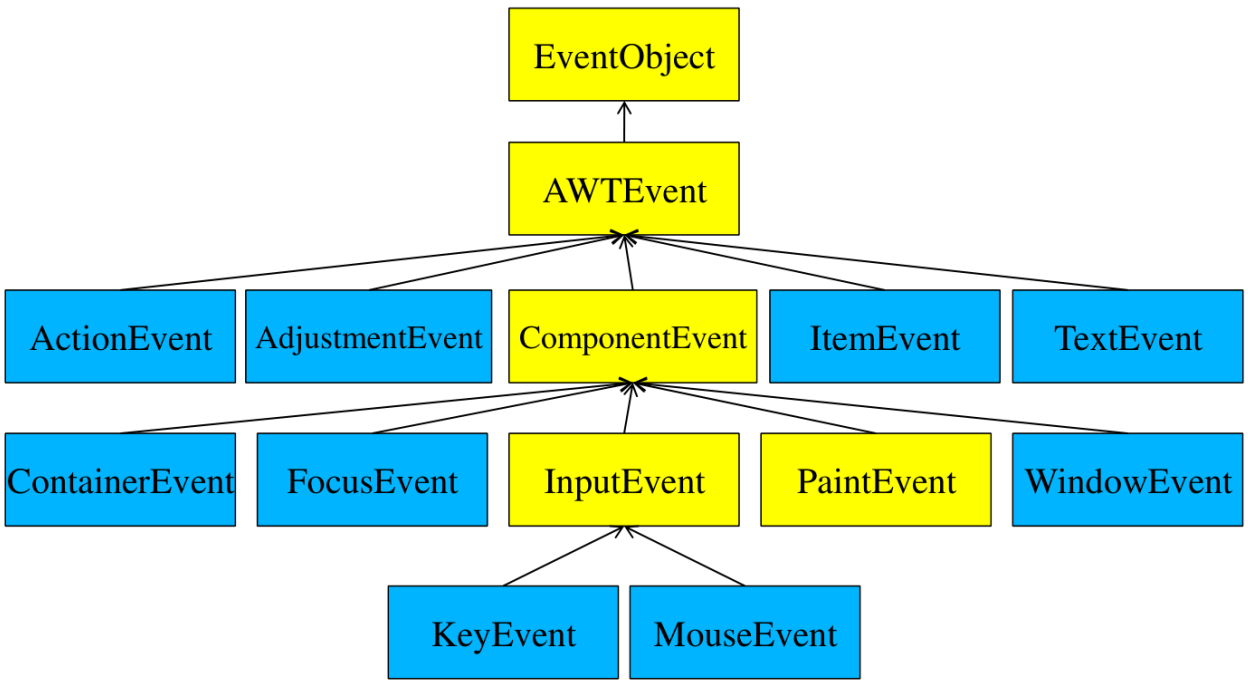
Nel caso di un listener per diversi oggetti, l'origine può essere ricavata tramite il metodo `Object getSource()`.

Eventi

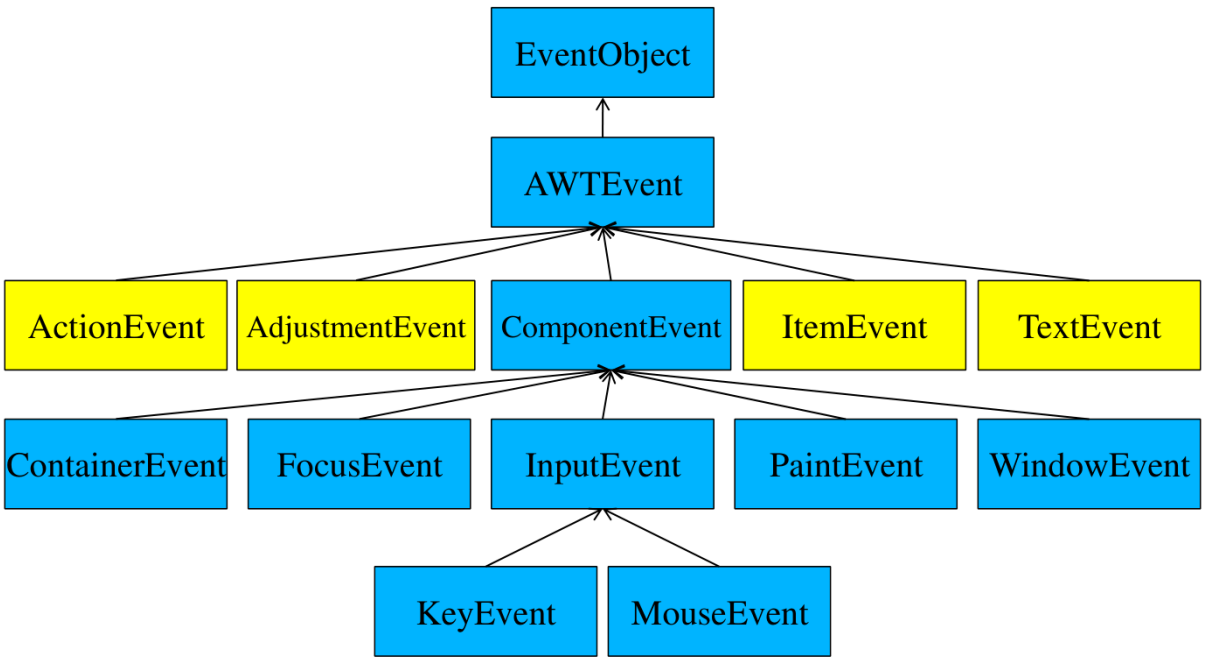
Gerarchia degli eventi AWT


EventObject
AWTEvent
ActionEvent
AdjustmentEvent
ComponentEvent
ItemEvent
TextEvent
ContainerEvent
FocusEvent
InputEvent
PaintEvent
WindowEvent
KeyEvent
MouseEvent

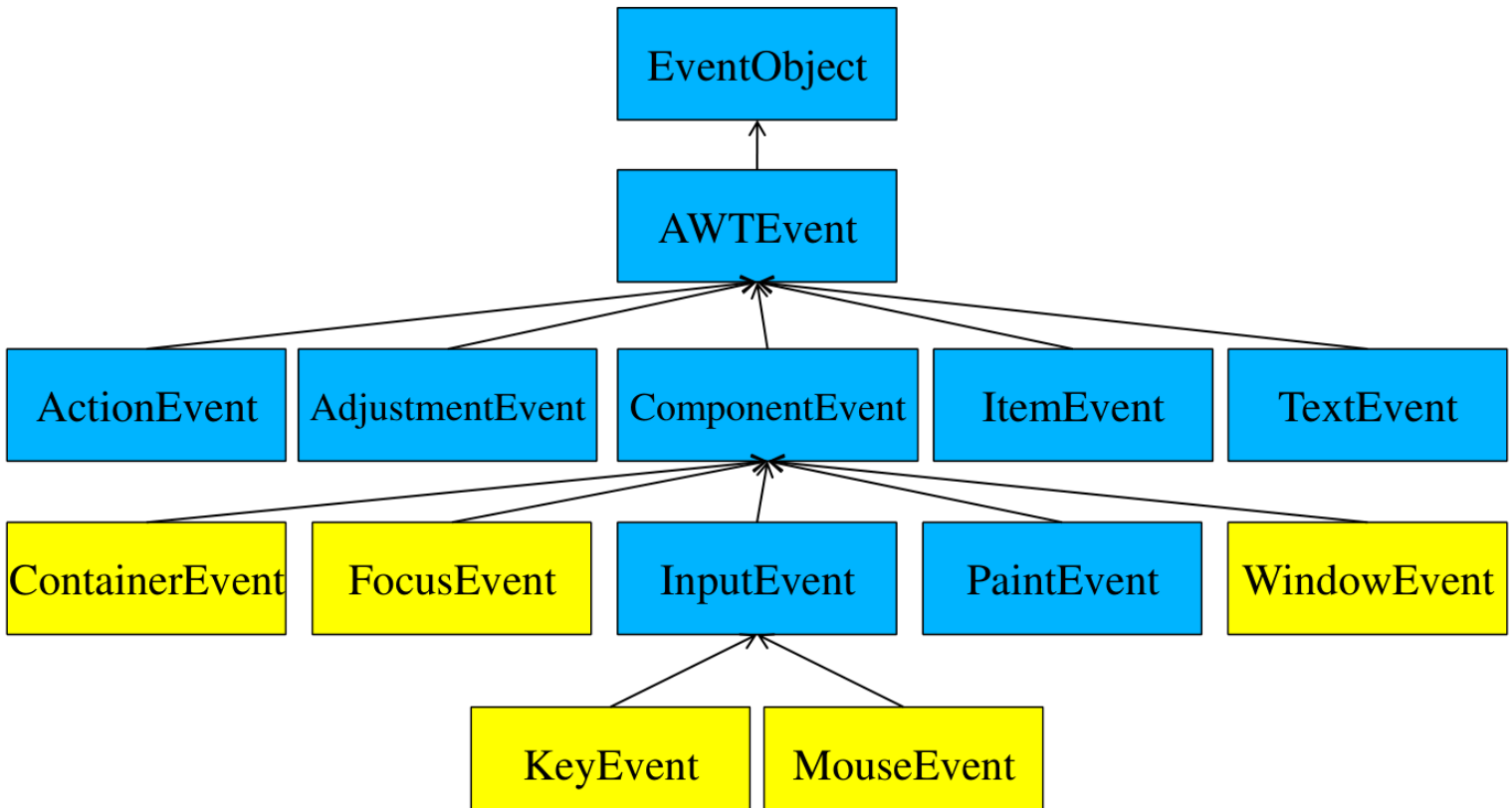
Gerarchia degli eventi AWT: classi astratte o usate raramente



Gerarchia degli eventi AWT: eventi semantici



Gerarchia degli eventi AWT: eventi elementari


EventObject
AWTEvent
ActionEvent
AdjustmentEvent
ComponentEvent
ItemEvent
TextEvent
ContainerEvent
FocusEvent
InputEvent
PaintEvent
WindowEvent
KeyEvent
MouseEvent

Eventi e listener

Si consiglia di usare la **documentazione** Java e un buon tool per il disegno delle interfacce (e.g., `WindowBuilder`), perché ricordare a memoria eventi, listener e metodi è inutile, e probabilmente impossibile.

Tra gli eventi di basso-medio livello è bene considerare quelli generati da **tastiera** e quelli generati dal **mouse** dato che rappresentano gli input di default usati dagli utenti in molte (*tutte*) le applicazioni grafiche.

Eventi da tastiera

Evento	Listener	Adapter
`KeyEvent`	`KeyListener`	`KeyAdapter`

I metodi da implementare sono i seguenti.

Metodo	Significato
`void keyPressed(KeyEvent e)`	Tasto premuto (low level)
`void keyReleased(KeyEvent e)`	Tasto rilasciato (low level)
`void keyTyped(KeyEvent e)`	Tasto digitato (high level)

Java assume una tastiera virtuale dove **ogni tasto ha un codice**. Ad esempio: `VKA`, ..., `VKZ`, `VK`, ..., `VK`, `VKPAGEDOWN`, `VKPAGEUP`, etc.

Eventi da tastiera

La classe `KeyEvent` fornisce alcuni metodi per individuare il tasto premuto:

```
int getKeyCode()
char getKeyChar()
```

Oltre ad un metodo per ottenere una descrizione testuale:

```
String getKeyText(int code)
```

Ci sono anche altri metodi che permettono di gestire l'uso dei modifier (shift, control, alt).

Eventi da mouse

Evento	Listener	Adapter
`MouseEvent`	`MouseListener`	`MouseAdapter`
	`MouseMotionListener`	`MouseMotionAdapter`
	`MouseInputListener`	`MouseInputAdapter`

I metodi principali sono i seguenti.

Metodo	Significato
`void mouseEntered(MouseEvent e)`	Mouse entra in un componente
`void mouseExited(MouseEvent e)`	Mouse esce da un componente
`void mousePressed(MouseEvent e)`	Bottone premuto
`void mouseReleased(MouseEvent e)`	Bottone rilasciato
`void mouseClicked(MouseEvent e)`	Bottone cliccato
`void mouseMoved(MouseEvent e)`	Movimento

Metodo	**Significato**
`void mouseDragged(MouseEvent e)`	Movimento (bottone premuto)

Eventi da mouse

La classe `MouseEvent` fornisce metodi utili per individuare alcune informazioni quali il numero di click, il bottone premuto, le coordinate dello schermo relative alla posizione del mouse.

La classe `Cursor` permette di gestire il cursore del mouse.

Vedere gli esempi sul codice per approfondire.

Pattern MVC

Model View Controller

MVC è uno dei più famosi pattern di progettazione, secondo il quale si separano:

- *Modello logico* (Model): rappresenta i dati dell'applicazione.
- *Aspetto visuale* (View): è la rappresentazione visuale (GUI).
- *Interazioni* con il resto dell'applicazione (Controller): cattura gli input della view e li traduce in cambiamenti di Model (e viceversa).

Model View Controller

MODEL

- Rappresenta i *dati* e le *regole* che ne governano accesso e modifica
- Spesso è un'approssimazione software di processi del mondo reale

VIEW

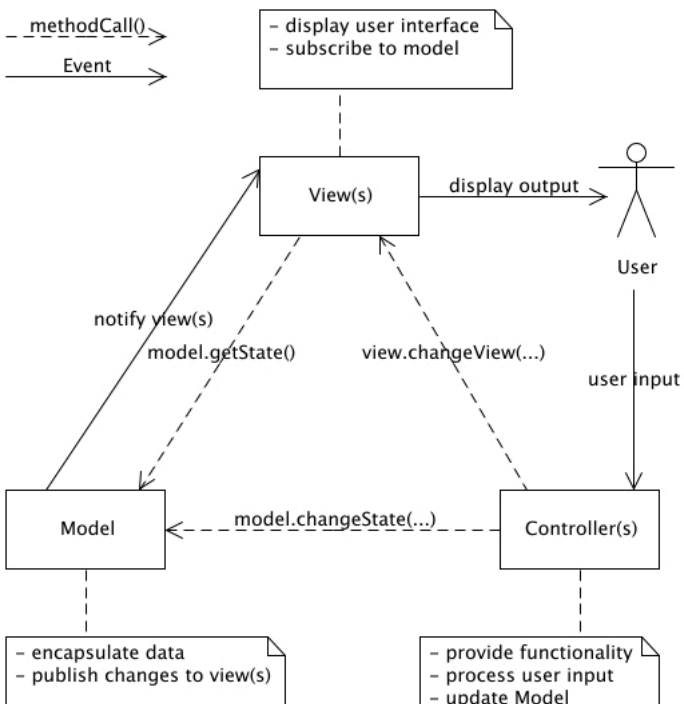
- *Aspetto grafico* dei dati del modello o di una parte di essi
- Mantiene aggiornati i dati grafici con i cambiamenti di modello
- Gli aggiornamenti possono avvenire in due modi:
  - *Push model*: la vista è notificata dal modello
  - *Pull model*: la vista è deve reperire le informazioni dal modello

CONTROLLER

- Trasforma le interazioni dell'utente in *azioni* che il modello deve compiere

- In alcuni contesti il controller seleziona viste alternative in base all’azione eseguita

MVC schema


methodCall()
Event
- display user interface
- subscribe to model
View(s)
display output
User
notify view(s)
model.getState()
view.changeView(...)
user input
Model
model.changeState(...)
Controller(s)
- encapsulate data
- publish changes to view(s)
- provide functionality
- process user input
- update Model

Punti di forza di MVC

1. La view non modifica il model direttamente
2. Il model non ha riferimenti diretti alla view
   - Modelli indipendenti dalla rappresentazione grafica
   - Più rappresentazioni grafiche dello stesso modello
3. La view non ha riferimenti diretti al controller
   - La view non dipende dal controller
   - Più controller per la stessa view

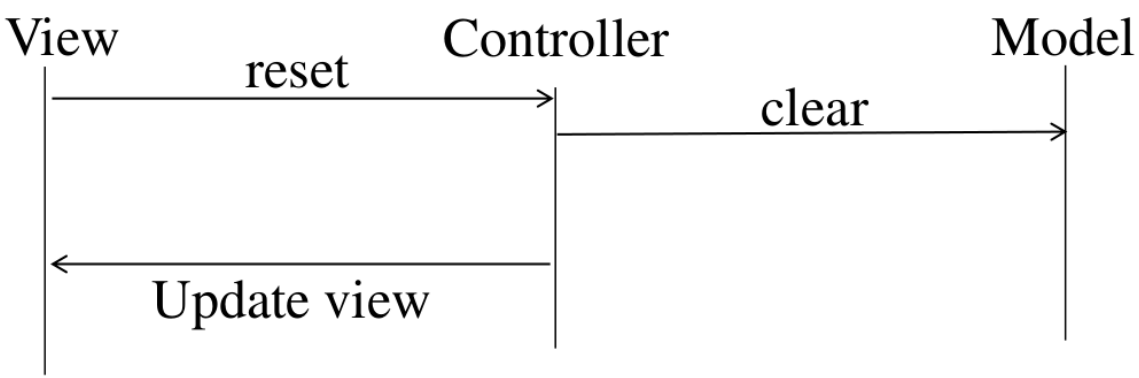
Lo stesso paradigma MVC si presta ad (almeno) due implementazioni

1. Il model viene cambiato solo dal controller
2. Il model subisce anche cambiamenti esterni (e.g., da thread)

Semplice esempio di MVC

Vediamo insieme una *calcolatrice* con due operazioni: `multiply` (per una costante arbitraria), e `clear`.

- *View*: bottoni e testo
- *Controller*: chiama il metodo `multiply` del model, aggiorna la view
- *Model*: contiene la costante e le due operazioni `multiply` e `clear`



Note sulla struttura

- Il controller ha un riferimento sia al model che alla view
- La view ha un riferimento al model
- Il model non ha riferimenti (completamente passivo)

Codice del controller

```
userInput = m_view.getUserInput()
m_model.multiply(userInput)
m_view.update()
```

I due oggetti utilizzati sono:

- `m_view` è il riferimento alla view
- `m_model` è il riferimento al modello

Il metodo update di `m_view` leggerà i dati dal modello (attraverso un riferimento di view a model).

Cosa fare in caso di aggiornamenti sul modello effettuati dall'esterno?

Questa è una situazione comune, in quanto il modello può essere la rappresentazione virtuale di un impianto reale, o di un database, etc.

Completamento esempio di MVC

In questo caso si utilizza il modello *push*:

- Il modello diventa *osservabile*, quando avviene un cambiamento notifica i listener ad esso registrati
- In Java è possibile usare `java.util.Observer` e `java.util.Observable`
- Il controller continua a non conoscere la view, si limita a fare broadcast di eventi a tutti i listener interessati

Il controller viene notificato a sua volta dalla view:

- La view fa broadcast di un evento in seguito ad un input dell'utente
- Il controller è un listener di quell'evento

- Il controller è l’unico ad avere un riferimento esplicito al modello attraverso il quale invoca i metodi per modificare fare eseguire operazioni al modello.

Esercizio MVC

Prendere il codice appena visto e realizzare l'esercizio presentato nel file *Esercizio bottone MVC*.

Swing e MVC

Alcuni componenti Swing sono pensati per implementare il pattern MVC.

La principale differenza rispetto a quanto visto è che il controller e la view sono nello *stesso elemento*

- La view è l'aspetto del componente Swing (bottoni, slider, etc.)
- Il controller sono gli `EventListener` o gli `ActionListener`

Questo, in generale, è un buon paradigma poiché permette di incentrare l'applicazione sui suoi dati piuttosto che sulla sua GUI.

I componenti Swing hanno un proprio modello contenente dati relativi all'oggetto (ad esempio le varie proprietà del componente).

I programmi (e i programmatori) devono solo **collegare i propri modelli ai modelli dei componenti Swing**.

Esempio MVC con `JButton`

- **Model**: stato del bottone (premuto, abilitato, etc.)
- **View**: l'aspetto del bottone
- **Controller**: le azioni specificate all'interno del metodo `actionPerformed`

Layout

Layout

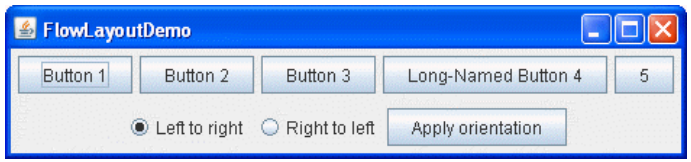
Ogni volta che disponiamo gli oggetti in uno spazio vuoto creiamo un layout

Per rispettare i requisiti di portabilità di Java, il layout dei componenti grafici deve essere robusto rispetto alle variazioni di sistema, schermo, dimensioni, etc..

Java gestisce la disposizione dei componenti dentro i Container attraverso speciali oggetti che si chiamano `LayoutManager`.

Il metodo `void setLayout(LayoutManager mgr)` permette di impostare il layout.

`FlowLayout`



È il layout predefinito, le componenti vengono aggiunte da sinistra a destra riempendo le righe disponibili.

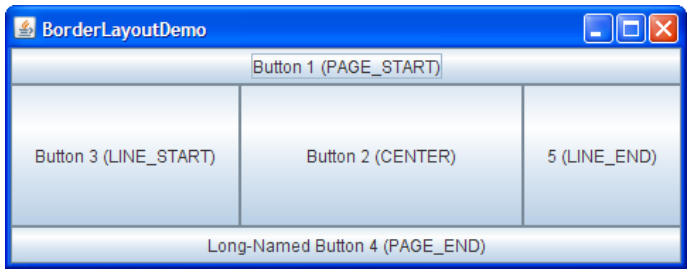
Le dimensioni sono determinate in base alle esigenze di ciò che si deve aggiungere.

I costruttori utilizzabili sono:

```
    FlowLayout()
    FlowLayout(int align)
    FlowLayout(int align, int hgap, int vgap)
```

Dove il parametro `align` può essere `FlowLayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`.

`BorderLayout`



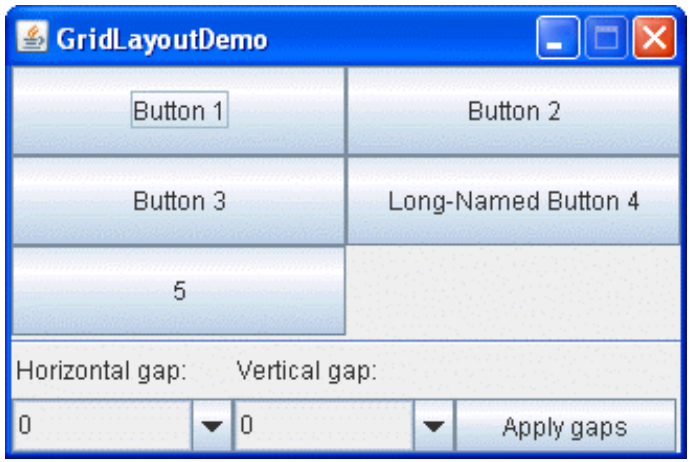
È il layout predefinito per i frame, in cui i componenti vengono aggiunti in *zone* del Container (North, South, East, West, Center).

Le dimensioni dei componenti vengono controllate ed alterate dal manager.

I costruttori utilizzabili sono:

```
BorderLayout()
BorderLayout(int hgap, int vgap)
```

`GridLayout`

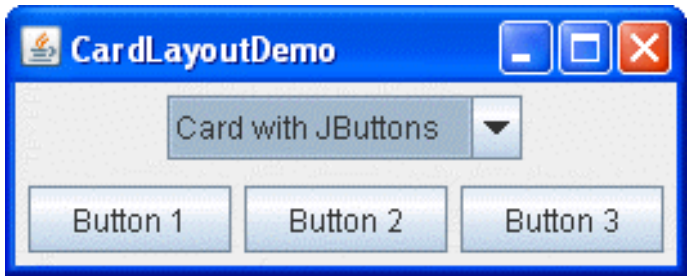


Il layout divide il Container in una griglia di celle rettangolari. I componenti vengono aggiunti cella per cella.

I costruttori utilizzabili sono:

```
GridLayout(int righe, int colonne)
GridLayout(int righe, in colonne, int hgap, int vgap)
```

`CardLayout` e `OverlayLayout`



`CardLayout`

- Ogni componente del Container viene trattato come una carta, solo una carta per volta può essere visualizzata
- Il flip (passaggio) tra i vari componenti è gestito dal programmatore via codice con metodi appositi

`OverlayLayout`

- Sovrappone i componenti l'uno sull'altro e li visualizza tutti

- Utile per pannelli grafici complessi ottenuti per sovrapposizione

Altri layout

`AbsoluteLayout`

- Si ottiene settando a `null` il layout
- Si posizionano manualmente i componenti nel container e si richiama il metodo `repaint`
- Utile se di dispone di un tool grafico

`GridBagLayout` e `SpringLayout`: particolarmente flessibili ma complessi.

Personalizzati: è possibile creare un layout da zero ma è un'operazione particolarmente complessa.

Componenti

JButton

Costruttori disponibili

- `JButton()`
- `JButton(String testo)`
- `JButton(Icon icona)`
- `JButton(String testo, Icon icona)`

Passi per la gestione corretta del bottone:

1. Creo un pannello
2. Creo il bottone con le sue proprietà e lo aggiungo al pannello che sarà il suo contenitore
3. Il pannello implementa l'interfaccia `ActionListener`
4. Implemento il metodo ereditato `actionPerformed` (azione da compiere quando il bottone è premuto)

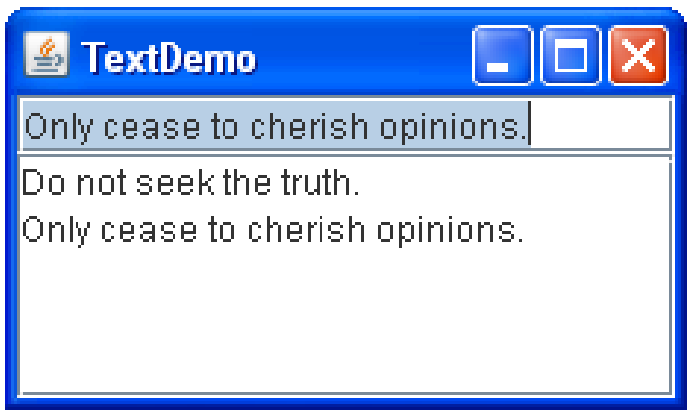
5. Registro il pannello presso il bottone come suo `ActionListener`

Caselle e aree di testo

Le classi principali per gestire le aree di testo sono le seguenti:

- `JTextComponent`: qui sono definiti molti dei metodi principali
- `JTextField`: caselle di testo standard
- `JPasswordField`: caselle di testo oscurate
- `JTextArea`: aree di testo con più righe

`JTextField`



Costruttori:

- `JTextField()`
- `JTextField(int numCol)`
- `JTextField(String str)`
- `JTextField(String str, int numCol)`

dove `str` è la stringa di inizializzazione della casella e `numCol` dà la misura indicativa del testo visibile. Caratteri aggiuntivi vengono conservati ma non visualizzati.

`JTextField` (cont.)

Metodi:

- `void setColumns(int newNumCol)`: cambia il numero di caratteri
- `String getText()`: restituisce il testo contenuto nella casella
- `void setText(String str)`: imposta il testo contenuto nella casella
- `void setEditable(boolean b)`: abilita/disabilita l'editabilità della casella

`JTextField` (cont.)

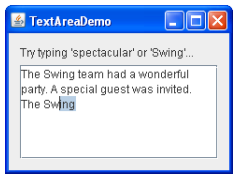
Eventi e Listener:

- `ActionEvent` (associato a `ActionListener`): lanciato quando si digita un testo nella casella di testo (compreso l'invio finale)
- `DocumentEvent` (associato a `DocumentListener`): si registra al documento della casella di testo (il suo contenuto)

Metodi dell'interfaccia `DocumentListener`:

- `void insertUpdate(DocumentEvent e)`: lanciato quando avviene un inserimento di testo
- `void removeUpdate(DocumentEvent e)`: lanciato quando avviene una rimozione di testo
- `void changedUpdate(DocumentEvent e)`: lanciato quando avviene un cambiamento di testo

JTextArea



2 Costruttori:

- `JTextArea()`
- `JTextArea(int rows, int cols)`
- `JTextArea(String s)`
- `JTextArea(String s, int rows, int cols)`

Metodi utili:

- `void setColumns(int cols)`
- `void setRows(int rows)`
- `void setText(String s)`
- `String getText()`
- `void append(String s)`
- `void insert(String s, int pos)`

`JTextArea`

Altri metodi utili:

- `void setLineWrap(boolean b)`: se b è vero la visualizzazione va a capo quando il testo raggiunge il margine destro di visualizzazione a ogni riga
- `void setWrapStyleWord(boolean b)`: se b è vero si va a capo all'inizio di parola, altrimenti il testo viene spezzato dove capita

`JTextArea` supporta cut, copy e paste

`JScrollPane`

Se il testo è troppo lungo occorre scorrere all'interno dell'area di testo.

Per poterlo fare bisogna inserire il contenuto in un `JScrollPane`, in cui le barre di scorrimento si attivano quando il contenuto eccede l'area a disposizione.

Questa soluzione non si applica solo alle aree di testo.

`JLabel`

Utile per etichettare porzioni di interfaccia. Costruttori:

- `JLabel(String text)`
- `JLabel(Icon icona)`
- `JLabel(String text, int align)`
- `JLabel(String text, Icon icona, int align)`

Dove `align` può valere `SwingConstants.LEFT`, `SwingConstants.RIGHT`, `SwingConstants.CENTER`.

Metodi utili:

- `void setText(String str)`: imposta un nuovo testo per l'etichetta
- `void setIcon(Icon icona)`: imposta una nuova icona per l'etichetta
- `void setFont(Font font)`: imposta un nuovo font per l'etichetta

Caselle di scelta

Spesso l'input testuale non è il metodo più adatto per interagire con l'utente. Ci sono occasioni in cui è preferibile offrire agli utenti una serie finita di scelte.

Questo risultato si può ottenere attraverso diversi componenti grafici

- `JCheckBox`, `JToggleButton`, `JRadioButton`: Scelta multipla
- `JComboBox`, `Jlist`: Scelta da una lista
- `JSpinner`, `JSlider`: Impostazione di un valore

Scelta non mutualmente esclusiva

Per le opzioni NON mutuamente esclusive, si usano `JCheckBox` e `JToggleButton`.
Costruttori:

- `JCheckBox(String etichetta)`
- `JCheckBox(String label, boolean stato)`
- `JCheckBox(String label, Icon icona)`

Metodi utili:

- `boolean isSelected()`, `void setSelected(boolean b)`: restituisce/imposta lo stato della casella

Eventi:

- `ActionEvent`, generato ad ogni cambio di stato

Scelte mutualmente esclusive

Nel caso di scelte mutualmente esclusive, un'opzione esclude la scelta di altre opzioni (ad esempio il colore del testo).

In questi casi sono preferibili i *gruppi* di bottoni `JRadioButton` e `ButtonGroup`.

Costruttori:

- `JRadioButton(String label)`
- `JRadioButton(String label, boolean stato)`
- `JRadioButton(String label, Icon icona)`

Metodi utili:

- `boolean isSelected()`, `void setSelected(boolean b)`: restituisce/imposta lo stato della casella.

Eventi:

- `ActionEvent`, generato ad ogni cambio di stato

`JRadioButton` all'interno di `ButtonGroup`

I bottoni costruiti vanno aggiunti **logicamente** ad un gruppo in modo da ottenere il funzionamento mutuamente **esclusivo**.

Costruttori:

- `ButtonGroup()`

Metodi utili:

- `void add(JRadioButton b)`: aggiunge il radiobutton al gruppo

Elenchi

Se le opzioni a disposizione sono molte (ad esempio la scelta di un font) i pulsanti di scelta non sono una soluzione valida dato che occupano troppo spazio.

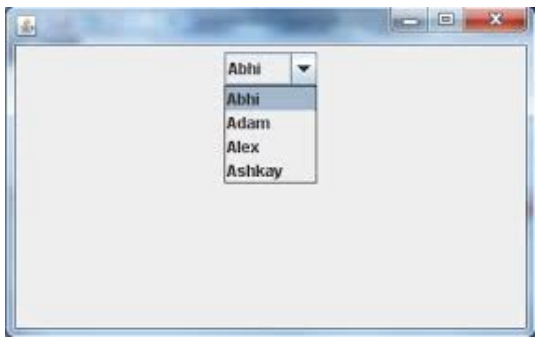
Un altro metodo per operare una scelta è presentare un lungo elenco di opzioni che viene presentato con un elenco a tendina o fisso.

Questo viene presentato in Swing con due tipi di componenti:

- `JComboBox`
- `JList`

`JComboBox`

Sono liste a tendina con possibilità di editare l'input (opzionale).



- `ActionEvent`, generato ad ogni cambio di selezione

Costruttori:

- `JComboBox()`
- `JComboBox(Object[] items)`

Eventi:

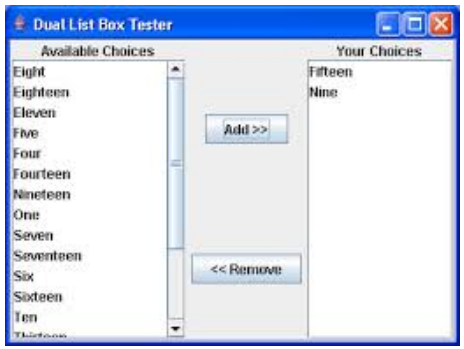
Metodi utili:

- `void addItem(Object obj)`, `void removeItem(Object obj)`: aggiunge/elimina una voce
- `Object getSelectedItem()`: restituisce l'item selezionato
- `int getSelectedIndex()`: restituisce l'indice dell'item selezionato

`JList`

Costruttori:

- `JList()`
- `JList(Object[] items)`



Metodi utili:

- `Object getSelectedValue()`, `Object[] getSelectedValues()`: restituisce gli item selezionati
- `int getSelectedIndex()`, `int[] getSelectedIndices()`: restituisce l'indice degli item selezionati
- `void setSelectionMode(int mode)`: imposta la modalità di selezione

`JList`

Eventi e listener:

- `ListSelectionEvent`, generato quando si cambia la selezione
- `ListSelectionListener`

L'interfaccia del listener richiede solo un metodo:

- `void valueChanged(ListSelectionEvent e)`

`JList`

`JList` mostra chiaramente l'approccio MVC.

È possibile modificare come gli elementi sono disposti nella lista (lavorando con il model della lista) in una delle due seguenti modalità:

- Implementare l'interfaccia `ListModel`
- Estendere la classe `AbstractListModel`

È possibile modificare il sistema di visualizzazione degli elementi (lavorando con il renderer della lista) in una delle due seguenti modalità:

- Implementare l'interfaccia `ListCellRenderer`
- Estendere la classe `DefaultListCellRenderer`

Bordi

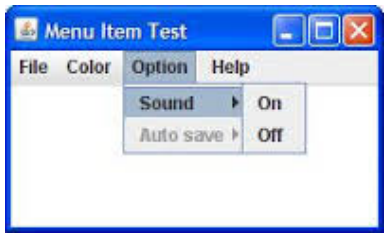
Java permette di aggiungere ad ogni componente Swing un bordo.

Solitamente conviene associare bordi solo ai pannelli. Per farlo si crea un border con i metodi della classi astratta `BorderFactory` e la si associa all'oggetto desiderato.

Esistono diversi tipi di bordo, con e senza titoli.

Vedere l'esempio *BorderFrame*.

Menù



Sono componenti che si aggiungono, solitamente, alla barra superiore dei frame (file, inserisci, modifica, etc).

Ne esistono anche di speciali: *a comparsa* (pop-up), *floating* (tool-bar).

Alcuni componenti Swing per la gestione dei menù sono `JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem`, `JRadioButtonMenuItem`

Tutti questi componenti generano eventi di tipo `ActionEvent`, con un comportamento identico a quello di `JButton`.

Si possono aggiungere menù ai menù ottenendo dei sottomenù.

`JDialog`

Usato per scelta di opzioni e input di dati

È sempre collegato ad un `JFrame` o ad un altro `JDialog` genitore, collegarlo ad altri componenti porta a problemi di gestione delle finestre.

Costruttori:

- `JDialog()`
- `JDialog(Dialog owner)`
- `JDialog(Frame owner)`
- `JDialog(Dialog owner, boolean modal)`
- `JDialog(Frame owner, boolean modal)`

- JDialog(Dialog owner, String title)
- JDialog(Frame owner, String title)
- JDialog(Dialog owner, String title, boolean modal)
- JDialog(Frame owner, String title, boolean modal)

`JDialog` modali

I JDialog creati in modalità *modale* provocano l'interruzione dell'esecuzione del programma fintanto che la finestra di dialogo non sarà chiusa.

Questa modalità è attivata impostando a `true` il corrispondente flag del costruttore (`modal`).

Il resto della documentazione è identico a quello dei `JFrame`.

`JFileChooser`

Le GUI moderne offrono la possibilità di selezionare un file da aprire o salvare tramite apposite finestre.

Java offre questa possibilità con la classe `JFileChooser`, i cui metodi principali sono:

- `File getSelectedFile()`: restituisce il file selezionato
- `void setCurrentDirectory(File dir)`: imposta la directory corrente
- `void setDialogTitle(String title)`: imposta la barra del titolo
- `int showOpenDialog(Component parent)`: visualizza una finestra per aprire file
- `int showSaveDialog(Component parent)`: visualizza una finestra per salvare file

`JFileChooser`

La classe `JFileChooser` permette di personalizzare i tipi di file accettati e l'icona da visualizzare per i vari tipi di file:

- `void setAcceptAllFileFilterUsed(boolean b)`: imposta se deve essere visualizzata la voce «Tutti i file» nella lista di file accettati
- `void setFileFilter(FileFilter filter)`: imposta l'accettazione di un tipo di file
- `void setFileView(FileView fileView)`: imposta l'icona da visualizzare per un tipo di file

`JFileChooser` – `FileFilter`

`FileFilter` è una classe astratta indicante un «filtro» per un tipo di file. Un oggetto figlio di `FileFilter` si imposta nel `JFileChooser` per eliminare i file indesiderati dalla finestra di dialogo.

Estendere un `FileFilter` significa implementare i metodi:

- `abstract boolean accept(File f)`: restituisce se il file deve essere accettato
- `abstract String getDescription()`: restituisce la descrizione del filtro

`JFileChooser` – `FileView`

`FileView` è una classe astratta indicante l'aspetto «visivo» per un tipo di file.

Estendere un `FileView` significa implementare i seguenti metodi

- `Icon getIcon(File f)`
- `String getDescription(File f)`
- `String getName(File f)`
- `String getTypeDescription(File f)`
- `Boolean isTraversable(File f)`

`JFileChooser` – Accessory

Infine è possibile aggiungere a `JFileChooser` un intero `JComponent` per permettere visualizzazioni complesse come anteprime, informazioni sul file, etc.

Il metodo per fare ciò è:

- `void setAccessory(JComponent newAccessory)`

`JColorChooser`

Permette di selezionare un colore attraverso tre diversi pannelli di scelta:

- `Colori campione`
- `Colori nel modello HSB (Hue, Saturation, Brightness)`
- `Colori nel modello RGB (Red, Green, Blue)`

Costruttori:

- `JColorChooser()`
- `JColorChooser(Color initialColor)`
- `JColorChooser(ColorSelectionModel model)`

Metodi:

- `Color getColor()`
- `void setColor(Color color)`

- `static Color showDialog(Component c, String title, Color initialC)`: visualizza una finestra per la selezione del colore.