Java

java beans e programmazione visuale

G. Prencipe
prencipe@di.unipi.it

Cos'è un JavaBean

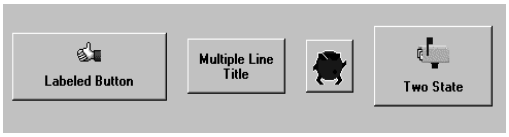
- Un **JavaBean** (o semplicemente Bean) è un ***componente software riusabile***
  - Analogia con i componenti elettronici
- I Bean possono essere ***manipolati visualmente*** all'interno di appositi strumenti di programmazione avanzati
- Si possono realizzare semplici applicazioni senza scrivere nemmeno una riga di codice!

Componenti software

- Il concetto di **componente** è molto diffuso in altre industrie:
  - Elettronica: valvole, condensatori, resistenze, ...
  - Automobilistica: "ricambi non originali"
- Vantaggi dei componenti
  - Facilità di assemblaggio
  - Interfaccia uniforme
  - Versioni alternative (con la stessa interfaccia) fra cui scegliere

Componenti software

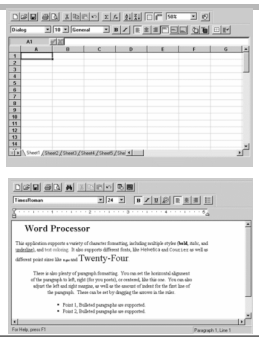
- Componenti base di interfaccia utente
  - Classici: pulsanti, campi di testo, menu, ecc.
  - Tradizionalmente disponibili come *librerie* o *toolkit* di sistema
  - Usati come Bean, possono essere più facilmente integrati



Componenti software

- Componenti avanzati di interfaccia utente
  - Servono per fornire funzioni di manipolazione fuori standard
  - Esempio: un calendario per selezionare una data
  - Possono essere usati tali e quali in tante applicazioni: vantaggio!

Componenti software



- Intere micro-applicazioni
  - Serve un foglio elettronico? un programma di videoscrittura?
  - Usiamo un Bean più "ciccioso" degli altri...
  - L'interfaccia di base è comunque la stessa
  - L'offerta è vasta:
    - programmi di grafica
    - display di mappe per i GIS

Componenti software

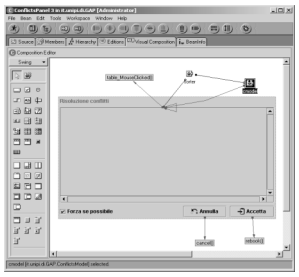
- Componenti ***non-visuali***
  - Non hanno un'interfaccia grafica visibile all'utente
  - Però forniscono servizi ad altri Bean:
    - Bean che fanno calcoli
    - Bean che si connettono in rete
    - Bean che accedono a basi di dati
    - Bean che memorizzano e gestiscono uno stato
  - L'interfaccia per il programmatore è la stessa di quella dei Bean visuali

Componenti software

- Chiunque può costruire nuovi Bean!
  - Partendo da zero
  - Creando sottoclassi di Bean esistenti
  - Trasformando vecchie classi in Bean
- I nuovi Bean possono essere usati come tutti gli altri
- Gli strumenti di programmazione non sono limitati a un insieme di Bean predefinito

Assemblaggio

- I Bean vengono ***assemblati*** con degli strumenti adeguati
- Il risultato è un'applicazione basata sui Bean
- Si realizzano applicazioni "tirando i fili" anziché scrivendo codice



Struttura di un Bean

- Un Bean ***è una classe Java***...
- ... i cui metodi hanno dei nomi particolari
  - La denominazione standard consente agli strumenti di assemblaggio di scoprire le caratteristiche di Bean "sconosciuti"
- Un Bean può essere usato in due contesti:
  - A **tempo di progettazione** (*design time*)
  - A **tempo d'esecuzione** (*run time*)

Caratteristiche di un Bean

- ***Proprietà***: analoghe agli attributi di una classe
  - Per una certa proprietà **xxx** si creano (tipicamente) due metodi
  - **setXxx()** e **getXxx()**
  - Per una proprietà booleana è possibile alternativamente utilizzare un metodo chiamato **isXxx()** per testare se la proprietà è vera o no
  - L'accesso avviene *esclusivamente* tramite metodi **get...()** e **set...()**

Caratteristiche di un Bean

- ***Eventi***: un Bean può generare eventi, che poi vengono propagati e causano delle reazioni
  - Non hanno un corrispondente nativo nel linguaggio
  - Simili al modello dei listener delle Swing
    - **addBounceListener(BounceListener)** e **removeBounceListener(BounceListener)** per gestire l'evento **BounceEvent**
- ***Metodi***: solita roba, corrispondono ai metodi della classe
  - Sono gli altri metodi nel bean (che non seguono le convenzioni dette sopra) e sono semplicemente **public**

Esempio

```
public class Frog {
    private int jumps;
    private boolean jmpr;
    private ….;

    public int getJumps() {return jumps;}
    public void setJumps(int newJumps) {jumps=newJumps;}
    public boolean isJumper(){return jmpr;}
    public void setJumper(boolean j){impr=j;}
    public void addActionListener(ActionListener l){….}

    // Metodo ordinario
    public void gracchia(){System.out.println("Gracchia!!";}
}
```

Proprietà

- Definiscono lo stato corrente di un Bean
- Per i Bean visuali, le proprietà influenzano spesso la rappresentazione grafica
- Poiché la modifica di una proprietà avviene attraverso metodi, è possibile far *reagire* il Bean a modifiche alle sue proprietà
  - In particolare, modifiche alle proprietà possono generare eventi (vedremo fra breve)

Esempio:

Proprietà JButton

	Proprietà	Descrizione
Grafica	x, y, width, height	posizione e dimensione del pulsante
	border, borderPainted	quale bordo disegnare intorno al pulsante
	font	fonte da usare per l'etichetta
	icon, disabledIcon	icona da usare per il pulsante attivo o disabilitato
Stato	enabled	il pulsante è abilitato?
	selected	il pulsante è premuto?
	showing	il pulsante è visibile?
	dropTarget	chi avvisare se viene fatto il drag&drop su questo pulsante
⋮	parent	il contenitore che contiene il pulsante

Esempio:

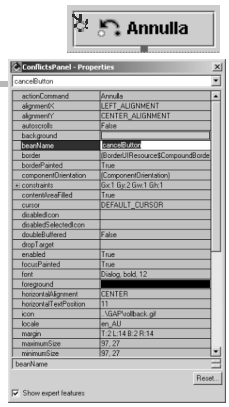
Proprietà JButton


```
void setFont(Font f)
Font getFont()
```

```
void setSelected(boolean b)
boolean isSelected()
```

	Proprietà	Descrizione
Grafica	x, y,	e dimensione del pulsante
	bord	ordo disegnare intorno al pulsante
	font	fonte da usare per l'etichetta
	icon, disa	lsante attivo o
Stato	enabled	
	selected	il pulsante è premuto?
	showing	il pulsante è visibile?
	dropTarget	chi avvisare se viene fatto il drag&drop su questo pulsante
⋮	parent	il contenitore che contiene il pulsante

Property Editor



- A ***design time***, gli strumenti di sviluppo (nell'esempio, *VisualAge* della *IBM*) forniscono un'interfaccia grafica per impostare le proprietà dei Bean
- A **run time**, il programma può modificare dinamicamente le proprietà chiamando i metodi **set...()** e ispezionarle chiamando **get...()** o **is...()**

Eventi

- Un Bean può generare eventi in risposta a stimoli esterni
  - Azioni dell'utente
  - Cambiamenti di valore delle proprietà
  - Timer, eventi di rete, ...
  - ... e qualunque altra cosa
- Altri Bean che sono interessati a sapere quando si verificano questi eventi possono **registrarsi** presso il Bean che li genera
- Quando si verificherà l'evento, verrà chiamato un particolare metodo dell'ascoltatore (eventualmente passando dei parametri)

Esempio:

Eventi JButton

	Evento	Descrizione
Azioni utente	actionEvents	qualunque azione dell'utente
	mouseEvents	qualunque operazione col mouse
	focusEvents	qualunque cambiamento di focus
	keyEvents	qualunque pressione di tasti
Proprietà	enabled	lo stato di abilitazione è cambiato
	selected	lo stato di selezione è cambiato
	showing	lo stato di visibilità è cambiatio
	text	l'etichetta del pulsante è cambiata
⋮	vetoableChangeEvents	qualcuno ha posto il veto su un cambiamento di valore di una proprietà

Esempio:

Eventi JButton

	Evento	Descrizione
Azioni utente	actionEvents	qualunque azione dell'utente
	mouseEvents	qualunque operazione col mouse
	focusEvents	qualunque cambiamento di focus
	keyEvents	qualunque pressione di tasti
Proprietà	enabled	lo stato di abilitazione è cambiato
	selected	lo stato di selezione è cambiato
	showing	lo stato di visibilità è cambiatio
	text	l'etichetta del pulsante è cambiata
⋮	vetoableChangeEvents	qualcuno ha posto il veto su un cambiamento di valore di una proprietà

Proprietà ed Eventi

- Le proprietà possono essere...
  - **semplici:** hanno un valore che può essere
    - Solo letto (**read only**) – es. *showing* di JButton
    - Solo scritto (**write only**) – estremamente raro
    - Letto e scritto – caso più comune
  - **bounded:** se il loro valore cambia, viene generato un evento a cui altri possono reagire
  - **constrained**: un ascoltatore può opporsi (porre il veto) a un cambiamento di valore

Proprietà ed Eventi

- Gli eventi sono
  - rappresentati da **classi**, che formano gerarchie di ereditarietà (come per le eccezioni)
  - Identificati da un **nome**, che descrive il tipo di evento
  - Accompagnati da **parametri** (attributi delle classi che li rappresentano) che ne specificano i dettagli
    - per esempio: gli eventi **MouseEvent** hanno come attributi la posizione (x,y) e il numero di click consecutivi (per identificare il doppio click)

Ok, ma cosa c'è dietro?

- Nell'infrastruttura dei JavaBeans non c'è niente di "magico", tutto è fatto con mezzi comuni
- I Bean sono normalissime classi, e possono essere scritti o usati anche con TextPad
- In particolare, il legame fra proprietà ed eventi è esplicitamente visibile nel codice sorgente

Codice per le proprietà

- Esempio: una proprietà *BaseDate* di tipo Date
- *Leggibile*, *scrivibile*, e *bounded* (genera eventi)

```
private Date fieldBaseDate = new Date();

public Date getBaseDate() {
    return fieldBaseDate;
  }

public void setBaseDate(Date baseDate) {
    Date oldValue = fieldBaseDate;
    fieldBaseDate = baseDate;
    firePropertyChange("baseDate", oldValue, baseDate);
  }
```

PropertyChangeEvents

- Cosa fa
  `firePropertyChange("baseDate",oldValue,baseDate)`
- Se il vecchio valore è uguale al nuovo (ovvero, se il valore della proprietà non è cambiato), non fa nulla
- Altrimenti, genera un evento di tipo **PropertyChange** (del gruppo dei **ChangeEvents**), passando come parametri il nome della proprietà "baseDate", il vecchio e il nuovo valore

Come si ricevono gli eventi

- Chi vuole essere informato di un evento si **registra** presso chi genera l'evento
- Il metodo da chiamare per registrarsi è **`add`***TipoEvento*`Listener(`***TipoEvento***`Listener l)`**
- Quando si verifica un evento, chi lo genera chiama il listener di tutti gli ascoltatori registrati
- Quando non si è più interessati, ci si **de-registra** chiamando il metodo **`remove`***TipoEvento*`Listener(`***TipoEvento***`Listener l)`**

Come si ricevono gli eventi

- L'interfaccia *TipoEvento*`Listener` definisce i metodi da chiamare
- Per `PropertyChangeListener`:
  - `public void propertyChange(PropertyChangeEvent evt)`
- Per `VetoableChangeListener`:
  - `public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException`

Generazione degli eventi
(codice leggermente semplificato)

```
public void firePropertyChange(String propertyName,
                               Object oldValue, Object newValue)
{
  if (oldValue != null && newValue != null
                       && oldValue.equals(newValue))
    return;

  PropertyChangeEvent evt =
            new PropertyChangeEvent(source, propertyName,
                                    oldValue, newValue);

  if (listeners != null) {
    for (int i = 0; i < listeners.size(); i++) {
      PropertyChangeListener listener =
                (PropertyChangeListener)listeners.elementAt(i);
      listener.propertyChange(evt);
    }
  }
}
```

Ricezione degli eventi
(codice leggermente semplificato)

```
public void propertyChange(PropertyChangeEvent evt)
{
  if (evt.getSource() == CalendarPage.this &&
              (evt.getPropertyName().equals("baseDate")))
  /* è cambiata baseDate */

  if (evt.getSource() == CalendarPage.this.getDayButton() &&
              (evt.getPropertyName().equals("background")))
  /* è cambiata la proprietà background di DayButton */

  /* ... e così via ... */
}
```

- Ma per fortuna, non è davvero necessario scrivere codice di questo tipo
- Gli strumenti di programmazione lo fanno per noi!

Programmazione visuale

- Abbiamo visto come una delle caratteristiche migliori di Java è legata al riutilizzo del codice
  - La parte di codice più riutilizzabile è chiaramente la classe
- Per queste caratteristiche, sarebbe auspicabile avere un modo per costruire applicazioni in modo automatico
  - Si definiscono le componenti dell'applicazioni, le si selezionano e assemblano come se si stessero assemblando chip su una scheda

Programmazione visuale

- Un modo rapido per sviluppare applicazioni
- Si piazzano componenti visuali e non visuali su una superficie di lavoro
- Si connettono proprietà, eventi e metodi dei vari componenti secondo la logica dell'applicazione
- Si scrive quel po' di codice extra che serve, e lo si connette ai componenti
- *I connettori sono essi stessi componenti!*
- Un esempio visto è il *Visual Editor* di *Eclipse*

Programmazione visuale

- La *programmazione visuale* (*Visual Programming*) ebbe enorme successo con il *Visual Basic*
  - Le componenti erano rappresentate graficamente
  - In effetti, uno dei settori in cui questo modo di costruire applicazioni è utilizzato è legato alla costruzione di interfacce grafiche
    - Come abbiamo visto con VE
- In generale, sistemare le componenti su un piano di lavoro non è sufficiente per completare un programma

Programmazione visuale

- Spesso è necessario modificare le caratteristiche delle componenti (colori, testo, database a cui la componente è collegata)
- Le caratteristiche che possono essere modificate a tempo di design sono le *proprietà*
- È possibile così manipolare le proprietà delle componenti all'interno del *costruttore automatico di applicazioni* (*application builder tool*)

Programmazione visuale

- Un oggetto (componente) è comunque qualcosa in più di un insieme di proprietà
  - È costituito anche da comportamenti
- I comportamenti di una componente sono rappresentati dagli eventi
  - Definiscono cosa può accadere alla componente durante l'esecuzione
  - Si decide cosa fare in seguito a un evento legando ad esso un pezzo di codice

Programmazione visuale e JavaBeans

- Il costruttore di applicazioni interroga dinamicamente le componenti e ricava le proprietà e gli eventi che la componente può gestire
- Ad esempio, i beans sono utilizzati dal *Visual Editor* di *Eclipse* per costruire le interfacce

Java

java beans e programmazione visuale

fine

Programmazione visuale

- Un metodo di programmazione innovativo
- Per sua natura, si presta ad essere spiegato *in vivo* su un esempio
- Scriviamo – da zero – un client **grafico** per il TimeServer che abbiamo sviluppato nella prima lezione!
- Metteremo insieme: JavaBeans, VisualAge e programmazione di rete con i socket