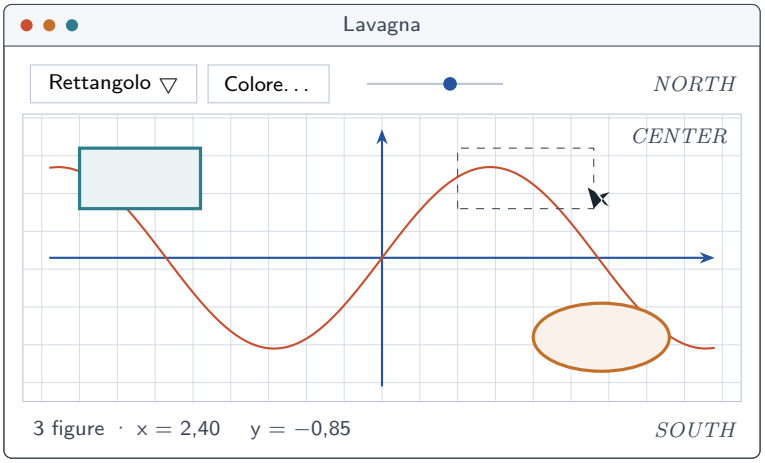
PAJC · Programmazione Avanzata in Java e C · 2026/27

Dispensa 5

Interfacce grafiche con Swing

Componenti e layout, eventi, finestre di dialogo, grafica con le primitive, Graphics2D, coordinate, mouse e animazioni



Componenti che si compongono, un pannello che si disegna.

prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

massimiliano.redolfi@unibs.it

Versione 0.1.4

Quest'opera è distribuita con licenza Creative Commons "Attribuzione – Non commerciale – Condividi allo stesso modo 4.0 Internazionale".

Indice

1 Introduzione: una lavagna che si rompe **1**
- 1.1 Il problema . . . 1
- 1.2 L'immagine guida: evento, stato, disegno . . . 4
- 1.3 Che cosa serve . . . 5
- 1.4 La strada che percorreremo . . . 6
- Esercizi . . . 6
- Risposte . . . 7

2 La finestra e i suoi componenti **8**
- 2.1 Finestre, contenitori, componenti . . . 8
- 2.2 I componenti di tutti i giorni . . . 10
- 2.3 Chi decide dove: i layout manager . . . 13
- 2.4 Comporre con i pannelli . . . 15
- 2.5 Il layout assoluto e il designer . . . 17
- 2.6 L'esempio guida: la Lavagna composta . . . 18
- Esercizi . . . 20
- Risposte . . . 22

3 Eventi, listener e modelli **24**
- 3.1 Chi avvisa chi . . . 24
- 3.2 Un listener per molti componenti . . . 25
- 3.3 Eventi semantici e di basso livello . . . 26
- 3.4 Dati veri nei componenti: oggetti e modelli . . . 28
- 3.5 Come appare una voce: il renderer . . . 30
- 3.6 Un componente nostro, con i suoi eventi . . . 31
- 3.7 L'esempio guida: la barra che funziona . . . 33
- Esercizi . . . 35
- Risposte . . . 37

4 Finestre di dialogo **39**
- 4.1 Fermarsi a chiedere . . . 39
- 4.2 Messaggi e domande: `JOptionPane` . . . 40
- 4.3 Scegliere un file . . . 43
- 4.4 Scegliere un colore . . . 44
- 4.5 Una finestra di dialogo nostra . . . 45
- 4.6 L'esempio guida: la Lavagna che chiede . . . 48
- Esercizi . . . 49
- Risposte . . . 52

5 Il componente che si disegna **53**
- 5.1 Dove si disegna . . . 53
- 5.2 Chi chiama `paintComponent`, e quando . . . 55
- 5.3 Le coordinate dei pixel . . . 56

5.4 Le primitive di `Graphics` . . . 57
5.5 Colori e testo . . . 59
5.6 L'esempio guida: la Lavagna che disegna . . . 60
Esercizi . . . 64
Risposte . . . 66

6 `Graphics2D`: forme, tratti e colori **67**
6.1 Perché un secondo contesto . . . 67
6.2 Le forme come oggetti . . . 68
6.3 Percorsi e aree . . . 69
6.4 Il tratto . . . 71
6.5 Il colore come pittura . . . 72
6.6 Il contesto da non sporcare . . . 73
6.7 L'esempio guida: figure con lo spessore . . . 73
Esercizi . . . 76
Risposte . . . 78

7 Coordinate e trasformazioni **80**
7.1 Due sistemi di riferimento . . . 80
7.2 Trasformare il sistema . . . 81
7.3 Disegnare attorno all'origine . . . 83
7.4 Il piano cartesiano . . . 85
7.5 Il grafico di una funzione . . . 86
7.6 L'esempio guida: la Lavagna con lo zoom . . . 87
Esercizi . . . 89
Risposte . . . 92

8 Mouse, tastiera e tempo **93**
8.1 Il mouse . . . 93
8.2 Un gesto in tre eventi . . . 94
8.3 La rotellina e le proprietà che avvisano . . . 95
8.4 La tastiera . . . 96
8.5 Le scorciatoie . . . 97
8.6 Il tempo: `javax.swing.Timer` . . . 98
8.7 L'esempio guida: la Lavagna completa . . . 100
Esercizi . . . 101
Risposte . . . 103

9 Palestra **105**
Problemi . . . 105

10 Dove andare adesso **110**
10.1 Separare il Model dalla View . . . 110
10.2 La grafica per pixel . . . 110
10.3 Tabelle, alberi e testo . . . 111
10.4 L'aspetto . . . 111
10.5 Accessibilità e internazionalizzazione . . . 111
10.6 Oltre Swing . . . 111
10.7 Nelle altre dispense . . . 111
10.8 Riferimenti . . . 112

Capitolo 1

Introduzione: una lavagna che si rompe

1.1 Il problema

Il programma che accompagna tutta la dispensa è una piccola applicazione da disegno, la *Lavagna*. Una finestra con una barra in alto, in cui si sceglie la forma (rettangolo, ellisse, mano libera), il colore e lo spessore del tratto; sotto, un'area bianca in cui si disegna trascinando il mouse; in fondo una riga di stato che dice quante figure ci sono e dove si trova il puntatore. Il tasto destro cancella la figura che sta sotto il puntatore, *Ctrl+Z* annulla l'ultima, un menu *File* apre e salva il disegno, e chiudendo la finestra con delle modifiche non salvate il programma lo fa notare. Per finire, un pulsante *Riproduci* ridisegna le figure una alla volta, nell'ordine in cui sono state tracciate.

Nessuna di queste funzioni è difficile da descrivere, e per costruire la finestra abbiamo a disposizione uno strumento che sembra fatto apposta: il *designer visuale* dell'ambiente di sviluppo (in Eclipse si chiama WindowBuilder). Trasciniamo i componenti sulla finestra, li sistemiamo con il mouse, e il designer scrive il codice al posto nostro. Il risultato, per la Lavagna, è questo metodo, chiamato dal costruttore della classe:

```
private void initialize() {
    frame = new JFrame();
    frame.setTitle("Lavagna");
    frame.setBounds(100, 100, 600, 400);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(null);

    JComboBox<String> cmbForma = new JComboBox<>();
    cmbForma.setModel(new DefaultComboBoxModel<>(
            new String[] { "Rettangolo", "Ellisse", "Mano libera" }));
    cmbForma.setBounds(10, 10, 130, 27);
    frame.getContentPane().add(cmbForma);

    JButton btnColore = new JButton("Colore...");
    btnColore.setBounds(150, 10, 100, 27);
    frame.getContentPane().add(btnColore);
```

Lo possiamo leggere senza fatica anche senza conoscere Swing. Creiamo una finestra (`JFrame`) di 600 per 400 pixel; dichiariamo che il suo contenuto non ha un *layout* (`setLayout(null)`), cioè che ogni componente verrà messo a mano; e poi, per ciascun componente, lo creiamo, gli assegniamo posizione e dimensioni con `setBounds(x, y, larghezza, altezza)` e lo aggiungiamo. Il resto del metodo continua allo stesso modo con lo slider dello spessore, l'area in cui disegneremo e la riga di stato:

```
JSlider sldSpessore = new JSlider(1, 20, 3);
sldSpessore.setBounds(260, 10, 150, 27);
frame.getContentPane().add(sldSpessore);

JPanel pnlArea = new JPanel();
pnlArea.setBackground(Color.WHITE);
pnlArea.setBounds(10, 45, 580, 300);
frame.getContentPane().add(pnlArea);

JLabel lblStato = new JLabel("0 figure");
lblStato.setBounds(10, 350, 300, 16);
frame.getContentPane().add(lblStato);
```

Sono i numeri che il designer ha letto dalla posizione in cui abbiamo lasciato cadere i componenti con il mouse: ognuno ha il suo rettangolo, misurato in pixel dall'angolo in alto a sinistra del contenuto della finestra.

Per disegnare aggiungiamo all'area bianca un *listener* del mouse: un oggetto i cui metodi vengono chiamati quando il pulsante del mouse viene premuto e rilasciato. Alla pressione ricordiamo il punto di partenza; al rilascio prendiamo il contesto grafico del pannello con `getGraphics()` e ci disegniamo un rettangolo. È la soluzione che viene in mente per prima, ed è sbagliata: il riquadro rosso segnala il codice da non imitare, e il resto del capitolo mostra perché.

```
pnlArea.addMouseListener(new MouseAdapter() {
    private Point inizio;

    public void mousePressed(MouseEvent e) {
        inizio = e.getPoint();
    }

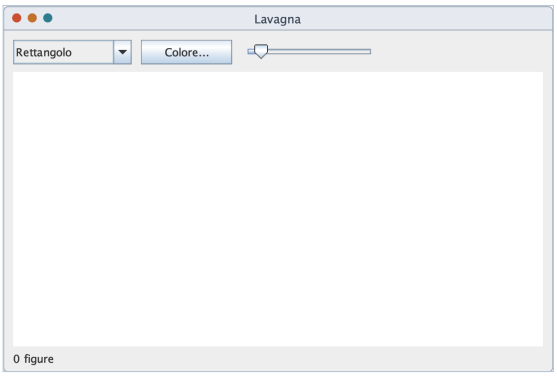
    public void mouseReleased(MouseEvent e) {
        // SBAGLIATO: non si disegna con getGraphics() in un listener
        Graphics g = pnlArea.getGraphics();
        g.setColor(Color.BLUE);
        int x = Math.min(inizio.x, e.getX()), y = Math.min(inizio.y, e.getY());
        int w = Math.abs(e.getX() - inizio.x), h = Math.abs(e.getY() - inizio.y);
        g.drawRect(x, y, w, h);
        lblStato.setText("ultima figura in " + x + ", " + y);
    }
});
```

Attenzione 1.1: `getGraphics()` non è il modo di disegnare

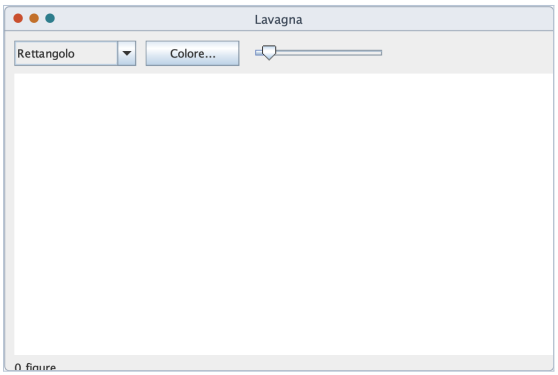
Il codice compila e, al primo tentativo, sembra funzionare. Ma ciò che si disegna su un contesto ottenuto con `getGraphics()` non viene ricordato da nessuno: alla prima occasione in cui Swing ridisegna il pannello, il rettangolo sparisce. In un programma Swing non si disegna mai così. Il modo corretto, che il capitolo 5 costruisce passo per passo, è conservare le figure in un campo e disegnarle in `paintComponent`.

Lanciamo il programma: mostra la finestra che ci aspettiamo, e il rettangolo compare

davvero dove lo abbiamo tracciato. Per qualche secondo sembra che manchi solo il resto delle funzioni. Poi cominciamo a usarlo.

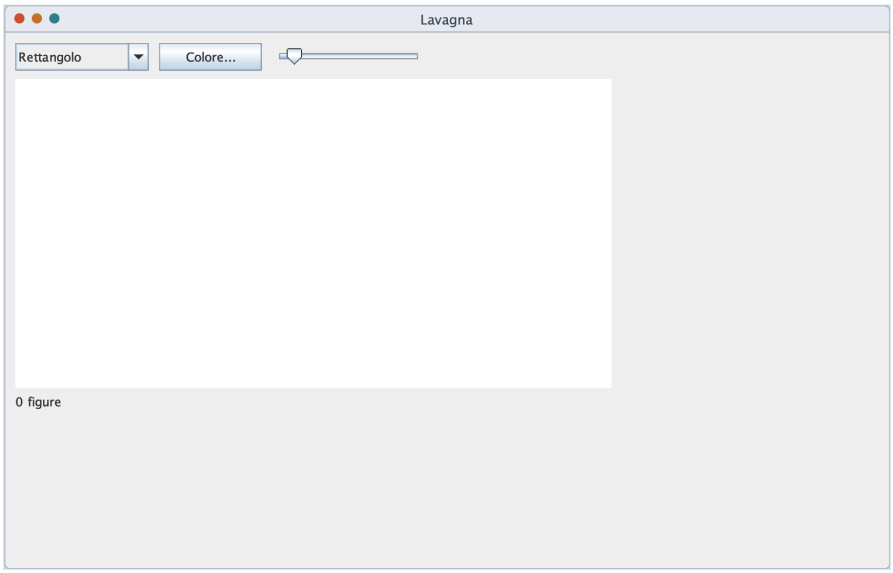


La Lavagna del designer, come la vediamo sul computer su cui l'abbiamo costruita.



La stessa finestra con i bordi di Windows: area e riga di stato tagliate.

Ingrandiamo la finestra. I componenti restano dove sono, con le misure che avevano. L'area bianca, che dovrebbe essere la parte importante, resta di 580 per 300 pixel nell'angolo in alto a sinistra, e tutto lo spazio guadagnato è grigio e inutile. Se la rimpiccioliamo succede il contrario: la riga di stato e poi l'area spariscono oltre il bordo.



La Lavagna del designer ingrandita: lo spazio in più resta vuoto.

Cambiamo computer. I 600 per 400 pixel di `setBounds` sono la misura della finestra *intera*, bordi e barra del titolo compresi, e i bordi non sono uguali dappertutto. Sul sistema dove è nata la finestra lo spazio interno era di 600 per 372 pixel; con i bordi di Windows è di 584 per 361, e la riga di stato, che finiva al pixel 366, viene tagliata (la seconda schermata qui sopra). Lo stesso accade se cambiano i caratteri: con un font predefinito più grande, o con lo schermo ingrandito al 125%, la scritta "Colore…" non sta più nei suoi cento pixel.

Copriamo la finestra. Disegniamo tre rettangoli, poi portiamo davanti un'altra finestra, oppure riduciamo la Lavagna a icona e la riapriamo, oppure la ridimensioniamo: i rettangoli

sono spariti. Non c'è nessun errore e nessun messaggio; semplicemente, ciò che avevamo disegnato non c'è più. Basta anche meno: se apriamo la tendina delle forme sopra un rettangolo e la richiudiamo, la parte di rettangolo che la tendina copriva è cancellata.

Chiudiamo. La X della finestra termina il programma all'istante, con tutto il disegno, senza chiedere nulla. Del resto non ci sarebbe niente da salvare: il programma non sa quali figure sono state disegnate, perché non le ha mai tenute da nessuna parte. Le ha soltanto dipinte sullo schermo.

Ognuno di questi difetti ha una causa precisa, e nessuna è un errore di battitura. Ne nascono le tre domande della dispensa.

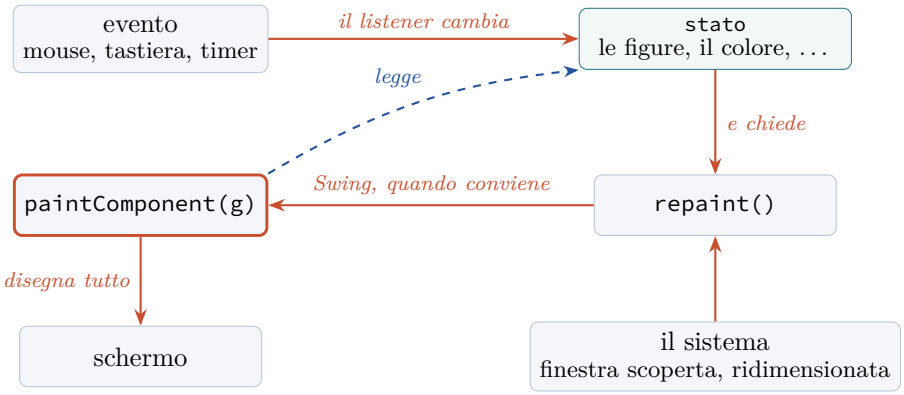
- **Come si compone una finestra che si adatta?** Alle dimensioni che l'utente sceglie, ai bordi e ai caratteri del sistema su cui gira, ai testi che cambiano. Le coordinate assolute non possono farlo: serve chi decida le misure a ogni ridimensionamento.
- **Dove si disegna, perché il disegno resti?** Lo schermo non ricorda niente: quando una parte della finestra va ridisegnata, Swing la ridisegna da capo. Il programma deve sapere che cosa c'era, e saperlo ridisegnare quando glielo si chiede.
- **Come si parla con l'utente?** Rispondere al mouse e alla tastiera, ma anche chiedere: quale file aprire, dove salvare, quale colore, se buttare via le modifiche. Sono finestre che si aprono, aspettano una risposta e la restituiscono.

Il capitolo 9 riprende la Lavagna e la porta fino in fondo, con tutte le funzioni descritte all'inizio. In mezzo c'è tutto ciò che serve per arrivarci, e soprattutto un modo di pensare il programma diverso da quello del designer.

1.2 L'immagine guida: evento, stato, disegno

Il primo programma Java che abbiamo scritto faceva una cosa dall'inizio alla fine: leggeva, calcolava, stampava, terminava. Un programma con un'interfaccia grafica no. Costruisce la finestra, la mostra, e poi *aspetta*: che l'utente prema un pulsante, muova il mouse, scriva un carattere; che il sistema chieda di ridisegnare una parte della finestra appena scoperta; che scatti un timer. Ognuna di queste cose è un *evento*, e il programma è fatto di pezzi di codice che reagiscono agli eventi. Chi esegue quei pezzi è sempre lo stesso thread di Swing, che estrae gli eventi da una coda uno alla volta: come funziona, e perché un listener non deve mai durare a lungo, lo spiega in dettaglio *Thread e concorrenza*. Qui basta sapere che è così.

Dentro questo ciclo c'è un'immagine che tornerà in ogni capitolo. Gli eventi *non disegnano*: cambiano lo *stato* del programma, cioè i dati che descrivono ciò che deve comparire (la lista delle figure, il colore scelto, la posizione del mouse). Poi chiedono un ridisegno con `repaint()`, e non fanno nient'altro. Quando le conviene, Swing chiama il metodo `paintComponent` del componente, che disegna lo stato da capo, tutto. Le richieste di ridisegno possono venire anche dal sistema, quando la finestra viene scoperta o ridimensionata: il metodo che risponde è lo stesso, e siccome disegna a partire dallo stato, il risultato è lo stesso.



Il ciclo di ogni interfaccia di questa dispensa: gli eventi cambiano lo stato, `paintComponent` lo disegna.

La Lavagna del designer salta il passaggio centrale: il listener disegna direttamente sullo schermo, e lo stato non esiste. Per questo il disegno sparisce alla prima richiesta di ridisegno, e per questo non c'è niente da salvare. La versione giusta, nel capitolo 5, tiene una lista di figure; il listener del mouse aggiunge una figura alla lista e chiama `repaint()`; `paintComponent` percorre la lista e disegna. Il disegno non può più sparire, perché si può sempre rifare, e salvarlo vuol dire scrivere la lista in un file.

Lo stesso schema, in piccolo, sta dentro i componenti di Swing. Una tendina non tiene le sue voci dentro di sé: le legge da un *modello*, e quando il modello cambia la tendina si ridisegna da sola (capitolo 3). Una finestra di dialogo non modifica la finestra che l'ha aperta: restituisce un valore, e chi l'ha aperta aggiorna il proprio stato (capitolo 4). Un timer non anima niente: a ogni scatto cambia la posizione degli oggetti e chiede un ridisegno (capitolo 8).

Osservazione 1.1

Il salto è nel modo di pensare il disegno. Il designer porta a ragionare per *azioni*: quando succede questo, disegna quello. Swing chiede di ragionare per *descrizioni*: dato lo stato, ecco come appare la finestra. Il momento in cui disegnare non lo decide il programma ma Swing, che può chiedere lo stesso disegno una volta o cento; il programma deve solo essere sempre in grado di farlo. È la stessa idea della pagina web che si ricostruisce dai dati, e di molte librerie grafiche moderne: chi la capisce qui la ritrova dappertutto.

1.3 Che cosa serve

Le basi del linguaggio e degli oggetti, le collezioni e la lettura e scrittura di file di testo (*Java Fast Forward*): la Lavagna salva le sue figure in un file di testo, una per riga, e le tiene in una `List`. I record, dalla stessa dispensa: molti dati di questa (una figura, un colore con il suo nome, la scelta fatta in una finestra di dialogo) sono valori, e diventano record. Le classi anonime, le lambda e i method reference (*Dal nome alla funzione*): quasi ogni listener è una lambda, e la scala dalla classe anonima alla lambda si vede qui all'opera. Le interfacce (*Le interfacce in Java moderno*): un listener è un'interfaccia, e molti listener hanno metodi `default` o classi *adapter*. Del rapporto fra Swing e i thread basta ciò che si è detto nella sezione precedente; chi vuole sapere che cosa c'è sotto, o deve eseguire un lavoro lungo senza bloccare la finestra, trova tutto in *Thread e concorrenza*.

Il codice è Java 21 e usa solo la libreria standard. Le schermate mostrano le finestre con l'aspetto *Metal*, quello predefinito di Swing quando il programma non ne sceglie un altro; su macOS l'aspetto predefinito è quello del sistema, e i componenti si disegnano in modo diverso, ma si comportano allo stesso modo.

Restano fuori, di proposito, i componenti per i dati strutturati (tabelle e alberi), il testo con stili e formattazione, la grafica per pixel con `BufferedImage`, la scelta dell'aspetto (*look and feel*) e JavaFX: ne parla, con un assaggio ciascuno, il capitolo 10. Resta fuori anche l'organizzazione di un'applicazione grafica grande in Model, View e Controller: è un tema di architettura, che merita una trattazione a sé; qui se ne vede soltanto la versione in piccolo che sta dentro i componenti.

1.4 La strada che percorreremo

Il capitolo 2 risponde alla prima domanda. Che cosa sono una finestra, un componente e un contenitore; i componenti di tutti i giorni; i *layout manager*, che decidono posizione e dimensione dei componenti a ogni ridimensionamento; come si annidano i pannelli per ottenere la disposizione voluta; che cosa genera il designer e come lo si corregge. Alla fine la Lavagna è composta e si adatta, ma non fa ancora nulla.

Il capitolo 3 la fa reagire. Il meccanismo degli eventi e degli listener, gli eventi dei componenti, i *modelli* che stanno dietro alle tendine e alle liste, i *renderer* che decidono come appare ogni voce, e un componente nostro che genera i suoi eventi. Alla fine la barra della Lavagna funziona.

Il capitolo 4 è quello delle finestre di dialogo: i messaggi e le domande di `JOptionPane`, la scelta di un file e di un colore con le finestre che il sistema mette a disposizione, e una finestra di dialogo scritta da noi che restituisce la scelta dell'utente. La Lavagna apre, salva e chiede prima di chiudere.

Il capitolo 5 risponde alla seconda domanda: il componente che si disegna da solo, ridefinendo `paintComponent` in un `JComponent` o in un `JPanel`; chi lo chiama e quando, `repaint`, il sistema di coordinate dei pixel e le primitive del contesto grafico `Graphics`: linee, rettangoli, ovali, archi, poligoni, testo, colori e font. La Lavagna smette di perdere il disegno.

Il capitolo 6 passa a `Graphics2D`, la versione completa del contesto grafico: le forme come oggetti (`Shape`), i percorsi con segmenti e curve, lo spessore e lo stile del tratto, le sfumature, la trasparenza, l'antialiasing. Le figure della Lavagna diventano forme, con lo spessore scelto nella barra.

Il capitolo 7 cambia sistema di riferimento: le coordinate del problema e quelle dei pixel, le trasformazioni (traslazione, scala, rotazione), il piano cartesiano con l'origine al centro, il grafico di una funzione. La Lavagna ottiene una griglia e lo zoom.

Il capitolo 8 riprende mouse e tastiera nel dettaglio e aggiunge il tempo: un gesto fatto di tre eventi, la scelta di una figura con il puntatore, i tasti premuti insieme, le scorciatoie, le animazioni con `javax.swing.Timer`.

Il capitolo 9 è la palestra: otto programmi completi, in difficoltà crescente, dall'esercizio d'esame alla Lavagna finita. Il capitolo 10 indica dove continuare.

Esercizi

Esercizio 1.1: Contare i pixel (⋆)

Nella Lavagna del designer lo spazio interno della finestra è di 600 per 372 pixel sul sistema dove è nata, e di 584 per 361 con i bordi di Windows. Usando i numeri delle `setBounds`, dire quali componenti vengono tagliati sul secondo sistema, da quale lato e di quanti pixel. Che cosa succede su un sistema in cui i caratteri predefiniti sono più grandi del 25%?

Esercizio 1.2: Quando sparisce (⋆)

Nella Lavagna del designer si disegna un rettangolo. Per ciascuna azione dire se il rettangolo sopravvive, e perché: (a) si sposta la finestra trascinandola per la barra del titolo; (b) la si riduce a icona e la si riapre; (c) la si allarga di un pixel; (d) si apre la tendina delle forme sopra il rettangolo e la si richiude; (e) si traccia un secondo rettangolo che si sovrappone al primo.

Esercizio 1.3: Chi disegna (⋆)

Nella Lavagna del designer, quale thread esegue `mouseReleased`? E quale disegna il rettangolo? Il problema del disegno che sparisce è un problema di thread? Che cosa cambierebbe se il rettangolo fosse disegnato da un altro thread?

Risposte

Risposta 1.1. L'area va da $x = 10$ a $x = 590$ e da $y = 45$ a $y = 345$: con 584 pixel di larghezza perde 6 pixel sul lato destro, mentre in altezza ci sta. La riga di stato va da $y = 350$ a $y = 366$ e lo spazio finisce a 361: perde 5 dei suoi 16 pixel in basso, cioè la parte inferiore delle lettere. Nessun altro componente è toccato, perché stanno tutti in alto a sinistra. Con caratteri più grandi del 25% i testi non stanno più nelle misure fissate: "Colore…" nel suo pulsante di 100 pixel viene abbreviato con i puntini, "Rettangolo" nella tendina pure, e la riga di stato, alta 16 pixel, taglia le lettere anche sul sistema originale. Le misure scritte a mano valgono soltanto per il sistema e i caratteri con cui sono state prese.

Risposta 1.2. (a) Di solito sopravvive: i sistemi moderni conservano l'immagine di ogni finestra e la spostano senza chiedere di ridisegnarla; su un sistema che non lo fa sparirebbe. Non è una garanzia, quindi, ma un caso. (b) Sparisce: riaprendo la finestra il sistema chiede di ridisegnarla, e il pannello ridisegna ciò che sa, cioè uno sfondo bianco. (c) Sparisce, per lo stesso motivo: ogni ridimensionamento ridisegna tutto. (d) Sparisce la parte coperta dalla tendina: alla chiusura Swing ridisegna la zona che la tendina occupava, e il pannello la ridisegna bianca. (e) Sopravvive: il secondo rettangolo viene disegnato sopra, senza cancellare niente. Il risultato dipende dalle circostanze, e un programma il cui aspetto dipende dalle circostanze è sbagliato.

Risposta 1.3. Entrambe le cose le fa lo stesso thread, quello che Swing usa per gli eventi (lo si può verificare stampando `Thread.currentThread().getName()`: `AWT-EventQueue-0`). Il problema non è di thread ma di *memoria*: il disegno viene fatto sui pixel dello schermo e da nessun'altra parte, e quando Swing ridisegna il pannello non sa che c'erano dei rettangoli. Disegnare da un altro thread peggiorerebbe le cose: i componenti di Swing non sono fatti per essere toccati da thread diversi da quello degli eventi, e il disegno si mescolerebbe in modo imprevedibile con quello di Swing.

Capitolo 2

La finestra e i suoi componenti

La Lavagna del capitolo 1 aveva tutti i componenti giusti nei posti sbagliati: messi a mano, con coordinate che valevano solo per la finestra, il sistema e i caratteri con cui erano state misurate. Questo capitolo risponde alla prima delle tre domande, come si compone una finestra che si adatta. Si parte da che cosa c'è in una finestra Swing, si passano in rassegna i componenti di tutti i giorni, e si arriva al punto centrale: i *layout manager*, gli oggetti che calcolano posizione e dimensione dei componenti ogni volta che serve. Alla fine la Lavagna avrà la stessa barra, la stessa area e la stessa riga di stato, senza un solo `setBounds`, e si adatterà a qualunque dimensione. Non farà ancora niente: gli eventi sono il tema del capitolo 3.

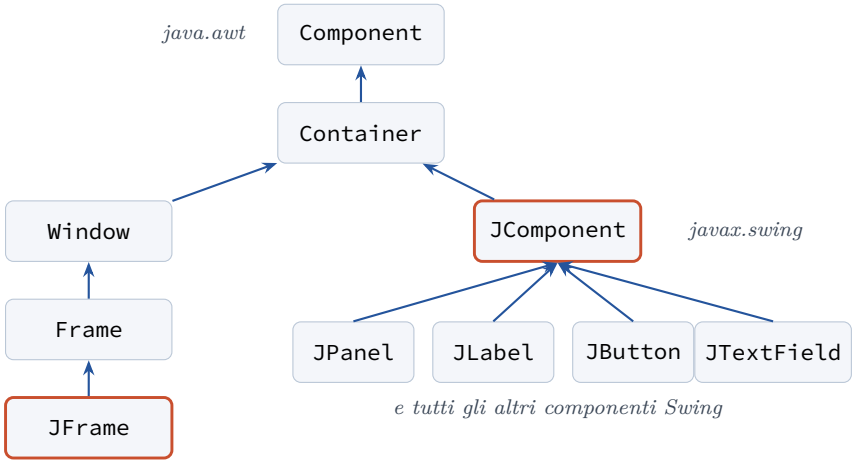
2.1 Finestre, contenitori, componenti

Guardando la Lavagna si vede una finestra che contiene una barra, un'area e una riga di stato; la barra contiene a sua volta una tendina, un pulsante e uno slider. È una struttura ad albero, e Swing la rappresenta proprio così. Ogni oggetto che occupa un rettangolo sullo schermo è un *componente*; i componenti che possono contenerne altri sono *contenitori*; in cima all'albero c'è una *finestra*, l'unico contenitore che il sistema operativo conosce davvero.

Definizione 2.1: Componente, contenitore, finestra

Un *componente* è un oggetto di una sottoclasse di `java.awt.Component`: ha una posizione e una dimensione dentro il suo contenitore, sa disegnarsi e riceve gli eventi del mouse e della tastiera che lo riguardano. Un *contenitore* è un componente di una sottoclasse di `java.awt.Container`, che tiene una lista di componenti figli e li dispone al proprio interno. Una *finestra di primo livello* (`JFrame`, `JDialog`) è un contenitore che non sta dentro nessun altro: è il sistema operativo a darle una cornice, una barra del titolo e un posto sullo schermo.

Le classi sono di due famiglie, e i nomi lo dicono. Quelle di `java.awt` (AWT, *Abstract Window Toolkit*) sono la base, la prima libreria grafica di Java: `Component`, `Container`, `Window`, `Frame`, e anche un `Button` e una `Label` che oggi non si usano più. Quelle di `javax.swing`, con la `J` davanti, sono costruite sopra: `JComponent` è la radice di tutti i componenti Swing, e ogni componente che si usa ne discende.



Le classi principali: una freccia va dalla sottoclasse alla superclasse. Ogni componente Swing è anche un `Container`.

La figura spiega due fatti che sorprendono. Il primo: `JFrame` non discende da `JComponent`, perché una finestra di primo livello è un oggetto del sistema operativo, e Swing la ottiene estendendo la `Frame` di AWT. Il secondo: ogni componente Swing, anche un `JButton`, è un `Container`, e tecnicamente potrebbe contenerne altri; nella pratica i contenitori che si usano per raggruppare sono i `JPanel`, e nient'altro.

Attenzione 2.1: Una famiglia sola

`Button` e `JButton`, `Label` e `JLabel`, `Panel` e `JPanel`: le classi AWT senza la `J` esistono ancora e compilano, e l'autocompletamento dell'ambiente di sviluppo le propone volentieri. Mescolarle con quelle di Swing produce errori di disegno difficili da capire, perché le due famiglie si disegnano in modo diverso. Nei programmi Swing i componenti sono sempre quelli con la `J`; da `java.awt` si importano solo le classi di supporto: i layout, i colori, i font, gli eventi.

2.1.1 La prima finestra

La finestra più semplice possibile contiene un'etichetta. La scriviamo come una classe che *è* una finestra, cioè che estende `JFrame`, e che costruisce il proprio contenuto nel costruttore:

```
public class PrimaFinestra extends JFrame {

    public PrimaFinestra() {
        super("Prima finestra");                        // il titolo
        add(new JLabel("Ciao, Swing", SwingConstants.CENTER));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300, 120);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new PrimaFinestra().setVisible(true));
    }
}
```

Il costruttore passa il titolo alla superclasse, aggiunge un'etichetta con il testo centrato, dice che cosa fare quando l'utente chiude la finestra e ne fissa la dimensione. Il `main` non costruisce

la finestra direttamente: chiede a Swing di farlo con `invokeLater`, che esegue la lambda sul thread degli eventi. È la regola dei programmi Swing, "tutto ciò che tocca i componenti si fa su quel thread", e vale anche per la costruzione; il perché è in *Thread e concorrenza*. Per ultima si chiama `setVisible(true)`, che mostra la finestra: prima si costruisce tutto, poi si mostra.

Dove finisce l'etichetta? Una `JFrame` non tiene i componenti direttamente: al suo interno ha un *root pane*, che contiene a sua volta l'eventuale barra dei menu e un pannello per il contenuto, il *content pane*. È lì che vanno i componenti, e `add` chiamato sulla finestra li inoltra al content pane: `add(c)` e `getContentPane().add(c)`, la forma che usa il designer, fanno la stessa cosa.

La riga `setDefaultCloseOperation` merita attenzione, perché il valore predefinito non è quello che ci si aspetta.

Costante	**Che cosa succede alla X della finestra**
`HIDE_ON_CLOSE` (predefinito)	la finestra si nasconde; il programma continua
`DISPOSE_ON_CLOSE`	la finestra si chiude e libera le risorse
`EXIT_ON_CLOSE`	il programma termina (`System.exit(0)`)
`DO_NOTHING_ON_CLOSE`	niente: decide il programma, con un listener

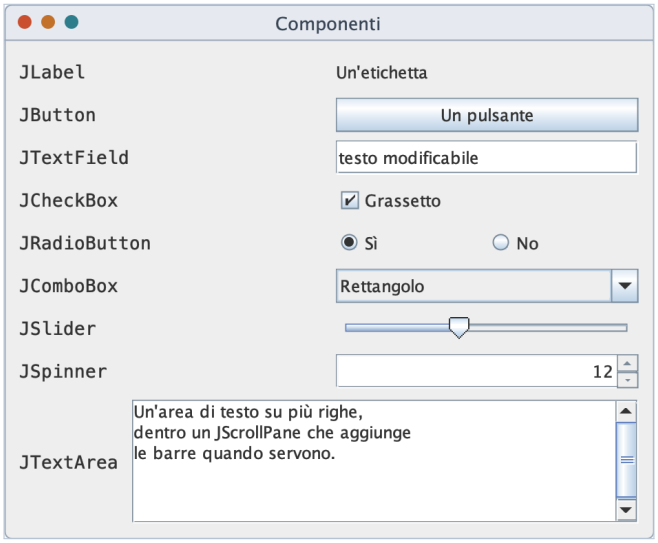
Attenzione 2.2: Il programma che non termina

Senza `setDefaultCloseOperation` la X nasconde la finestra e basta: il programma resta in vita, invisibile, e in Eclipse il quadratino rosso della console resta acceso. Per la finestra principale di un'applicazione si sceglie `EXIT_ON_CLOSE`, oppure `DISPOSE_ON_CLOSE`, con cui il programma termina da solo quando non resta nessuna finestra aperta. `DO_NOTHING_ON_CLOSE` serve quando prima di chiudere si vuole chiedere qualcosa all'utente: è il caso della Lavagna, nel capitolo 4.

Il designer visuale scrive la stessa finestra in un altro modo: una classe qualunque con un campo `private JFrame frame`, un metodo `initialize()` che lo costruisce, e un `main` con una classe anonima al posto della lambda. Sono due stili equivalenti. Estendere `JFrame` è naturale quando la classe *è* la finestra e non fa altro; tenere la finestra in un campo è preferibile quando la classe ha un compito più ampio e la finestra è solo uno dei suoi strumenti. In questa dispensa le finestre estendono `JFrame` e i pannelli che si disegnano estendono `JPanel`.

2.2 I componenti di tutti i giorni

La libreria offre qualche decina di componenti; una manciata basta per quasi tutto. La schermata mostra i più comuni, ciascuno accanto al nome della sua classe.



I componenti di tutti i giorni, con l'aspetto Metal.

Ogni componente ha un valore che l'utente può cambiare, e un paio di metodi per leggerlo e scriverlo. La tabella raccoglie quelli che servono davvero.

Classe	Che cosa è	Il suo valore
`JLabel`	un testo o un'icona, non modificabili	`getText`, `setText`
`JButton`	un pulsante	(nessuno: si preme)
`JTextField`	una riga di testo	`getText`, `setText`
`JTextArea`	più righe di testo	`getText`, `append`
`JCheckBox`	una casella sì/no	`isSelected`, `setSelected`
`JRadioButton`	una scelta fra alternative	`isSelected`, con un `ButtonGroup`
`JComboBox<T>`	una tendina di voci	`getSelectedItem`, `getSelectedIndex`
`JList<T>`	un elenco visibile di voci	`getSelectedValue`
`JSlider`	un intero fra un minimo e un massimo	`getValue`, `setValue`
`JSpinner`	un valore con le frecce	`getValue`, `setValue`

Il frammento seguente crea alcuni componenti della barra e, più avanti nel programma, ne legge i valori. È il modo in cui un programma interroga la propria interfaccia: quando serve, chiede a ogni componente che cosa contiene.

```
JTextField txtNome = new JTextField(20);              // 20 colonne di larghezza
JCheckBox chkGrassetto = new JCheckBox("Grassetto");
String[] forme = { "Rettangolo", "Ellisse" };
JComboBox<String> cmbForma = new JComboBox<>(forme);
JSlider sldSpessore = new JSlider(1, 20, 3);          // minimo, massimo, iniziale
JSpinner spnDimensione = new JSpinner(new SpinnerNumberModel(12, 6, 72, 1));

String nome = txtNome.getText();
boolean grassetto = chkGrassetto.isSelected();
String forma = (String) cmbForma.getSelectedItem();
int spessore = sldSpessore.getValue();
int dimensione = (Integer) spnDimensione.getValue();
```

Due righe hanno un cast, e non per distrazione. `getSelectedItem` restituisce un `Object` anche in una `JComboBox<String>`, perché il metodo è più vecchio dei generici e non è mai stato cambiato; `JSpinner` può contenere numeri, date o voci di un elenco, e il suo valore è un

`Object` per la stessa ragione. Il numero passato a `JTextField` non è un limite alla lunghezza del testo ma la larghezza *preferita* del campo, misurata in caratteri: un'informazione per i layout della sezione 2.3.

2.2.1 Testo lungo: lo `JScrollPane`

Una `JTextArea` cresce con il testo che contiene, e da sola non ha barre di scorrimento. Le barre le aggiunge un contenitore apposito, lo `JScrollPane`, che mostra una porzione del componente che gli si affida e permette di spostarla:

```
JTextArea txtNote = new JTextArea(5, 30);       // righe e colonne preferite
txtNote.setLineWrap(true);                      // a capo a fine riga...
txtNote.setWrapStyleWord(true);                 // ...fra due parole
add(new JScrollPane(txtNote), BorderLayout.CENTER);     // lo scroll pane
```

A essere aggiunto alla finestra è lo scroll pane, non l'area: aggiungere l'area e poi anche lo scroll pane, o solo l'area, è un errore frequente, il cui sintomo è un testo che esce dal bordo senza barre. Lo stesso vale per una `JList` con molte voci.

2.2.2 Bordi, margini e menu

Un componente può avere un *bordo*, che è anche il modo standard di dargli un margine. `BorderFactory` li fabbrica: `createEmptyBorder(alto, sinistra, basso, destra)` è un margine trasparente, `createLineBorder(colore)` una linea, `createTitledBorder("titolo")` una cornice con un'intestazione, utile per raggruppare i campi di un modulo. Si assegnano con `setBorder`.

La barra dei menu non sta nel content pane ma nel root pane, sopra di esso. Si compone con tre classi: `JMenuBar` è la barra, `JMenu` un menu a tendina, `JMenuItem` una voce. Si assegna alla finestra con `setJMenuBar`:

```
JMenu mnuFile = new JMenu("File");
mnuFile.add(new JMenuItem("Apri..."));
mnuFile.add(new JMenuItem("Salva"));
mnuFile.addSeparator();
mnuFile.add(new JMenuItem("Esci"));

JMenuBar barra = new JMenuBar();
barra.add(mnuFile);
setJMenuBar(barra);
```

I puntini dopo "Apri" non sono decorativi: per convenzione indicano che la voce apre una finestra in cui l'utente dovrà scegliere qualcosa, mentre "Salva" agisce subito. Vale anche per i pulsanti, come il "Colore…" della Lavagna.

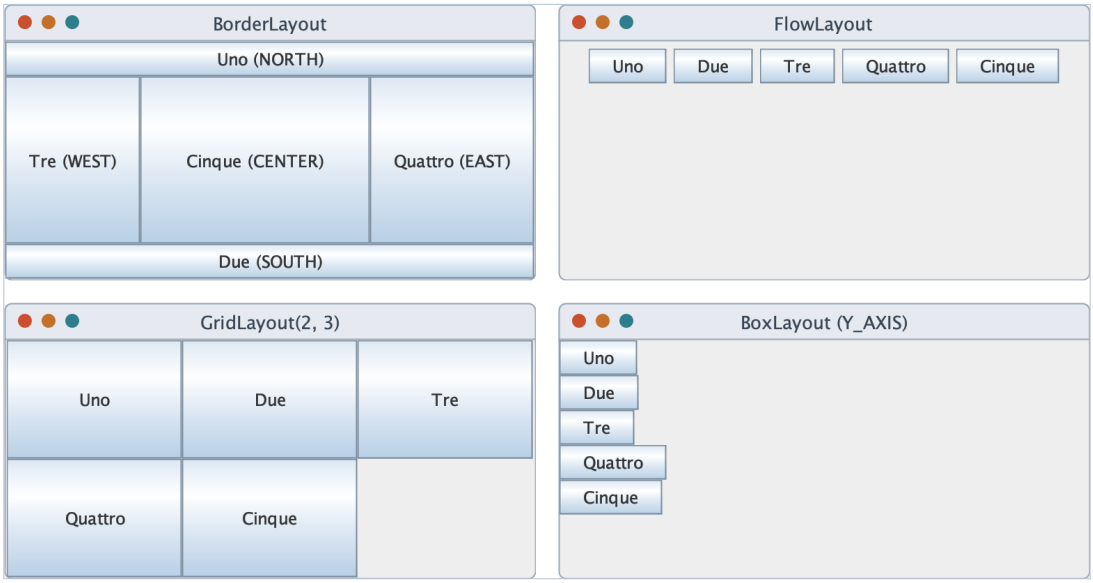
2.3 Chi decide dove: i layout manager

Torniamo al problema del capitolo 1. Posizione e dimensione di un componente non possono essere numeri scritti nel codice, perché dipendono da cose che il programma non conosce quando viene scritto: quanto è grande la finestra in questo momento, quanto sono grandi i caratteri su questo sistema, quanto è lungo il testo di un'etichetta. Vanno *calcolati*, e ricalcolati ogni volta che una di queste cose cambia. Swing affida il calcolo a un oggetto separato.

Definizione 2.2: Layout manager

Un *layout manager* è un oggetto associato a un contenitore (con `setLayout`) che ne dispone i componenti: ogni volta che il contenitore cambia dimensione, o che un componente viene aggiunto o tolto, calcola posizione e dimensione di ciascun figlio a partire dallo spazio disponibile e dalle dimensioni *preferite* dei figli. Calcola anche la dimensione preferita del contenitore stesso, cioè quella che gli serve per mostrare tutti i figli alla loro dimensione preferita.

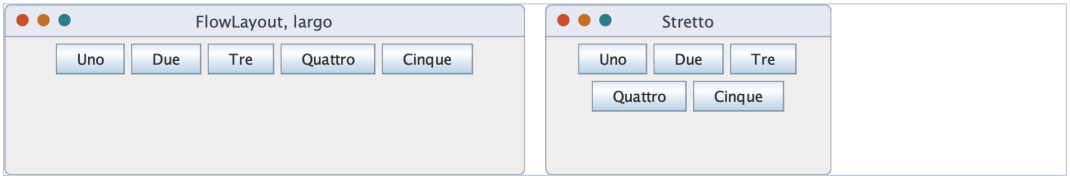
La parola chiave è *dimensione preferita*. Ogni componente sa quanto vorrebbe essere grande: un pulsante quanto il suo testo nel suo font, più il bordo; un campo di testo quanto le colonne che gli si sono chieste; un'etichetta quanto il testo che contiene. Lo sa perché lo calcola sul sistema su cui gira, con i caratteri che ci sono. Un layout che parte dalle dimensioni preferite si adatta quindi da solo ai caratteri e ai testi, e uno che usa lo spazio disponibile si adatta alla finestra: sono esattamente le due cose che il layout assoluto non sapeva fare. I layout di uso comune sono quattro; la schermata li mostra all'opera sugli stessi cinque pulsanti.



Gli stessi cinque pulsanti sotto quattro layout diversi.

`BorderLayout`. Divide il contenitore in cinque regioni: `NORTH` e `SOUTH` occupano tutta la larghezza, alla loro altezza preferita; `WEST` ed `EAST` lo spazio che resta in altezza, alla loro larghezza preferita; `CENTER` prende tutto il resto. È il layout delle finestre: la regione centrale è quella che cresce, e attorno stanno barre e pannelli di dimensione fissa. Il componente si aggiunge indicando la regione, `add(c, BorderLayout.NORTH)`; senza indicazione va in `CENTER`. Ogni regione contiene un solo componente, e le regioni vuote non occupano spazio.

`FlowLayout`. Mette i componenti in fila, da sinistra a destra, ciascuno alla sua dimensione preferita, e centra la fila nel contenitore; quando la larghezza non basta, va a capo come le parole di un paragrafo. È il layout delle barre di pulsanti. Il costruttore accetta l'allineamento (`FlowLayout.LEFT`, `CENTER`, `RIGHT`) e la spaziatura orizzontale e verticale fra i componenti.



Un `FlowLayout` che non ha abbastanza larghezza manda a capo i componenti.

`GridLayout`. Divide il contenitore in una griglia di celle tutte uguali e mette un componente per cella, riga per riga, stirandolo fino a riempirla. Le dimensioni preferite non contano: conta solo lo spazio. È il layout delle tastiere, delle scacchiere e delle colonne di pulsanti della stessa larghezza. `new GridLayout(righe, colonne)`; se uno dei due numeri è zero, viene calcolato dal numero di componenti: `new GridLayout(0, 1)` è una colonna lunga quanto serve.

`BoxLayout`. Mette i componenti in colonna (`Y_AXIS`) o in riga (`X_AXIS`), alla loro dimensione preferita, senza stirarli oltre la dimensione massima. Si costruisce passando il contenitore stesso, `new BoxLayout(pnl, BoxLayout.Y_AXIS)`, e si usa con `Box.createVerticalStrut(n)` per uno spazio fisso e `Box.createVerticalGlue()` per uno spazio che assorbe quello in eccesso.

Ogni contenitore nasce con un layout, e i due default da ricordare sono diversi fra loro:

Contenitore	Layout predefinito
content pane di `JFrame` e di `JDialog`	`BorderLayout`
`JPanel`	`FlowLayout` (centrato)
`Box`	`BoxLayout`

Il primo default spiega perché la prima finestra della sezione 2.1 mostrava l'etichetta al centro e grande quanto la finestra: `add` senza regione l'aveva messa in `CENTER`. E spiega un errore che prima o poi si commette.

Attenzione 2.3: Due componenti al centro

Nel content pane di una finestra, aggiungere due componenti senza indicare la regione non li mette uno accanto all'altro: li mette entrambi in `CENTER`, e il secondo prende il posto del primo, che scompare.

```
add(new JLabel("Nome:"));       // in CENTER...
add(new JTextField(20));        // ...anche lui: l'etichetta sparisce
```

Per avere due componenti affiancati si usa un pannello, che con il suo `FlowLayout` li mette in fila, e si aggiunge alla finestra il pannello:

```
JPanel riga = new JPanel();     // FlowLayout
riga.add(new JLabel("Nome:"));
riga.add(new JTextField(20));
add(riga, BorderLayout.NORTH);
```

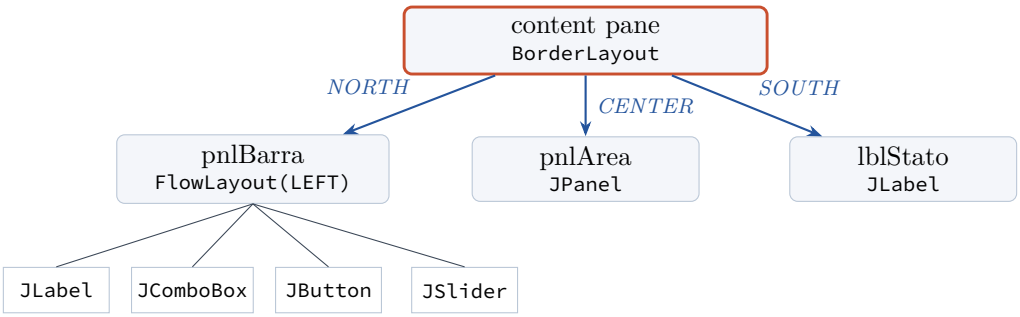
Con i layout la finestra non ha più bisogno di una dimensione scritta a mano. Il metodo `pack()` chiede al content pane la sua dimensione preferita, che il layout calcola dai figli, e ridimensiona la finestra perché la contenga esattamente, aggiungendo i bordi e la barra del titolo del sistema su cui gira. Al posto di `setSize(300, 120)` si scrive `pack()`, dopo aver aggiunto tutti i componenti.

Osservazione 2.1

Swing ha altri layout. `GridBagLayout` dispone i componenti in una griglia con righe e colonne di misure diverse, e con vincoli per ogni cella: può fare tutto, al prezzo di un codice lungo e poco leggibile. `CardLayout` mostra un componente alla volta, come le pagine di una procedura guidata. `GroupLayout` è quello che usano alcuni designer visuali. Con i quattro di questa sezione, annidati come nella prossima, si compone quasi ogni finestra, e il codice resta leggibile.

2.4 Comporre con i pannelli

Una finestra vera ha parti diverse che si comportano in modo diverso: la barra della Lavagna è una fila di componenti alla loro misura, l'area deve prendere tutto lo spazio, la riga di stato è un'etichetta larga quanto la finestra. Nessun layout da solo fa tutte e tre le cose, e non ce n'è bisogno: ogni `JPanel` ha il suo layout, e un pannello può stare dentro un altro. Si compone la finestra come un albero di pannelli, scegliendo per ogni nodo il layout adatto alla sua parte.

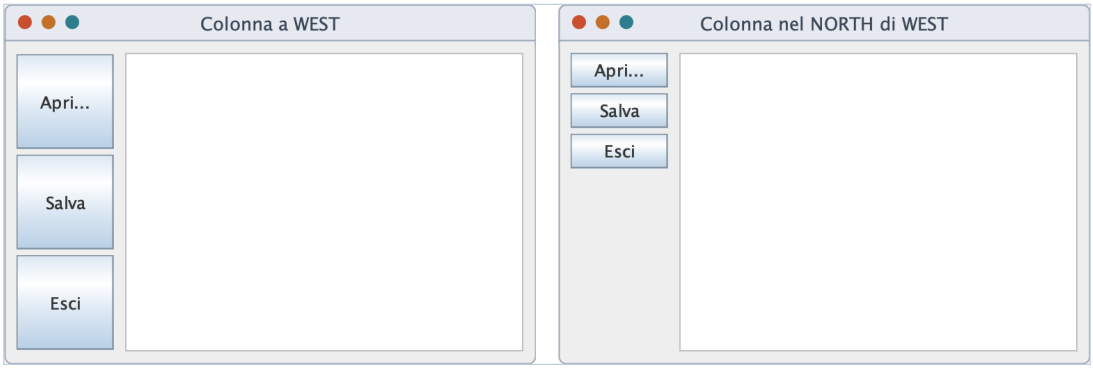


L'albero dei contenitori della Lavagna: il content pane divide la finestra, la barra mette in fila i suoi componenti.

Il metodo per arrivarci è chiedersi, per ogni parte, che cosa deve succedere quando la finestra cresce. La parte che deve crescere va nel `CENTER` di un `BorderLayout`; le parti che restano della loro misura vanno ai lati, ciascuna in un pannello con il layout che serve dentro di lei.

2.4.1 Il pannello che resta in alto

Un caso ricorrente mostra bene come si ragiona. Si vuole una colonna di pulsanti a sinistra della finestra, tutti della stessa larghezza, e al centro un'area di testo. La prima idea è un pannello con `GridLayout(0, 1)`, che fa proprio una colonna di celle uguali, messo nella regione `WEST`. Il risultato è la prima finestra della schermata: la regione `WEST` è alta quanto la finestra, il `GridLayout` divide quell'altezza fra i tre pulsanti, e i pulsanti diventano enormi.



A sinistra la colonna messa direttamente a `WEST`; a destra la stessa colonna nel `NORTH` di un pannello a `WEST`.

Il rimedio è un pannello in più. La colonna va nella regione `NORTH` di un pannello con `BorderLayout`, e quel pannello va a `WEST`. In `NORTH` la colonna riceve la sua altezza preferita, cioè tre pulsanti alla loro altezza, e lo spazio che avanza resta vuoto sotto di lei:

```
JPanel pnlPulsanti = new JPanel(new GridLayout(0, 1, 4, 4));  // celle uguali
pnlPulsanti.add(new JButton("Apri..."));
pnlPulsanti.add(new JButton("Salva"));
pnlPulsanti.add(new JButton("Esci"));

JPanel pnlOvest = new JPanel(new BorderLayout());
pnlOvest.add(pnlPulsanti, BorderLayout.NORTH);  // alla sua altezza preferita
add(pnlOvest, BorderLayout.WEST);
```

Il `GridLayout` dà ai pulsanti la stessa larghezza, quella del più largo; il `BorderLayout` esterno li tiene in alto alla loro altezza. Ognuno dei due layout fa la cosa che sa fare, e la combinazione fa quello che nessuno dei due farebbe da solo. Lo stesso trucco, con `SOUTH` al posto di `NORTH`, tiene in fondo una fila di pulsanti; con un `FlowLayout(FlowLayout.RIGHT)` in `SOUTH` si ottiene la classica fila "Annulla, OK" allineata a destra.

2.4.2 Spazi e misure

Due componenti attaccati l'uno all'altro, o al bordo della finestra, sono il segno più evidente di un'interfaccia trascurata. Gli spazi *fra* i componenti li danno i layout: i parametri `hgap` e `vgap` di `BorderLayout`, `FlowLayout` e `GridLayout` (i due numeri dopo righe e colonne nel `GridLayout(0, 1, 4, 4)` qui sopra). Gli spazi *attorno* a un gruppo li dà un bordo vuoto sul pannello che lo contiene, `setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8))`. Con gli stessi valori ovunque (4 pixel fra componenti vicini, 8 dal bordo) la finestra ha un aspetto ordinato senza alcuno sforzo.

Le misure, invece, si lasciano ai componenti. C'è un'eccezione: un pannello vuoto non ha una dimensione preferita sensata, perché non ha niente da misurare (un `JPanel` vuoto con il suo `FlowLayout` chiede 10 per 10 pixel, i soli margini). L'area della Lavagna è un pannello vuoto, in cui disegneremo noi, e dirle quanto vorrebbe essere grande è compito nostro:

```
pnlArea.setPreferredSize(new Dimension(640, 400));
```

È una dimensione *preferita*, non fissa: `pack` la usa per la misura iniziale della finestra, e il `BorderLayout` la ignora appena la finestra cresce, dando all'area tutto lo spazio del `CENTER`.

Attenzione 2.4: Le misure ai componenti con un testo

`setPreferredSize` su un pulsante, un'etichetta o una tendina riporta il problema del capitolo 1 dalla porta di servizio: il pulsante non misura più il suo testo, e con un carattere più grande il testo non ci sta. Si usa solo sui componenti che non hanno un contenuto da misurare, come un pannello in cui si disegna; per tutti gli altri le misure le calcola Swing.

Resta da decidere che cosa succede quando l'utente *rimpicciolisce* la finestra. Sotto la dimensione preferita i layout comprimono i componenti finché possono, poi li tagliano. Il modo più semplice per evitarlo è impedirlo: dopo `pack()`, `setMinimumSize(getSize())` fissa come dimensione minima quella appena calcolata.

2.5 Il layout assoluto e il designer

Possiamo ora dire con precisione che cosa non andava nella Lavagna del capitolo 1. La chiamata `setLayout(null)` toglie al contenitore il suo layout manager: nessuno calcola più niente, e ogni componente resta nel rettangolo che gli è stato assegnato con `setBounds`. Le conseguenze sono tre, e sono le stesse in ogni finestra costruita così.

- **Non si adatta alla finestra.** Nessuno ridistribuisce lo spazio quando la finestra cresce o si riduce.
- **Non si adatta al sistema.** I rettangoli sono misurati sui caratteri, i bordi e l'aspetto del computer su cui è stato disegnato il layout; su un altro computer i testi non ci stanno più, o la finestra taglia i componenti.
- **Non si adatta alle modifiche.** Allungare un'etichetta, aggiungere un campo, tradurre l'interfaccia in un'altra lingua: ogni cambiamento obbliga a spostare a mano i componenti vicini, e poi quelli vicini a loro.

Il designer visuale non è il colpevole. WindowBuilder, per esempio, propone il layout assoluto quando si crea una finestra, perché è quello in cui trascinare i componenti è più immediato; ma permette di assegnare a ogni contenitore uno qualunque dei layout di questo capitolo, dalla sua palette, e da quel momento i componenti si trascinano nelle regioni del `BorderLayout` o nelle celle della griglia. Usato così, è un modo comodo di vedere subito il risultato. Il codice che genera ha sempre la stessa forma: il campo `frame`, il `main` con la classe anonima e il blocco `try`, il metodo `initialize()`. Il designer rilegge quel codice per ricostruire la finestra, e conviene non stravolgerne la struttura se si vuole continuare a usarlo.

Osservazione 2.2

Le coordinate assolute hanno un posto legittimo, ma non fra i componenti: quando un pannello disegna una scacchiera, un grafico o le figure della Lavagna, le posizioni sono calcolate a ogni disegno a partire dalla dimensione attuale del pannello. È il tema del capitolo 5, ed è lo stesso principio dei layout: le misure si calcolano, non si scrivono.

2.6 L'esempio guida: la Lavagna composta

Ricostruiamo la Lavagna con tutto ciò che abbiamo visto: l'albero di pannelli della figura della sezione 2.4, nessun `setBounds`, `pack` per la dimensione iniziale, e la barra dei menu. I componenti sono campi della classe, perché nei prossimi capitoli i listener dovranno raggiungerli; il costruttore li dispone.

```
public class Lavagna extends JFrame {

    private static final String[] FORME =
            { "Rettangolo", "Ellisse", "Mano libera" };

    private final JComboBox<String> cmbForma = new JComboBox<>(FORME);
    private final JButton btnColore = new JButton("Colore...");
    private final JSlider sldSpessore = new JSlider(1, 20, 3);
    private final JPanel pnlArea = new JPanel();
    private final JLabel lblStato = new JLabel("0 figure");

    public Lavagna() {
        super("Lavagna");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setJMenuBar(creaMenu());

        JPanel pnlBarra = new JPanel(new FlowLayout(FlowLayout.LEFT, 8, 6));
        pnlBarra.add(new JLabel("Forma:"));
        pnlBarra.add(cmbForma);
        pnlBarra.add(btnColore);
        pnlBarra.add(new JLabel("Spessore:"));
        pnlBarra.add(sldSpessore);
        add(pnlBarra, BorderLayout.NORTH);

        pnlArea.setBackground(Color.WHITE);
        pnlArea.setPreferredSize(new Dimension(640, 400));
        add(pnlArea, BorderLayout.CENTER);

        lblStato.setBorder(BorderFactory.createEmptyBorder(4, 8, 4, 8));
        add(lblStato, BorderLayout.SOUTH);

        pack();
        setMinimumSize(getSize());
        setLocationRelativeTo(null);
    }
```

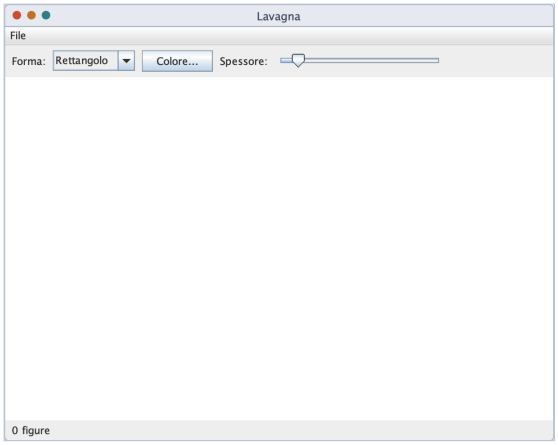
Il costruttore si legge come l'albero della figura: la barra, con il suo `FlowLayout` allineato a sinistra, a nord; l'area, l'unico componente con una dimensione preferita scritta a mano, al centro; la riga di stato, con un margine, a sud. In fondo `pack` calcola la dimensione della finestra, `setMinimumSize` la fissa come minima e `setLocationRelativeTo(null)` centra la finestra sullo schermo. Il resto della classe costruisce il menu e avvia il programma:

```
private JMenuBar creaMenu() {
    JMenu mnuFile = new JMenu("File");
    mnuFile.add(new JMenuItem("Apri..."));
    mnuFile.add(new JMenuItem("Salva"));
    mnuFile.add(new JMenuItem("Salva come..."));
    mnuFile.addSeparator();
    mnuFile.add(new JMenuItem("Esci"));

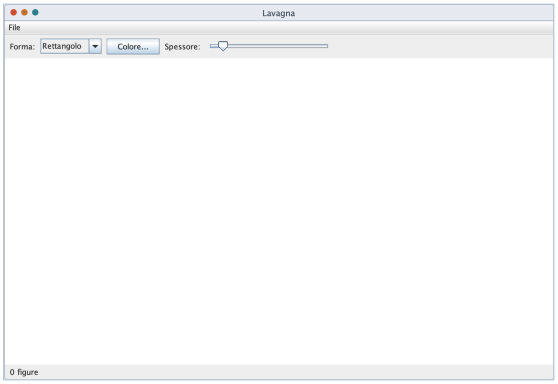
    JMenuBar barra = new JMenuBar();
    barra.add(mnuFile);
    return barra;
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new Lavagna().setVisible(true));
}
```

Il menu ha un metodo suo per non allungare il costruttore: costruire un'interfaccia produce molte righe ripetitive, e dividerle in metodi con un nome (`creaMenu`, `creaBarra`) è il modo più semplice di tenerle leggibili. Il risultato, alla dimensione iniziale e ingrandito:



La Lavagna composta, alla dimensione calcolata da `pack`.



La stessa finestra ingrandita: cresce solo l'area.

Ingrandendo la finestra la barra resta in alto con i componenti alla loro misura, la riga di stato resta in fondo, e tutto lo spazio in più va all'area, che è quello che volevamo. Sul sistema con i bordi di Windows, o con caratteri più grandi, la finestra iniziale è semplicemente un po' più grande: nessun componente viene tagliato, perché nessuna misura è stata presa su un altro sistema. La Lavagna non fa ancora niente: la tendina si apre, lo slider si sposta, ma nessuno ascolta. È il compito del prossimo capitolo.

Ricetta: Comporre una finestra

1. Disegnare a mano la finestra e chiedersi, per ogni parte, che cosa succede quando la finestra cresce: la parte che deve crescere è il `CENTER`.
2. Mettere attorno al `CENTER`, nelle regioni di un `BorderLayout`, le parti di misura fissa: barre a nord e a sud, colonne a ovest e a est.
3. Per ogni parte un `JPanel` con il layout adatto al suo contenuto: una fila è un `FlowLayout`, una griglia o una colonna di elementi uguali un `GridLayout`; se una parte ha a sua volta parti diverse, si ripete dal punto 1.
4. Niente `setBounds` né `setSize`: margini con bordi vuoti e spazi dei layout, `setPreferredSize` solo sui pannelli in cui si disegna, `pack()` alla fine; poi provare la finestra ingrandendola e rimpicciolendola.

Esercizi

Esercizio 2.1: Che cosa si vede (⋆)

Disegnare a mano la finestra prodotta da questo codice, indicando quali pulsanti si vedono e dove. Che cosa cambia ingrandendola?

```
JFrame f = new JFrame("Prova");
f.add(new JButton("A"));
f.add(new JButton("B"));
f.add(new JButton("C"), BorderLayout.SOUTH);
f.add(new JButton("D"), BorderLayout.EAST);
f.pack();
f.setVisible(true);
```

Esercizio 2.2: Un pannello, tre pulsanti (⋆)

Un `JPanel` creato con `new JPanel()` riceve tre pulsanti e viene messo nel `CENTER` di una finestra. Come sono disposti i pulsanti? Che cosa succede quando si restringe la finestra fino a metà della larghezza dei tre pulsanti? E se il pannello fosse in `NORTH`?

Esercizio 2.3: Righe e colonne (⋆)

Come vengono disposti sette pulsanti in un `GridLayout(2, 3)`? E in un `GridLayout(0, 3)`? E in un `GridLayout(3, 0)`? Per ciascun caso dire quante righe e quante colonne si vedono e quante celle restano vuote.

Esercizio 2.4: La finestra che non si chiude (⋆)

Un programma costruisce una finestra e la mostra, senza aver mai chiamato `setDefaultCloseOperation`. L'utente preme la X. Che cosa vede? Il programma è terminato? Come si dimostra, e che cosa cambia con `DISPOSE_ON_CLOSE`?

Esercizio 2.5: L'area minuscola (⋆⋆)

Dalla Lavagna di questo capitolo si toglie la riga `pnlArea.setPreferredSize(new Dimension(640, 400))`. Quanto è alta l'area nella finestra che `pack` produce, e perché? Che cosa succede quando l'utente ingrandisce la finestra? Perché lo stesso problema non si presenta con la barra?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 2.6: La calcolatrice in finestra (⋆)

Scrivere la finestra di una calcolatrice, solo la composizione: in alto un campo di testo non modificabile, con il testo allineato a destra e un carattere grande, che fa da display; sotto una tastiera di sedici pulsanti in quattro righe (`7 8 9 /`, `4 5 6 *`, `1 2 3 -`, `0 . = +`). Ingrandendo la finestra il display deve allargarsi senza diventare più alto, e i pulsanti devono crescere tutti allo stesso modo. La dimensione iniziale la decide `pack`.

Soluzione: progetto `sol-d05-c02`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 2.5: I pulsanti in un ciclo

Sedici `add` scritti a mano sono sedici occasioni di sbagliare l'ordine. Le etichette stanno meglio in un array di stringhe, nell'ordine della griglia, e un ciclo crea e aggiunge i pulsanti: il `GridLayout` li dispone riga per riga nell'ordine in cui arrivano.

Esercizio 2.7: Un modulo ben allineato (⋆⋆)

Scrivere la finestra di un modulo di iscrizione con quattro campi (nome, cognome, email, anno di nascita), ciascuno preceduto dalla sua etichetta, e in fondo a destra i pulsanti "Annulla" e "Iscrivi". Le etichette devono essere allineate fra loro e i campi anche; ingrandendo la finestra i campi si allargano ma non si allungano in altezza, e i pulsanti restano in fondo a destra.
Suggerimento: una colonna di etichette e una colonna di campi, ciascuna un `GridLayout(0, 1)`, affiancate in un pannello con `BorderLayout` (etichette a ovest, campi al centro); quel pannello va nel `NORTH` di un altro, perché non cresca in altezza.

Soluzione: progetto `sol-d05-c02`, package `it.unibs.pajc.redolfi.e07`.

Esercizio 2.8: Il blocco note (⋆⋆)

Comporre la finestra di un blocco note. A sinistra l'elenco dei titoli delle note (una `JList` con qualche titolo d'esempio, in uno scroll pane), con sotto i pulsanti "Nuova" ed "Elimina" affiancati e della stessa larghezza. A destra, in alto, l'etichetta "Titolo:" e un campo di testo sulla stessa riga; sotto, il testo della nota (una `JTextArea` che va a capo fra le parole, in uno scroll pane), che occupa tutto lo spazio rimasto. In fondo una riga di stato. Ingrandendo la finestra cresce soprattutto il testo; l'elenco cresce solo in altezza. Niente deve far nulla: si compone soltanto.

Soluzione: progetto `sol-d05-c02`, package `it.unibs.pajc.redolfi.e08`.

Attenzione 2.6: L'elenco che si allarga

Un elenco messo a `WEST` è largo quanto la sua voce più lunga: un titolo lungo allarga tutta la colonna e ruba spazio al testo. Una larghezza preferita stabile la dà `setPrototypeCellValue` con una stringa della lunghezza voluta, oppure `setPreferredSize` sullo scroll pane, che non ha un testo suo da misurare.

Esercizio 2.9: Il pannello di controllo (⋆⋆⋆)

Comporre una finestra con: una barra dei menu con i menu "File" (Apri, Salva, Esci) e "Aiuto" (Informazioni); a sinistra una colonna di cinque pulsanti di uguale larghezza che resta in alto, e in fondo alla stessa colonna un riquadro colorato alto 40 pixel; al centro un registro (`JTextArea` non modificabile, carattere a spaziatura fissa, in uno scroll pane); in fondo una barra di stato con un testo a sinistra e uno a destra. Margini di 8 pixel dal bordo e 4 fra componenti vicini. Verificare poi che la finestra regga caratteri più grandi del 50%: prima di costruirla, ingrandire tutti i font predefiniti di Swing.
Suggerimento: i font predefiniti stanno nella tabella di `UIManager`; un ciclo su `UIManager.getDefaults().keys()` che sostituisce ogni valore di tipo `FontUIResource` con uno più grande, chiamato nel `main` prima di creare la finestra, cambia i caratteri di tutti i componenti.

Soluzione: progetto `sol-d05-c02`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 2.1. Si vedono B al centro, C sotto e D a destra; A non si vede. Il content pane ha un `BorderLayout`: A e B, aggiunti senza regione, vanno entrambi in `CENTER`, e B prende il posto di A, che resta nel contenitore ma non viene più disposto. C occupa tutta la larghezza in fondo, alla sua altezza; D sta a destra, alla sua larghezza, fra il bordo superiore e C. Ingrandendo la finestra cresce soltanto B, in entrambe le direzioni; C si allarga e D si allunga, ma restano della loro altezza e larghezza.

Risposta 2.2. Un `JPanel` nasce con un `FlowLayout` centrato: i tre pulsanti stanno in fila, alla loro misura, centrati in orizzontale e in alto nel pannello. Restringendo la finestra il `FlowLayout` va a capo: il terzo pulsante (o il secondo e il terzo) passa su una nuova riga, sempre centrata. In `NORTH` il pannello riceve la sua altezza preferita, che il `FlowLayout` calcola come se i pulsanti stessero tutti su una riga: quando vanno a capo, la seconda riga non ha spazio e viene tagliata. È un limite noto del `FlowLayout` nelle regioni di misura preferita.

Risposta 2.3. Con entrambi i numeri diversi da zero il `GridLayout` rispetta le righe e calcola le colonne: `GridLayout(2, 3)` con sette componenti diventa 2 righe per 4 colonne ($\lceil 7/2 \rceil = 4$), con una cella vuota. `GridLayout(0, 3)`: 3 colonne e $\lceil 7/3 \rceil = 3$ righe, due celle vuote nell'ultima riga. `GridLayout(3, 0)`: 3 righe e $\lceil 7/3 \rceil = 3$ colonne, due celle vuote. Le colonne passate al costruttore contano solo quando le righe sono zero; per questo, per una colonna o una riga di lunghezza variabile, si scrive `GridLayout(0, 1)` o `GridLayout(1, 0)`.

Risposta 2.4. La finestra sparisce, e l'utente pensa che il programma sia finito. Non lo è: il comportamento predefinito è `HIDE_ON_CLOSE`, la finestra è solo nascosta, e il thread degli eventi di Swing tiene in vita la JVM. Lo dimostra la console di Eclipse, con il quadratino rosso ancora acceso, oppure una stampa dopo il `setVisible` di una finestra nascosta che non arriva mai a un "fine". Con `DISPOSE_ON_CLOSE` la finestra libera le sue risorse; se era l'ultima finestra aperta e nessun altro thread non daemon è in corso, la JVM termina da sola.

Risposta 2.5. L'area è alta 10 pixel: un `JPanel` vuoto con il suo `FlowLayout` ha come dimensione preferita i soli margini del layout, 5 pixel per lato. `pack` somma l'altezza della barra, dei 10 pixel dell'area e della riga di stato, e la finestra esce bassissima; la larghezza la decide la barra. Ingrandendo la finestra l'area cresce normalmente, perché il `CENTER` prende tutto lo spazio che avanza: il problema è solo la dimensione iniziale, e la minima, che `setMinimumSize` fissa a quella sbagliata. La barra non ne soffre perché contiene componenti con un testo, che sanno misurarsi.

Capitolo 3

Eventi, listener e modelli

Alla fine del capitolo 2 la Lavagna aveva la forma giusta e nessuna reazione: la tendina si apriva, lo slider si spostava, e il programma non se ne accorgeva. Questo capitolo la fa reagire. Si parte dal meccanismo con cui un componente avvisa il programma che è successo qualcosa, che è lo stesso in tutta la libreria; si passano in rassegna gli eventi che servono davvero; poi si guarda dentro i componenti che mostrano dei dati, le tendine e le liste, e si scopre che lo stesso meccanismo li tiene aggiornati. Alla fine scriviamo un componente nostro, con i suoi eventi, e la barra della Lavagna funziona: si sceglie la forma, il colore e lo spessore, e la riga di stato lo dice.

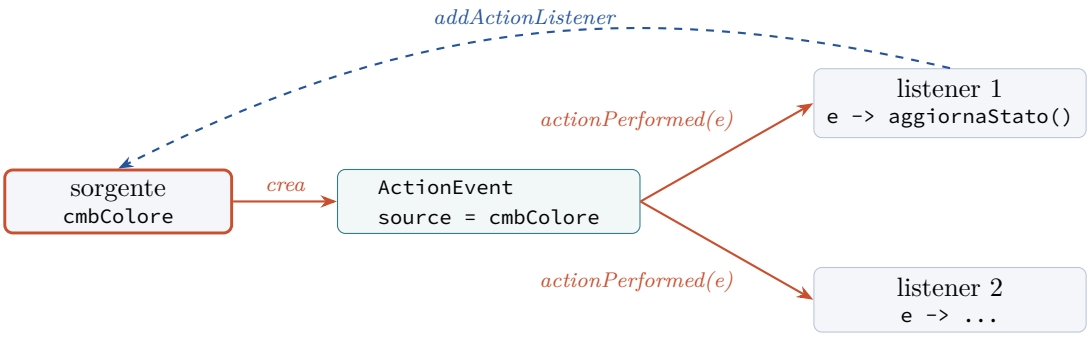
3.1 Chi avvisa chi

La Lavagna deve aggiornare la riga di stato quando l'utente sceglie un colore. Chi chiama chi? La tendina non può chiamare un metodo della Lavagna: è stata scritta anni prima, senza sapere che la Lavagna sarebbe esistita. La Lavagna non può chiedere alla tendina, a intervalli regolari, se è cambiato qualcosa: sprecherebbe tempo e risponderebbe in ritardo. La soluzione è rovesciare la dipendenza. La tendina tiene un elenco di oggetti interessati ai suoi cambiamenti, senza sapere chi siano, purché abbiano un metodo da chiamare; la Lavagna, se le interessa, si iscrive all'elenco. Quando la scelta cambia, la tendina chiama quel metodo su tutti gli iscritti.

È un'idea abbastanza generale da avere un nome: il pattern *Observer*, in cui un oggetto osservato avvisa gli osservatori che si sono registrati presso di lui. Swing lo applica a ogni componente, con un vocabolario suo.

Definizione 3.1: Sorgente, evento, listener

Nel *modello a delega degli eventi* di Swing la *sorgente* è il componente in cui succede qualcosa; l'*evento* è un oggetto che descrive che cosa è successo (chi l'ha generato, quando, con quali dati); il *listener* è un oggetto che implementa un'interfaccia `XxxListener` e si è registrato presso la sorgente con `addXxxListener`. Quando l'evento accade, la sorgente chiama il metodo dell'interfaccia su ogni listener registrato, passandogli l'evento.



I listener si registrano presso la sorgente; quando accade qualcosa, la sorgente crea un evento e lo passa a ciascuno.

L'interfaccia più usata è `ActionListener`, con il solo metodo `actionPerformed(ActionEvent e)`: la chiamano i pulsanti quando vengono premuti, le voci di menu quando vengono scelte, le tendine quando cambia la selezione, i campi di testo quando si preme Invio. Avendo un solo metodo astratto è un'interfaccia funzionale, e il listener si può scrivere in tre modi, gli stessi della scala che porta dalla classe anonima alla lambda (*Dal nome alla funzione*):

```
// 1. una classe anonima: la forma che genera il designer
cmbColore.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        aggiornaStato();
    }
});

// 2. una lambda: il parametro c'e', anche se non lo usiamo
cmbColore.addActionListener(e -> aggiornaStato());

// 3. un method reference: void coloreCambiato(ActionEvent e)
cmbColore.addActionListener(this::coloreCambiato);
```

Le tre righe fanno la stessa cosa. La lambda è la forma più comune, perché adatta qualunque metodo all'interfaccia: `aggiornaStato()` non riceve parametri, e la lambda ignora l'evento. Il method reference richiede invece un metodo con la stessa forma di `actionPerformed`, cioè che riceva un `ActionEvent`; quando il metodo c'è già con quella forma è il più leggibile. La classe anonima resta utile solo quando il listener ha bisogno di un suo stato, come il punto di partenza del listener del mouse del capitolo 1.

Chi esegue il listener è il thread degli eventi di Swing, quello che svuota la coda degli eventi: il metodo gira *dentro* il ciclo dell'immagine guida, e finché non termina la finestra non si ridisegna e non risponde. Un listener deve quindi fare poco e tornare subito: aggiornare lo stato, chiedere un ridisegno, al più aprire una finestra di dialogo. I lavori lunghi vanno altrove, come spiega *Thread e concorrenza*.

3.2 Un listener per molti componenti

La tastiera di una calcolatrice ha sedici pulsanti, e scrivere sedici listener quasi uguali sarebbe assurdo. Un listener è un oggetto come gli altri, e lo stesso oggetto può essere registrato su più sorgenti; dall'evento ricava chi lo ha generato. Per i pulsanti c'è un'informazione ancora più comoda, il *comando*, che per un `JButton` è, se non lo si cambia, il testo del pulsante:

```
ActionListener tasto =
        e -> txtDisplay.setText(txtDisplay.getText() + e.getActionCommand());

for (String t : TASTI) {
    JButton b = new JButton(t);
    b.addActionListener(tasto);          // lo stesso listener per tutti
    pnlTastiera.add(b);
}
```

Quando il comando non basta, `e.getSource()` restituisce la sorgente come `Object`: la si confronta con `==` con i componenti noti, o la si converte con un `instanceof`. È il modo

di distinguere due pulsanti con lo stesso testo, o di leggere il valore del componente che ha generato l'evento senza sapere quale fosse.

C'è poi un caso più frequente di quanto sembri, in cui non serve sapere chi ha generato l'evento. Nella Lavagna la riga di stato riassume forma, colore e spessore: che sia cambiato il colore o lo spessore, la cosa da fare è la stessa, rileggere tutti e tre i componenti e riscrivere la riga. Un solo metodo `aggiornaStato()`, chiamato da tutti i listener, è più semplice e più robusto di tre metodi che aggiornano ciascuno un pezzo, perché non può dimenticare niente. Lo si chiama anche alla fine del costruttore, così la riga di stato è giusta fin dall'inizio.

Attenzione 3.1: Registrato due volte, rimosso mai

Un listener registrato due volte viene chiamato due volte: succede quando la registrazione sta in un metodo che viene eseguito più volte (un `aggiorna()`, un pulsante "Reimposta"), e il sintomo è un'azione che si ripete, poi si ripete tre volte, poi quattro. E per togliere un listener con `removeActionListener` bisogna passare *lo stesso oggetto* che si era registrato:

```
btnVai.addActionListener(e -> vai());
btnVai.removeActionListener(e -> vai());   // lambda nuova: non toglie
```

Ogni lambda valutata crea un oggetto nuovo, e due lambda scritte uguali non sono lo stesso oggetto. Un listener che si dovrà togliere si tiene in un campo: `ActionListener vai = e -> vai();` e poi `add` e `remove` con quel campo.

3.3 Eventi semantici e di basso livello

Un pulsante si può premere in tanti modi: con un click del mouse, con la barra spaziatrice quando ha il fuoco, con Invio se è il pulsante predefinito della finestra, con una scorciatoia da tastiera. Per il programma è sempre la stessa cosa: il pulsante è stato attivato. Swing distingue quindi due livelli di eventi. Gli eventi *di basso livello* descrivono ciò che fanno i dispositivi: il pulsante del mouse premuto e rilasciato in un punto, un tasto premuto, il fuoco che passa da un componente all'altro. Gli eventi *semantici* descrivono che cosa significa per il componente: il pulsante attivato, la selezione cambiata, il valore modificato. Il componente traduce i primi nei secondi.

Attenzione 3.2: Il click non è l'azione

Un `MouseListener` su un `JButton` ascolta il mouse, non il pulsante: chi preme il pulsante con la tastiera non lo attiva, e un click che parte sul pulsante e finisce fuori, che per il pulsante non è una pressione, per il listener lo è. Sui componenti si ascoltano gli eventi semantici; quelli di basso livello servono sui pannelli che si disegnano da soli, dove non c'è nessun componente a tradurli (capitolo 8).

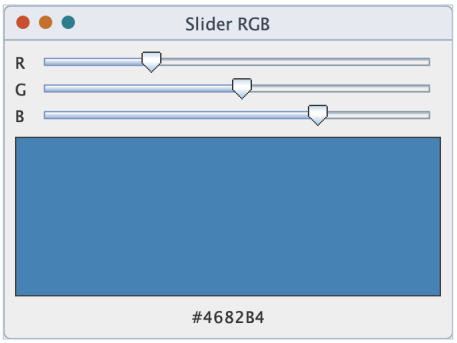
I listener che servono quasi sempre sono pochi. La tabella dice quali sono, che cosa ascoltano e chi li genera.

Interfaccia	Metodo	Chi lo genera, e quando
`ActionListener`	`actionPerformed`	pulsanti, voci di menu, tendine, Invio in un campo
`ChangeListener`	`stateChanged`	`JSlider`, `JSpinner`: il valore cambia
`ItemListener`	`itemStateChanged`	caselle e tendine: una voce scelta o lasciata
`ListSelectionListener`	`valueChanged`	`JList`: la selezione cambia
`DocumentListener`	tre metodi	il testo di un campo cambia (`getDocument()`)
`MouseListener`	cinque metodi	ogni componente: pressione, rilascio, click
`MouseMotionListener`	due metodi	ogni componente: movimento, trascinamento
`KeyListener`	tre metodi	il componente che ha il fuoco: tasti
`WindowListener`	sette metodi	le finestre: apertura, chiusura, icona

Le interfacce sopra la riga sono semantiche, quelle sotto di basso livello; mouse e tastiera sono il tema del capitolo 8, le finestre del capitolo 4. Le interfacce con più metodi non si possono scrivere come lambda, e implementarle obbligherebbe a scrivere cinque o sette metodi per usarne uno. Per questo ciascuna ha una classe *adapter* (`MouseAdapter`, `KeyAdapter`, `WindowAdapter`) che le implementa tutte con metodi vuoti: si estende l'adapter, di solito con una classe anonima, e si ridefinisce solo il metodo che interessa. Sono classi e non metodi `default` perché sono nate prima che le interfacce potessero avere metodi con un corpo.

3.3.1 Tre slider, un metodo

Gli slider generano un `ChangeEvent` a ogni spostamento del cursore, anche mentre il mouse è ancora premuto. L'esempio classico è il colore composto con tre slider, uno per ciascuna componente rossa, verde e blu, che colorano un pannello e scrivono il codice esadecimale del colore:



Tre slider, un pannello e un'etichetta: il colore si compone spostando i cursori.

Il programma segue la regola della sezione precedente: un solo metodo rilegge i tre valori e aggiorna tutto, e i tre listener lo chiamano.

```
sldR.addChangeListener(e -> aggiorna());      // un solo metodo per tutti e tre
sldG.addChangeListener(e -> aggiorna());
sldB.addChangeListener(e -> aggiorna());
aggiorna();                                   // e una volta all'inizio

private void aggiorna() {
    Color c = new Color(sldR.getValue(), sldG.getValue(), sldB.getValue());
    pnlColore.setBackground(c);
    lblEsadecimale.setText(String.format("#%02X%02X%02X",
            c.getRed(), c.getGreen(), c.getBlue()));
}
```

Osservazione 3.1

Gli eventi partono anche quando a cambiare il valore è il programma: `setValue` su uno slider genera un `ChangeEvent` come se l'utente avesse spostato il cursore. Se il programma imposta i tre slider uno dopo l'altro, per esempio leggendo un colore da un campo di testo, `aggiorna()` viene chiamato tre volte, e le prime due con un colore a metà. Di solito non è un problema; lo diventa quando il listener fa un lavoro costoso, o quando due componenti si aggiornano a vicenda. Per il primo caso `getValueIsAdjusting()` dice se il cursore è ancora trascinato, e si può aspettare il rilascio; per il secondo serve una bandiera che dica "questo cambiamento lo sto facendo io", e che il listener controlla prima di reagire.

3.4 Dati veri nei componenti: oggetti e modelli

Nella Lavagna del capitolo 2 la tendina conteneva stringhe. Per una tendina di colori, con stringhe, il programma dovrebbe a ogni scelta tradurre il nome in un colore, con uno `switch` che va aggiornato ogni volta che si aggiunge un colore e che non sa che cosa fare con un nome sbagliato. Una tendina può contenere oggetti qualunque, e mostra di ciascuno ciò che restituisce `toString()`. Basta quindi un tipo che tenga insieme il nome e il colore: è un valore, e diventa un record (*Java Fast Forward*).

```
public record Colore(String nome, Color colore) {

    public static final Colore[] PALETTE = {
            new Colore("Nero", Color.BLACK),
            new Colore("Blu", new Color(0x24, 0x54, 0x9C)),
            new Colore("Rosso", new Color(0xC9, 0x4F, 0x2E)),
            new Colore("Verde", new Color(0x2E, 0x7D, 0x4C)),
            new Colore("Arancio", new Color(0xE0, 0x8A, 0x1E)) };

    @Override
    public String toString() {
        return nome;                    // quello che la tendina mostra
    }
}
```

La tendina diventa una `JComboBox<Colore>`, costruita con la palette, e chi reagisce alla selezione riceve l'oggetto intero: `(Colore) cmbColore.getSelectedItem()`, e poi `.colore()`. Niente traduzioni, niente nomi sbagliati possibili. Il `toString` ridefinito serve solo alla tendina: senza, si vedrebbe quello generato dal record, `Colore[nome=Blu, colore=java.awt.Color[...]]`, con le tre componenti del colore.

3.4.1 Il modello

Dove stanno le voci di una tendina? Non nella tendina. Una `JComboBox` legge le sue voci da un oggetto separato, il suo *modello*, e lo stesso vale per una `JList`, una `JTable`, uno slider, un campo di testo. Il costruttore che riceve un array crea un modello e ci copia dentro le voci.

Definizione 3.2: Modello di un componente

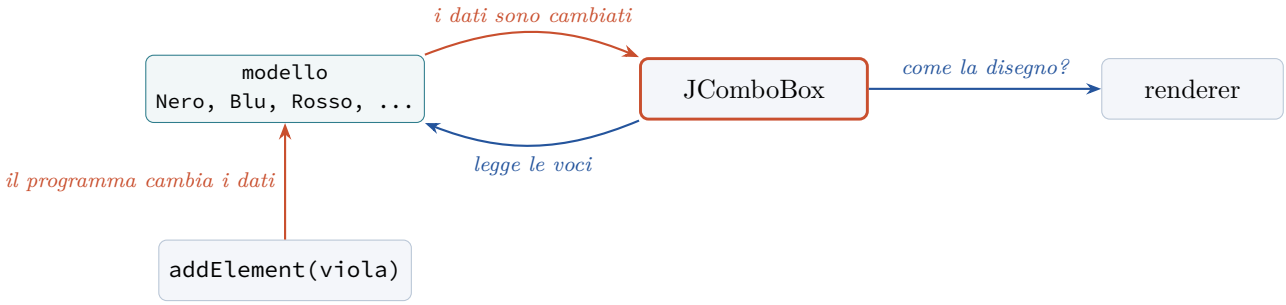
Il *modello* di un componente Swing è l'oggetto che contiene i dati che il componente mostra: le voci di una tendina (`ComboBoxModel`) o di una lista (`ListModel`), il valore di uno slider (`BoundedRangeModel`), il testo di un campo (`Document`). Il componente è un listener del suo modello: quando i dati cambiano, il modello lo avvisa, e il componente si ridisegna.

È l'immagine guida in piccolo: il modello è lo stato, il componente lo disegna. La conseguenza pratica è che per cambiare ciò che una tendina mostra non si tocca la tendina, si tocca il suo modello. Le implementazioni pronte sono `DefaultComboBoxModel` e `DefaultListModel`, con metodi che ricordano quelli di una lista:

```
var mdlColori = new DefaultComboBoxModel<Colore>(Colore.PALETTE);
JComboBox<Colore> cmbColore = new JComboBox<>(mdlColori);

Colore viola = new Colore("Viola", new Color(0x6A, 0x4C, 0x93));
mdlColori.addElement(viola);            // la tendina mostra subito la voce nuova
mdlColori.setSelectedItem(viola);       // e la seleziona
```

Il codice non chiama niente sulla tendina: aggiunge un elemento al modello, il modello avvisa i suoi listener, e fra questi c'è la tendina, che si aggiorna. Lo stesso modello può stare dietro a più componenti: una tendina e una lista costruite con lo stesso `DefaultComboBoxModel` mostrano sempre le stesse voci, e un `addElement` le aggiorna entrambe.



Il componente legge i dati dal modello, che lo avvisa quando cambiano, e chiede al renderer come disegnare ogni voce.

Attenzione 3.3: L'array non è il modello

`new JComboBox<>(Colore.PALETTE)` copia le voci in un modello nuovo: modificare poi l'array non cambia la tendina. Per una tendina o una lista il cui contenuto cambia durante il programma si crea il modello esplicitamente, lo si tiene in un campo e si lavora su quello. Con `JList` vale lo stesso: `new JList<>(array)` crea un modello che non si può modificare, e le liste che crescono usano un `DefaultListModel`.

3.5 Come appare una voce: il renderer

Una tendina di colori sarebbe più chiara se accanto a ogni nome ci fosse il colore. Il `toString` decide il testo, non l'aspetto; per l'aspetto una lista o una tendina consultano un altro oggetto, il *renderer*. È un `ListCellRenderer`, con un solo metodo, che riceve la voce da disegnare e restituisce il componente con cui disegnarla. La lista non aggiunge quel componente a se stessa: lo usa come un *timbro*, lo dipinge nella posizione di una riga, gli cambia testo e colori, lo dipinge nella riga successiva. Un solo componente basta per tutte le righe, che siano dieci o diecimila.

Il renderer predefinito, `DefaultListCellRenderer`, è una `JLabel` che mostra il `toString` della voce e i colori della selezione. Il modo più semplice di scriverne uno è estenderlo: si lascia che la superclasse prepari l'etichetta, e si aggiunge ciò che manca. Qui manca un'icona, cioè un oggetto `Icon` che sappia dipingere un quadratino colorato; ha due misure e un colore, ed è anche lei un valore:

```
public record IconaColore(Color colore, int lato) implements Icon {

    @Override
    public void paintIcon(Component c, Graphics g, int x, int y) {
        g.setColor(colore);
        g.fillRect(x, y, lato, lato);
        g.setColor(Color.DARK_GRAY);
        g.drawRect(x, y, lato - 1, lato - 1);
    }

    @Override
    public int getIconWidth() { return lato; }

    @Override
    public int getIconHeight() { return lato; }
}
```

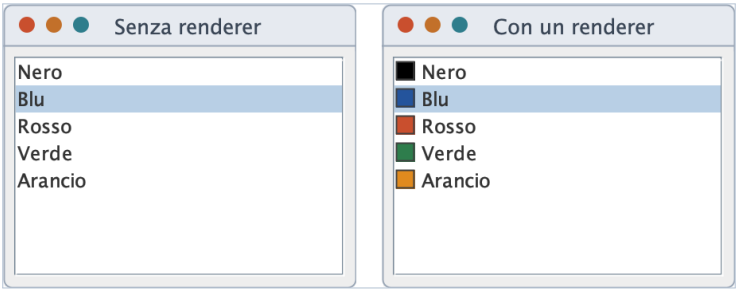
`paintIcon` è il primo metodo di questa dispensa che disegna con un contesto grafico `Graphics`: il capitolo 5 lo spiega per intero, qui basta che `fillRect` riempie un rettangolo e `drawRect` ne disegna il contorno. Il renderer usa l'icona:

```
public class ColoreRenderer extends DefaultListCellRenderer {

    @Override
    public Component getListCellRendererComponent(JList<?> list, Object value,
            int index, boolean isSelected, boolean cellHasFocus) {
        super.getListCellRendererComponent(list, value, index, isSelected,
                cellHasFocus);
        if (value instanceof Colore c)
            setIcon(new IconaColore(c.colore(), 12));
        return this;
    }
}
```

La chiamata a `super` fa tutto il lavoro solito: testo dal `toString`, colori di sfondo e di primo piano a seconda che la voce sia selezionata, bordo del fuoco, icona azzerata. Noi aggiungiamo l'icona e restituiamo `this`, cioè l'etichetta stessa, pronta per essere dipinta. Il renderer si

assegna con `setRenderer` a una tendina e con `setCellRenderer` a una lista; per la tendina viene chiamato anche per la voce mostrata a tendina chiusa, con `index` uguale a -1.



La stessa lista di `Colore`, con il renderer predefinito e con il nostro.

Attenzione 3.4: Il timbro ricorda

Il renderer è un solo oggetto riusato per tutte le righe: ciò che si imposta per una riga resta impostato per le successive, finché qualcuno non lo cambia. La chiamata a `super` rimette a posto testo, colori, icona e font, ma non ciò che aggiungiamo noi: un renderer che per alcune voci imposta un bordo o un suggerimento (`setToolTipText`) deve reimpostarli, a un valore neutro, anche per tutte le altre, altrimenti "passano" alle righe sotto. E il renderer viene chiamato a ogni ridisegno, per ogni riga visibile: niente lavoro costoso, e niente oggetti pesanti creati ogni volta (un `Font` per riga si tiene in una mappa).

3.6 Un componente nostro, con i suoi eventi

Per scegliere la forma la Lavagna vuole tre pulsanti affiancati, di cui uno solo premuto alla volta, come in molte barre degli strumenti. Swing li ha già: i `JToggleButton` restano premuti dopo il click, e un `ButtonGroup` garantisce che ne sia premuto uno solo. Ma alla finestra non interessano tre pulsanti: le interessa sapere qual è la forma scelta, ed essere avvisata quando cambia. È il momento di scrivere un componente nostro, che nasconda i tre pulsanti e offra quello che serve a chi lo usa: un metodo `getForma()` e un evento, "la forma è cambiata".

Le forme sono un insieme chiuso, quindi un `enum`, con l'etichetta da mostrare:

```
public enum Forma {
    RETTANGOLO("Rettangolo"), ELLISSE("Ellisse"), MANO_LIBERA("Mano libera");

    private final String etichetta;

    Forma(String etichetta) { this.etichetta = etichetta; }

    public String etichetta() { return etichetta; }
}
```

Il componente è un `JPanel` che contiene i pulsanti, uno per forma, e tiene la forma corrente in un campo. Ogni pulsante, quando viene premuto, chiama `scegli`:

```
public class SelettoreForma extends JPanel {

    private Forma forma = Forma.RETTANGOLO;

    public SelettoreForma() {
        super(new GridLayout(1, 0));
        ButtonGroup gruppo = new ButtonGroup();  // uno solo premuto
        for (Forma f : Forma.values()) {
            JToggleButton b = new JToggleButton(f.etichetta(), f == forma);
            b.addActionListener(e -> scegli(f));
            gruppo.add(b);
            add(b);
        }
    }

    public Forma getForma() {
        return forma;
    }

    private void scegli(Forma f) {
        if (f == forma)
            return;                     // niente di nuovo, niente evento
        forma = f;
        fireActionPerformed();
    }
```

Resta da far partire l'evento. Il componente deve fare ciò che fanno i componenti di Swing: tenere un elenco di listener, offrire i metodi per iscriversi e cancellarsi, e avvisarli tutti quando la forma cambia. La prima versione si scrive con una lista:

```
private final List<ActionListener> listeners = new ArrayList<>();

public void addActionListener(ActionListener l) { listeners.add(l); }

public void removeActionListener(ActionListener l) { listeners.remove(l); }

protected void fireActionPerformed() {
    var e = new ActionEvent(this, ActionEvent.ACTION_PERFORMED, forma.name());
    for (ActionListener l : listeners)
        l.actionPerformed(e);
}
```

L'evento ha come sorgente il componente stesso (`this`), un tipo (`ACTION_PERFORMED`) e, come comando, il nome della forma: un listener può leggerla da lì o chiederla con `getForma()`. I nomi dei metodi non sono scelti a caso: `addXxxListener`, `removeXxxListener` e `fireXxx` sono la convenzione di tutta la libreria, e chi usa il componente lo riconosce come un componente Swing.

La lista ha un difetto che si scopre tardi. Se un listener, durante la notifica, si toglie dall'elenco (capita: "avvisami la prima volta e basta"), il ciclo `for` sta scorrendo una lista che cambia sotto di lui, e lancia una `ConcurrentModificationException`. Swing ha una classe fatta per questo, `EventListenerList`, che è quella usata dai suoi componenti: tiene listener

di più tipi in un solo oggetto e, al momento della notifica, ne restituisce una copia, così che l'elenco si possa modificare mentre si notifica.

```
private final EventListenerList listeners = new EventListenerList();

public void addActionListener(ActionListener l) {
    listeners.add(ActionListener.class, l);
}

public void removeActionListener(ActionListener l) {
    listeners.remove(ActionListener.class, l);
}

protected void fireActionPerformed() {
    var e = new ActionEvent(this, ActionEvent.ACTION_PERFORMED, forma.name());
    for (ActionListener l : listeners.getListeners(ActionListener.class))
        l.actionPerformed(e);
}
```

Chi usa `SelettoreForma` non vede nessuna differenza fra le due versioni, ed è proprio il punto: un componente con i suoi eventi si usa come i componenti della libreria, `selForma.addActionListener(e -> aggiornaStato())`, e ciò che contiene resta affar suo. Domani i tre pulsanti potrebbero diventare una tendina o tre icone, e la Lavagna non se ne accorgerebbe.

Osservazione 3.2

Abbiamo scritto a mano il pattern Observer della sezione 3.1: una sorgente, un elenco di osservatori, una notifica. È lo stesso schema dei modelli della sezione 3.4, che avvisano i loro componenti, e lo stesso che si usa fuori dalle interfacce grafiche ogni volta che un oggetto deve far sapere ad altri che è cambiato senza dipendere da loro. Il controllo `if (f == forma) return` non è un dettaglio: un evento "cambiato" che parte quando non è cambiato niente è il modo più rapido per creare cicli di aggiornamenti fra componenti che si ascoltano a vicenda.

3.7 L'esempio guida: la barra che funziona

La Lavagna di questo capitolo ha nella barra il selettore della forma, la tendina dei colori con il suo renderer, un pulsante "Altro…" per i colori fuori dalla palette (lo collegheremo nel capitolo 4) e lo slider dello spessore. I campi cambiano così:

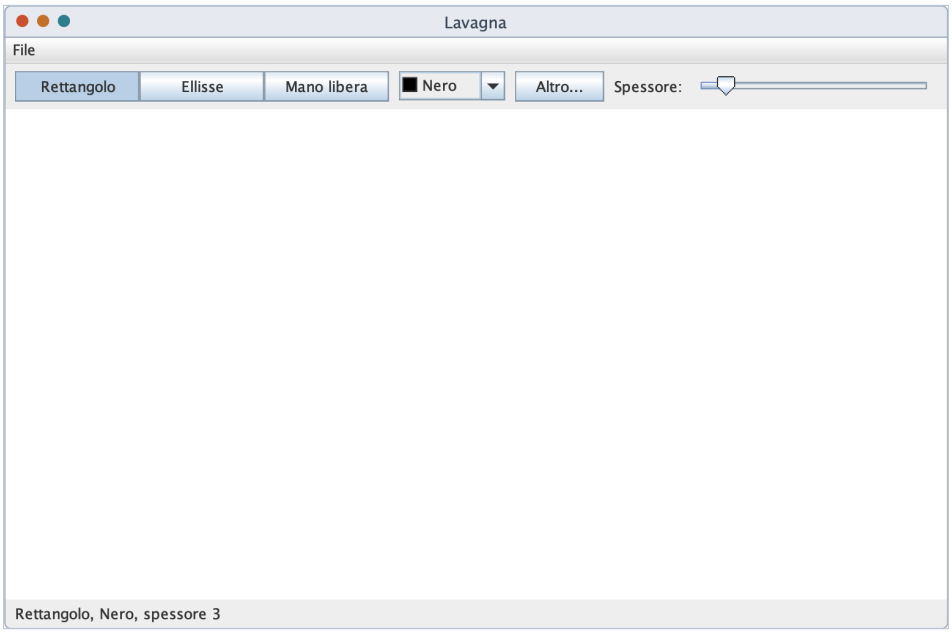
```
private final SelettoreForma selForma = new SelettoreForma();
private final JComboBox<Colore> cmbColore = new JComboBox<>(Colore.PALETTE);
private final JButton btnColore = new JButton("Altro...");
private final JSlider sldSpessore = new JSlider(1, 20, 3);
private final JPanel pnlArea = new JPanel();
private final JLabel lblStato = new JLabel();
```

Nel costruttore, dopo la composizione del capitolo 2, si assegna il renderer e si registrano i listener; tutti chiamano lo stesso metodo, che è anche l'ultima cosa che il costruttore fa prima di `pack`:

```
cmbColore.setRenderer(new ColoreRenderer());
selForma.addActionListener(e -> aggiornaStato());
cmbColore.addActionListener(e -> aggiornaStato());
sldSpessore.addChangeListener(e -> aggiornaStato());
aggiornaStato();
```

```
private void aggiornaStato() {
    Colore c = (Colore) cmbColore.getSelectedItem();
    lblStato.setText(selForma.getForma().etichetta() + ", " + c.nome()
            + ", spessore " + sldSpessore.getValue());
}
```

Tre sorgenti diverse, tre tipi di listener (un `ActionListener` sul nostro componente, uno sulla tendina, un `ChangeListener` sullo slider), un solo metodo. La riga di stato ora segue ogni scelta:



La Lavagna con la barra che funziona: il selettore della forma, la tendina dei colori con il renderer, lo spessore.

Per ora la riga di stato è l'unico posto in cui le scelte si vedono. Lo stato vero, cioè la forma, il colore e lo spessore con cui disegnare, sta ancora nei componenti, e `aggiornaStato` lo legge da lì; quando nel capitolo 5 l'area comincerà a disegnare, sarà lei a chiederli.

Ricetta: Far reagire una finestra

1. Decidere che cosa deve cambiare quando l'utente agisce: una riga di stato, un pannello, l'abilitazione di un pulsante.
2. Scrivere un metodo che lo ricalcola *tutto* dai valori dei componenti, senza chiedersi che cosa è cambiato.
3. Registrare su ogni componente coinvolto il listener semantico adatto (tabella della sezione 3.3), che chiama quel metodo.
4. Chiamare il metodo anche alla fine del costruttore, perché la finestra parta coerente.

Esercizi

Esercizio 3.1: Quante volte (⋆)

Quante volte viene stampato "click" alla prima pressione del pulsante? E alla seconda?

```
JButton btn = new JButton("Prova");
ActionListener stampa = e -> System.out.println("click");
btn.addActionListener(stampa);
btn.addActionListener(e -> System.out.println("click"));
btn.addActionListener(stampa);
btn.removeActionListener(e -> System.out.println("click"));
btn.addActionListener(e -> btn.removeActionListener(stampa));
```

Esercizio 3.2: Chi è stato (⋆)

Tre pulsanti `btnA`, `btnB`, `btnC`, tutti con il testo "OK", condividono il listener `e -> System.out.println(e.getActionCommand())`. Che cosa si stampa premendo `btnB`? Come si modifica il listener, senza cambiare il testo dei pulsanti, perché stampi "B"? Indicare due modi.

Esercizio 3.3: Due slider legati (⋆)

Due slider `s1` e `s2`, entrambi da 0 a 100, sono legati: ciascuno ha un `ChangeListener` che imposta l'altro allo stesso valore con `setValue`. L'utente sposta `s1` su 40. Il programma entra in un ciclo infinito? Che cosa cambierebbe se `s2` andasse da 0 a 200 e il legame fosse "`s2` vale il doppio di `s1`", con l'inverso arrotondato per difetto?

Esercizio 3.4: Una tendina di record (⋆)

Che cosa mostra la tendina `new JComboBox<>(Colore.PALETTE)` se dal record `Colore` si toglie il `toString` ridefinito? E se lo si lascia, ma si assegna un renderer che estende `DefaultListCellRenderer` e non chiama `super`? Che cosa manca, in questo secondo caso, quando la voce è selezionata?

Esercizio 3.5: Un evento di troppo (⋆⋆)

Nel `SelettoreForma` si toglie il controllo `if (f == forma) return;` da `scegli`. Che cosa succede quando l'utente preme il pulsante della forma già scelta? È un problema nella Lavagna di questo capitolo? Descrivere un uso del componente in cui lo diventa.

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 3.6: Il contatore (⋆)

Una finestra con un'etichetta grande che mostra un numero, inizialmente 0, e tre pulsanti: "+1", "−1" e "Azzera". Un solo listener, registrato su tutti e tre, decide che cosa fare dal comando dell'evento. Il pulsante "−1" deve essere disabilitato (`setEnabled(false)`) quando il numero è 0, e il contatore non deve mai diventare negativo.

Soluzione: progetto `sol-d05-c03`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 3.5: Il pulsante disabilitato in un posto solo

Se l'abilitazione di "−1" si aggiorna solo nel ramo di "−1", "Azzera" porta il numero a 0 lasciando il pulsante abilitato, e "+1" non lo riabilita mai. L'abilitazione è parte dello stato mostrato: si ricalcola in un metodo solo, dopo ogni azione.

Esercizio 3.7: Celsius e Fahrenheit (⋆⋆)

Due `JSpinner` affiancati, uno in gradi Celsius e uno in Fahrenheit ($F = C \cdot 9/5 + 32$), con un decimale. Cambiando uno dei due, l'altro si aggiorna; la conversione deve funzionare in entrambi i versi senza che i due spinner si rimbalzino il valore all'infinito e senza che l'arrotondamento sposti il valore appena scritto dall'utente.

Soluzione: progetto `sol-d05-c03`, package `it.unibs.pajc.redolfi.e07`.

Esercizio 3.8: La lista della spesa (⋆⋆)

Un campo di testo con un pulsante "Aggiungi" (che scatta anche con Invio nel campo), una lista delle voci e un pulsante "Rimuovi". Una voce vuota o già presente (senza distinguere maiuscole e minuscole) non viene aggiunta; dopo l'aggiunta il campo si svuota. "Rimuovi" toglie la voce selezionata ed è abilitato solo quando una voce è selezionata. Un'etichetta in fondo dice quante voci ci sono.

Soluzione: progetto `sol-d05-c03`, package `it.unibs.pajc.redolfi.e08`.

Attenzione 3.6: La lista costruita con un array

`new JList<>(array)` non si può modificare: le voci si aggiungono a un `DefaultListModel`, creato prima e passato al costruttore della lista. La lista si aggiorna da sola; il conteggio no, perché è un'etichetta: va riscritto dopo ogni modifica del modello.

Esercizio 3.9: Il tastierino (⋆⋆⋆)

Scrivere un componente `PnlTastierino`, un `JPanel` con i tasti da 0 a 9 e un tasto "C", che genera un `ActionEvent` a ogni pressione, con il tasto premuto come comando; i listener si gestiscono con una `EventListenerList`. Poi una finestra con un display e *due* tastierini, uno a sinistra e uno a destra, che scrivono nello stesso display: le cifre si accodano, "C" cancella. Chi usa il tastierino non deve sapere che dentro ci sono dei `JButton`.
Suggerimento: dentro il tastierino ogni pulsante ha un listener interno che chiama `fireActionPerformed` con il proprio testo; la finestra registra un solo listener su ciascun tastierino.

Soluzione: progetto `sol-d05-c03`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 3.1. Alla prima pressione "click" esce tre volte: `stampa` è registrato due volte e la lambda della quarta riga una; la `removeActionListener` della sesta riga riceve una lambda nuova e non toglie niente. Durante la stessa pressione la quinta lambda toglie `stampa`, ma ne toglie un'occorrenza sola (l'ultima registrata), e la notifica in corso usa comunque la copia dell'elenco presa all'inizio. Alla seconda pressione restano una registrazione di `stampa` e la lambda: "click" due volte, e la quinta lambda toglie l'ultima `stampa`. Dalla terza in poi, una volta sola.

Risposta 3.2. Stampa "OK": il comando di un pulsante è il suo testo. Primo modo: confrontare la sorgente, `if (e.getSource() == btnB) ...`, con i pulsanti in campi. Secondo modo: dare a ogni pulsante un comando suo con `btnB.setActionCommand("B")`, che cambia il comando senza cambiare il testo; il listener resta com'è. Il secondo modo è il migliore quando il listener deve solo sapere *che cosa* fare, il primo quando deve lavorare sul componente.

Risposta 3.3. No: `setValue` genera un evento solo se il valore cambia. `s1` va a 40 e avvisa; il suo listener porta `s2` a 40, che avvisa; il listener di `s2` chiama `s1.setValue(40)`, ma `s1` vale già 40 e non succede niente. Con il fattore 2 e l'arrotondamento, invece, il valore può tornare diverso: l'utente porta `s2` a 81, `s1` diventa 40, e il suo listener riporta `s2` a 80, togliendo all'utente il valore che aveva scelto. Non è un ciclo infinito, ma è un rimbalzo sbagliato; si evita con una bandiera che dice "sto aggiornando io" e fa ignorare l'evento all'altro listener.

Risposta 3.4. Senza `toString` ridefinito la tendina mostra quello generato dal record: `Colore[nome=Nero, colore=java.awt.Color[r=0,g=0,b=0]]`, con tutte le componenti del colore. Con un renderer che non chiama `super` il testo, i colori e l'icona restano quelli con cui `DefaultListCellRenderer` è stato creato o che noi impostiamo: se impostiamo solo l'icona, tutte le voci hanno un testo vuoto, e manca soprattutto l'evidenziazione della voce selezionata, che è `super` a fare scegliendo i colori di selezione della lista.

Risposta 3.5. Il pulsante già premuto genera comunque il suo `ActionEvent`, `scegli` viene chiamato con la forma corrente e `fireActionPerformed` avvisa i listener di un cambiamento che non c'è stato. Nella Lavagna di questo capitolo non si vede: la riga di stato viene riscritta uguale. Lo diventa quando il listener fa qualcosa che non deve ripetersi, per esempio segna il disegno come modificato, registra l'operazione per annullarla, o aggiorna un altro componente che a sua volta notifica: un evento falso ne produce altri, e due componenti che si ascoltano a vicenda possono non fermarsi più.

Capitolo 4

Finestre di dialogo

La Lavagna del capitolo 3 reagisce a ogni scelta fatta nella barra, ma non sa ancora fare una cosa che ogni programma fa di continuo: fermarsi e chiedere. Quale file aprire, con quale nome salvare, quale colore usare fuori dalla palette, se buttare via le modifiche prima di chiudere. Sono domande che il programma pone in un momento preciso e di cui gli serve la risposta per andare avanti, e si pongono con le *finestre di dialogo*. Swing ne ha di pronte per i casi comuni (messaggi, conferme, scelta di un file o di un colore), e permette di scriverne di nuove quando quelle non bastano. Questo capitolo le presenta tutte, a partire dal meccanismo che le rende semplici da usare: il metodo che apre la finestra torna solo quando l'utente ha risposto. Alla fine la Lavagna apre e salva un file, accetta qualunque colore, ha una finestra di proprietà e non perde più modifiche alla chiusura.

4.1 Fermarsi a chiedere

Il caso più semplice è una domanda con un sì o un no. Prima di sostituire un file esistente il programma chiede conferma, e a seconda della risposta salva o lascia perdere:

```
int r = JOptionPane.showConfirmDialog(this, "Sostituire il file esistente?",
        "Salva come", JOptionPane.YES_NO_OPTION);
if (r == JOptionPane.YES_OPTION)
    scrivi(file);
```

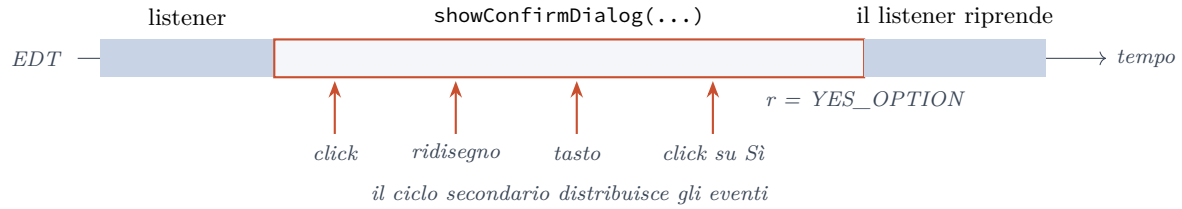
Il codice si legge dall'alto in basso come se l'utente rispondesse istantaneamente: la prima istruzione apre la finestra e restituisce la risposta, la seconda la usa. In realtà fra le due possono passare dieci secondi o dieci minuti, e `showConfirmDialog` torna solo quando l'utente ha premuto un pulsante o chiuso la finestra. È questa la proprietà che rende le finestre di dialogo così comode.

Definizione 4.1: Finestra di dialogo modale

Una *finestra di dialogo* (`JDialog`) è una finestra di primo livello che appartiene a un'altra, il suo *proprietario*: resta sopra di lui, gli si centra sopra, si chiude e si riduce a icona con lui. È *modale* se, finché è aperta, blocca l'input delle altre finestre dell'applicazione; per una finestra modale il metodo che la mostra (`setVisible(true)`, o i metodi `show...` delle finestre pronte) torna soltanto quando la finestra viene chiusa. Una finestra *non modale* lascia l'utente libero di lavorare anche nelle altre, e il metodo che la mostra torna subito.

Viene da chiedersi come possa il listener restare fermo ad aspettare, visto che il thread che lo esegue è lo stesso che deve gestire i click sulla finestra di dialogo. La risposta è che `setVisible(true)`, per una finestra modale, non si limita ad aspettare: avvia un ciclo degli eventi *secondario*, sullo stesso thread, che continua a estrarre eventi dalla coda e a distribuirli. I

click sul dialogo arrivano al dialogo, i ridisegni della finestra principale avvengono normalmente, l'input destinato alle finestre bloccate viene scartato. Quando il dialogo si chiude il ciclo secondario finisce, `setVisible` torna, e il listener riprende dalla riga successiva.



Una finestra modale vista dal thread degli eventi: il listener è sospeso dentro `show...`, ma gli eventi continuano a essere distribuiti.

La modalità di una `JDialog` si sceglie con un `ModalityType`. I due valori che servono sono `APPLICATION_MODAL`, che blocca tutte le finestre dell'applicazione e che è il comportamento delle finestre pronte, e `MODELESS`, la finestra non modale: una palette di strumenti, un pannello "Trova" che resta aperto mentre si lavora nel documento.

Attenzione 4.1: Il proprietario non è un dettaglio

Tutti i metodi che aprono una finestra di dialogo ricevono come primo argomento il proprietario, e passare `null` compila e funziona quasi: il dialogo compare al centro dello schermo invece che sulla finestra, può finire dietro di lei con un click sbagliato e, su alcuni sistemi, compare come una finestra a sé nella barra delle applicazioni. Il proprietario giusto è la finestra da cui parte la domanda: `this` in un metodo di una `JFrame`, `this` anche in un metodo di un `JDialog` che apre a sua volta un dialogo. Da un listener in una classe che non è una finestra, `SwingUtilities.getWindowAncestor(componente)` la trova.

4.2 Messaggi e domande: `JOptionPane`

`JOptionPane` raccoglie in una classe le finestre di dialogo più comuni: un messaggio da leggere, una domanda con alcune risposte possibili, la richiesta di un valore. Ciascuna si apre con un metodo statico `show...` che costruisce la finestra, la mostra in modo modale e restituisce la risposta.

4.2.1 Il messaggio

Il caso più semplice comunica qualcosa e aspetta che l'utente abbia letto. Il terzo e il quarto argomento sono il titolo e il *tipo* del messaggio, che decide l'icona:

```
JOptionPane.showMessageDialog(this, "Disegno salvato.");      // tipo INFORMATION
JOptionPane.showMessageDialog(this,
        "Impossibile scrivere casa.lav:\nil disco è pieno.",   // \n va a capo
        "Errore", JOptionPane.ERROR_MESSAGE);
```



I cinque tipi di messaggio, con le icone dell'aspetto Metal; il titolo di ogni finestra è il nome della costante.

Il tipo non è decorativo. `ERROR_MESSAGE` e `WARNING_MESSAGE` dicono all'utente, prima ancora che legga, che qualcosa è andato storto o può andare storto; su alcuni sistemi accompagnano la finestra con un suono. Un messaggio d'errore con l'icona dell'informazione, o una conferma banale con l'icona dell'errore, insegnano all'utente a non guardare le icone. Il messaggio può andare a capo con `\n`; e non è obbligatoriamente un testo, come si vedrà fra poco.

4.2.2 La domanda

`showConfirmDialog` pone una domanda e restituisce un intero che dice quale pulsante è stato premuto. Il quarto argomento sceglie i pulsanti (`YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`), e il risultato va confrontato con le costanti, mai con dei numeri:

```
int r = JOptionPane.showConfirmDialog(this, "Sostituire il file esistente?",
        "Conferma", JOptionPane.YES_NO_CANCEL_OPTION);
String esito = switch (r) {
    case JOptionPane.YES_OPTION -> "sì";
    case JOptionPane.NO_OPTION -> "no";
    case JOptionPane.CANCEL_OPTION -> "annullato";
    case JOptionPane.CLOSED_OPTION -> "chiuso con la X";
    default -> "imprevisto: " + r;
};
```

L'ultima costante è quella che si dimentica: l'utente può chiudere la finestra senza premere nessun pulsante, con la X o con Esc, e il risultato è `CLOSED_OPTION`. Un programma che tratta come "sì" tutto ciò che non è "no" sostituisce il file anche quando l'utente ha chiuso la finestra per non decidere. La regola sicura è l'opposta: si procede solo con la risposta esplicita, e tutto il resto vale come "annulla".

4.2.3 Il valore

`showInputDialog` chiede una stringa e la restituisce; se l'utente annulla, o chiude la finestra, restituisce `null`. Con un array di valori al posto del campo di testo mostra una tendina, e restituisce il valore scelto, come `Object`:

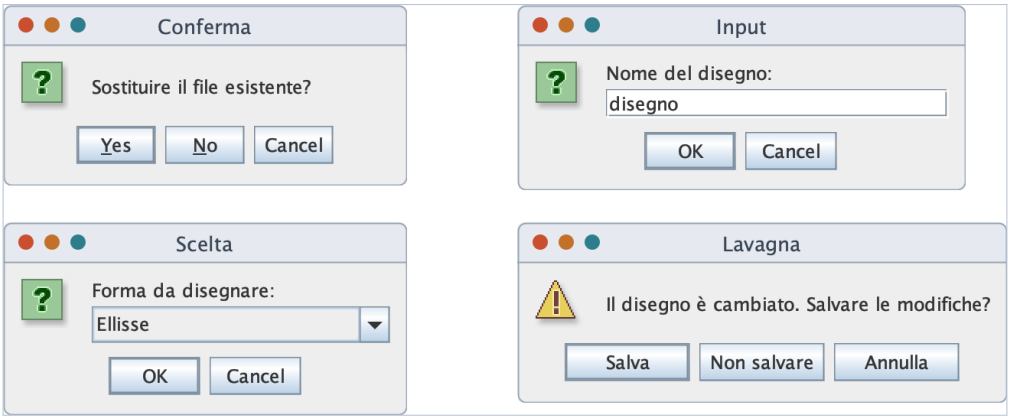
```
String nome = JOptionPane.showInputDialog(this, "Nome del disegno:", "disegno");
if (nome == null)
    return;                                    // annullato

String[] forme = { "Rettangolo", "Ellisse", "Mano libera" };
Object scelta = JOptionPane.showInputDialog(this, "Forma da disegnare:", "Scelta",
        JOptionPane.QUESTION_MESSAGE, null, forme, forme[1]);
```

4.2.4 Pulsanti nostri

Quando "Sì" e "No" non dicono abbastanza, `showOptionDialog` accetta un array di etichette e restituisce l'*indice* del pulsante premuto, oppure `CLOSED_OPTION`, cioè -1. È la finestra che la Lavagna mostra alla chiusura, dove "Salva", "Non salvare" e "Annulla" sono molto più chiari di tre pulsanti che rispondono a una domanda da rileggere:

```
String[] scelte = { "Salva", "Non salvare", "Annulla" };
int r = JOptionPane.showOptionDialog(this,
        "Il disegno è cambiato. Salvare le modifiche?", "Lavagna",
        JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE, null,
        scelte, scelte[0]);                    // scelte[0]: il pulsante di Invio
```



Conferma, input, scelta da un elenco e pulsanti nostri.

Le schermate mostrano un dettaglio pratico: i pulsanti predefiniti ("Yes", "No", "Cancel") hanno i testi della libreria, che nelle versioni recenti del JDK sono in inglese qualunque sia la lingua del sistema, mentre quelli passati da noi hanno i nostri. Per tradurre i predefiniti in tutta l'applicazione basta, all'inizio del `main`, `UIManager.put("OptionPane.yesButtonText", "Sì")` e lo stesso per `noButtonText` e `cancelButtonText`; oppure, per una domanda importante, si usa `showOptionDialog` con etichette che dicono che cosa succede.

Metodo	Che cosa chiede	Che cosa restituisce
`showMessageDialog`	niente: un avviso	niente (`void`)
`showConfirmDialog`	sì, no, annulla	`YES_OPTION`, …, `CLOSED_OPTION`
`showInputDialog`	una stringa, o una voce	la stringa o la voce; `null` se annulla
`showOptionDialog`	uno dei nostri pulsanti	l'indice; -1 se chiusa

Osservazione 4.1

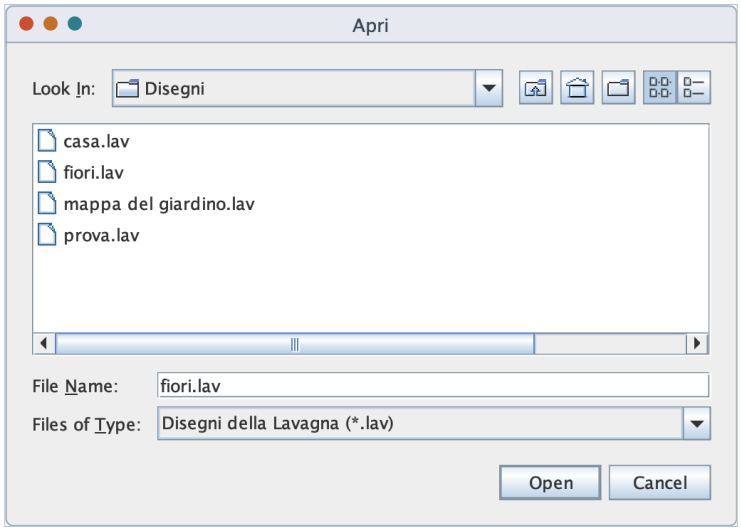
Il messaggio di tutti questi metodi è un `Object`, e se è un componente `JOptionPane` lo mostra così com'è. Un pannello con due campi e le loro etichette, passato a `showConfirmDialog` con `OK_CANCEL_OPTION`, diventa un piccolo modulo modale: se il risultato è `OK_OPTION` si leggono i campi. Per una domanda con due o tre valori è spesso tutto ciò che serve, senza scrivere una finestra di dialogo nostra.

4.3 Scegliere un file

Aprire e salvare un file richiede una finestra in cui l'utente navighi fra le cartelle e scelga un nome. Swing la offre con `JFileChooser`, un componente che si configura prima di mostrarlo e si interroga dopo:

```
JFileChooser fc = new JFileChooser(cartella);       // dove si parte
fc.setFileFilter(new FileNameExtensionFilter("Disegni (*.lav)", "lav"));
if (fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
    File f = fc.getSelectedFile();
    cartella = f.getParentFile();                   // si riparte da qui
    leggi(f);
}
```

`showOpenDialog` e `showSaveDialog` mostrano la finestra, con i testi adatti all'apertura o al salvataggio, e restituiscono `APPROVE_OPTION` se l'utente ha confermato, `CANCEL_OPTION` se ha annullato. Il file scelto lo dà `getSelectedFile`. Il filtro mostra solo i file con l'estensione indicata, ma l'utente può cambiarlo dalla tendina in basso, e un programma non deve fidarsene per decidere che il file sia valido.



Il `JFileChooser` in apertura, con il filtro sui disegni della Lavagna.

Alcune configurazioni ricorrono spesso. `setMultiSelectionEnabled(true)` permette di scegliere più file, che si leggono con `getSelectedFiles()`; `setFileSelectionMode` con la costante `DIRECTORIES_ONLY` fa scegliere una cartella; `setSelectedFile` propone un nome, utile nel salvataggio. Il chooser ricorda la cartella in cui l'utente si è fermato solo finché si riusa lo stesso oggetto; se se ne crea uno nuovo ogni volta, la cartella va tenuta in un campo, come nel codice qui sopra, e passata al costruttore.

Il salvataggio ha due insidie, entrambe a carico nostro. Il chooser non aggiunge l'estensione al nome scritto dall'utente, e non chiede conferma se il file esiste già:

```
if (fc.showSaveDialog(this) != JFileChooser.APPROVE_OPTION)
    return false;
File f = fc.getSelectedFile();
if (!f.getName().endsWith(".lav"))
    f = new File(f.getPath() + ".lav");     // l'estensione la mettiamo noi
if (f.exists() && !confermaSostituzione(f))
    return false;                           // e la conferma anche
return scrivi(f);
```

4.3.1 La finestra del sistema

Il `JFileChooser` è disegnato da Swing, e ha l'aspetto del resto dell'interfaccia Swing, che non è quello delle finestre del sistema operativo. AWT offre l'alternativa, `java.awt.FileDialog`, che mostra *la* finestra di scelta dei file del sistema, la stessa di tutti gli altri programmi:

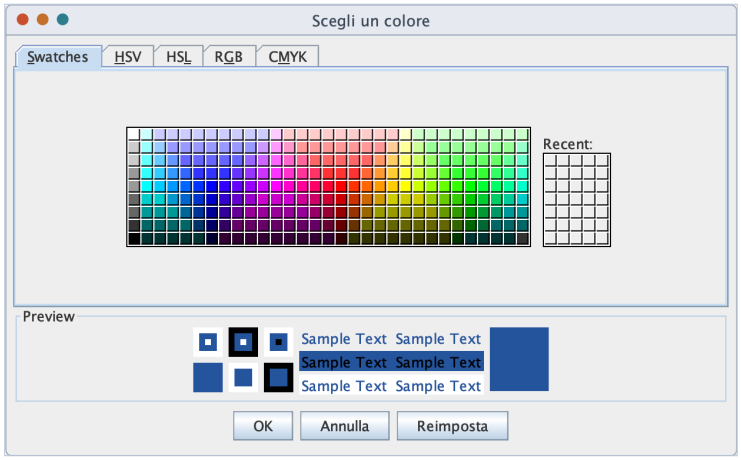
```
FileDialog fd = new FileDialog(this, "Apri un disegno", FileDialog.LOAD);
fd.setDirectory(cartella.getPath());
fd.setVisible(true);                        // modale: torna alla chiusura
if (fd.getFile() != null)                   // null se annullato
    leggi(new File(fd.getDirectory(), fd.getFile()));
```

Si guadagna la familiarità: l'utente ritrova le cartelle preferite, l'anteprima, la ricerca del suo sistema. Si perde il controllo: i filtri per estensione non funzionano su tutti i sistemi (su Windows `setFilenameFilter` viene ignorato), la scelta di una cartella invece di un file richiede trucchi diversi su ogni piattaforma, e il risultato arriva come due stringhe, cartella e nome, da ricomporre. Per un'applicazione che deve sembrare nativa è la scelta giusta; per tutto il resto il `JFileChooser` è più prevedibile.

4.4 Scegliere un colore

La palette della Lavagna ha cinque colori, e il pulsante "Altro…" deve offrirne qualunque altro. `JColorChooser.showDialog` apre una finestra con una tavolozza e le schede per comporre il colore in RGB, HSV e altri modelli, e restituisce il colore scelto, oppure `null` se l'utente annulla:

```
Color c = JColorChooser.showDialog(this, "Scegli un colore", attuale.colore());
if (c == null)
    return;                                 // annullato: niente cambia
```



Il `JColorChooser`, aperto sul colore corrente.

Il terzo argomento è il colore da cui parte la finestra: passare quello attuale permette all'utente di ritoccarlo invece di ricominciare. Il `JColorChooser` può anche aprirsi sopra un'altra finestra di dialogo, come succede nella finestra delle proprietà della prossima sezione, e allora il suo proprietario è quel dialogo, non la finestra principale: è il dialogo che aspetta la risposta, ed è sopra di lui che il colore deve comparire.

4.5 Una finestra di dialogo nostra

Le finestre pronte coprono le domande con una risposta sola. La Lavagna ha bisogno di qualcosa di più: una finestra "Proprietà" in cui decidere insieme se mostrare la griglia, con che passo, e con quale colore di sfondo. Si scrive estendendo `JDialog`, e si costruisce come una `JFrame`; la cosa che va pensata con cura è come restituisce la risposta.



La finestra delle proprietà della Lavagna: una finestra di dialogo nostra.

La risposta sono tre valori che vanno insieme, e diventano un record. La finestra lo riceve all'apertura, per partire dai valori attuali, e ne restituisce uno nuovo alla chiusura:

```
public class DialogProprieta extends JDialog {

    public record Proprieta(boolean griglia, int passo, Color sfondo) { }

    private final JCheckBox chkGriglia = new JCheckBox("Mostra la griglia");
    private final JSpinner spnPasso =
            new JSpinner(new SpinnerNumberModel(20, 5, 100, 5));
    private final JButton btnSfondo = new JButton();
    private Color sfondo;
    private Proprieta risultato;            // null se l'utente annulla
```

Il costruttore chiama quello di `JDialog` con il proprietario, il titolo e la modalità, imposta i componenti dai valori iniziali e li dispone (il modulo con un `GridLayout(0, 2)`, i pulsanti in fondo a destra, come nel capitolo 2). Poi collega i pulsanti:

```
public DialogProprieta(Window proprietario, Proprieta iniziali) {
    super(proprietario, "Proprietà", ModalityType.APPLICATION_MODAL);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);   // la X annulla
    // ... valori iniziali e composizione ...

    btnOk.addActionListener(e -> {
        risultato = new Proprieta(chkGriglia.isSelected(),
                (Integer) spnPasso.getValue(), sfondo);
        dispose();                                // chiedi() riprende da qui
    });
    btnAnnulla.addActionListener(e -> dispose());

    getRootPane().setDefaultButton(btnOk);        // Invio = OK
    getRootPane().registerKeyboardAction(e -> dispose(),     // Esc = Annulla
            KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0),
            JComponent.WHEN_IN_FOCUSED_WINDOW);
    pack();
    setLocationRelativeTo(proprietario);
}
```

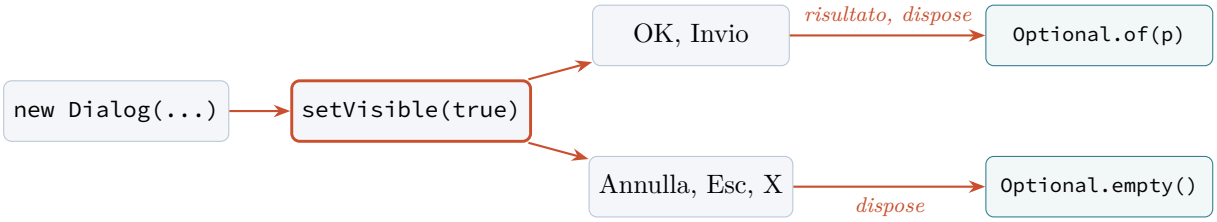
Ci sono quattro modi di chiudere la finestra, e solo uno conferma. "OK" fotografa i valori dei componenti in un record nuovo e chiude; "Annulla", la X e il tasto Esc chiudono e basta, e il risultato resta `null`. Invio equivale a "OK" ovunque sia il fuoco, grazie a `setDefaultButton`; per Esc non esiste un metodo equivalente, e si registra un'azione sul tasto (il meccanismo delle scorciatoie, ripreso nel capitolo 8).

Resta il metodo con cui chi apre la finestra riceve la risposta. Mostra la finestra e, siccome è modale, prosegue solo quando è stata chiusa; a quel punto il risultato è pronto:

```
public Optional<Proprieta> chiedi() {
    setVisible(true);                             // modale: torna alla chiusura
    return Optional.ofNullable(risultato);
}
```

Chi usa la finestra scrive una riga: `new DialogProprieta(this, attuali).chiedi()`, e riceve un `Optional` pieno se l'utente ha confermato, vuoto se ha annullato. `Optional` dice nel tipo ciò che `null` direbbe solo nella documentazione: la risposta può mancare, e chi la riceve

non può dimenticarlo.



La vita di una finestra di dialogo nostra: si crea, si mostra, si chiude in uno di quattro modi, restituisce un valore o niente.

Osservazione 4.2

La finestra non tocca niente della Lavagna: non cambia lo sfondo dell'area, non segna il disegno come modificato, non sa nemmeno che esiste una Lavagna. Riceve dei valori e ne restituisce altri. È l'immagine guida del capitolo 1 applicata alle finestre: la Lavagna aggiorna il proprio stato con la risposta, e da lì aggiorna ciò che mostra. Una finestra di dialogo scritta così si prova da sola, si riusa in un altro programma, e non può lasciare la finestra principale a metà di una modifica se l'utente annulla.

Attenzione 4.2: Nascondere non è chiudere

`setVisible(false)` nasconde la finestra senza distruggerla: la finestra tiene le sue risorse, e se la stessa istanza viene mostrata di nuovo si ripresenta con i valori dell'ultima volta, `risultato` compreso. Un "Annulla" alla seconda apertura restituirebbe allora la risposta della prima. Una finestra di dialogo come questa si crea nuova a ogni domanda e si chiude con `dispose()`.

Ricetta: Una finestra di dialogo modale nostra

1. Un record per la risposta, con tutti i valori che la finestra raccoglie.
2. Una classe che estende `JDialog`: costruttore con il proprietario e i valori iniziali, `APPLICATION_MODAL`, `DISPOSE_ON_CLOSE`.
3. "OK" costruisce il record nel campo `risultato` e chiama `dispose()`; "Annulla" chiama solo `dispose()`.
4. Invio con `setDefaultButton`, Esc con un'azione sul tasto; `pack()` e `setLocationRelativeTo(proprietario)` alla fine del costruttore.
5. Un metodo `chiedi()` che chiama `setVisible(true)` e restituisce `Optional.ofNullable(risultato)`.

4.6 L'esempio guida: la Lavagna che chiede

La Lavagna di questo capitolo usa tutte le finestre viste. Il file che apre e salva contiene per ora soltanto le proprietà (le figure arriveranno con il capitolo 5), ma il percorso è già quello completo: un campo `file` con il file corrente, `null` finché non si salva; un campo `modificato` che dice se ci sono modifiche non salvate; il titolo della finestra che mostra entrambi, come "Lavagna - casa.lav *".

Il colore personalizzato chiude un discorso del capitolo 3: la tendina dei colori ha ora un modello esplicito, e il colore scelto nel `JColorChooser` vi entra come voce nuova.

```
private void scegliColore() {
    Colore attuale = (Colore) cmbColore.getSelectedItem();
    Color c = JColorChooser.showDialog(this, "Altro colore", attuale.colore());
    if (c == null)
        return;                                  // annullato: niente cambia
    Colore nuovo = new Colore(String.format("#%02X%02X%02X",
            c.getRed(), c.getGreen(), c.getBlue()), c);
    mdlColori.addElement(nuovo);             // il modello avvisa la tendina...
    mdlColori.setSelectedItem(nuovo);        // ...che si aggiorna da sola
}
```

La finestra delle proprietà si apre con i valori correnti, e se l'utente conferma la Lavagna aggiorna il proprio stato e ciò che ne dipende:

```
private void modificaProprieta() {
    new DialogProprieta(this, proprieta).chiedi().ifPresent(p -> {
        proprieta = p;
        pnlArea.setBackground(p.sfondo());
        segnaModificato();                       // modificato = true e titolo
    });
}
```

La parte più delicata è la chiusura. La X della finestra non deve chiudere: deve chiedere, se ci sono modifiche, e chiudere solo se l'utente lo conferma. Per prima cosa si toglie alla finestra la sua reazione automatica e si ascolta l'evento della X:

```
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);    // la X la gestiamo noi
addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        esci();
    }
});
```

`WindowListener` ha sette metodi, e l'adapter permette di ridefinire solo `windowClosing`, che scatta quando l'utente chiede di chiudere. `esci()` è chiamato anche dalla voce "Esci" del menu, così i due percorsi sono lo stesso. La domanda vera sta in un metodo che serve anche ad "Apri", perché aprire un altro disegno butta via quello corrente esattamente come chiudere:

```
private boolean puoAbbandonare() {
    if (!modificato)
        return true;
    String[] scelte = { "Salva", "Non salvare", "Annulla" };
    int r = JOptionPane.showOptionDialog(this,
            "Il disegno è cambiato. Salvare le modifiche?", "Lavagna",
            JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE, null,
            scelte, scelte[0]);
    return switch (r) {
        case 0 -> salva();            // avanti solo se ha salvato davvero
        case 1 -> true;
        default -> false;             // Annulla, o la X del dialogo
    };
}

private void esci() {
    if (puoAbbandonare())
        dispose();                    // l'ultima finestra: il programma finisce
}
```

Il caso "Salva" restituisce il risultato di `salva()`, non `true`: se il disegno non ha ancora un nome, `salva` apre il `JFileChooser`, e se l'utente lì annulla il salvataggio non è avvenuto, e chiudere perderebbe le modifiche proprio dopo che l'utente ha detto di volerle salvare. Lo stesso vale per un errore di scrittura, che `scrivi` segnala con un messaggio d'errore e un `false`. `dispose()` sulla sola finestra dell'applicazione fa terminare il programma, come `EXIT_ON_CLOSE`, ma solo dopo che la domanda ha avuto la risposta giusta.

Attenzione 4.3: `EXIT_ON_CLOSE` e la domanda

Con `EXIT_ON_CLOSE` la finestra chiama `System.exit` appena l'utente preme la X, *dopo* aver avvisato i `WindowListener`: il listener mostra la domanda, e se la domanda è modale il programma termina comunque appena il listener torna, qualunque sia la risposta. Per chiedere prima di chiudere serve `DO_NOTHING_ON_CLOSE`, che lascia la decisione interamente al programma.

Esercizi

Esercizio 4.1: Il sì che non è un sì (⋆)

Che cosa c'è di sbagliato in queste righe? Indicare per quali azioni dell'utente il file viene cancellato, e correggerle.

```
int r = JOptionPane.showConfirmDialog(this, "Cancellare il file?",
        "Conferma", JOptionPane.YES_NO_OPTION);
if (r != JOptionPane.NO_OPTION)
    file.delete();
```

Esercizio 4.2: Un numero, per favore (⋆)

Il frammento chiede un numero all'utente. Elencare tutto ciò che può andare storto, dicendo che cosa succede in ciascun caso, e riscriverlo in modo che richieda il numero finché non è valido o l'utente non annulla.

```
int n = Integer.parseInt(JOptionPane.showInputDialog(this, "Quanti lati?"));
```

Esercizio 4.3: Prima e dopo (⋆)

Un listener esegue queste tre righe.

```
lblStato.setText("Prima");
JOptionPane.showMessageDialog(this, "Ciao");
lblStato.setText("Dopo");
```

Che cosa mostra l'etichetta mentre il messaggio è aperto? La finestra principale si ridisegna se la si copre e la si scopre mentre il messaggio è aperto? Perché?

Esercizio 4.4: La finestra riusata (⋆)

La Lavagna crea la sua `DialogProprieta` una volta sola, in un campo, e la chiude con `setVisible(false)` invece che con `dispose()`. L'utente apre la finestra, conferma con OK, la riapre e preme Annulla. Che cosa restituisce `chiedi()` la seconda volta? E che cosa mostrano i componenti alla seconda apertura?

Esercizio 4.5: Chiusa due volte (⋆⋆)

Nella Lavagna di questo capitolo si sostituisce `DO_NOTHING_ON_CLOSE` con `EXIT_ON_CLOSE`, lasciando il `WindowListener`. L'utente, con modifiche non salvate, preme la X e poi "Annulla" nella domanda. Che cosa succede? E se sceglie "Salva" e poi annulla nel `JFileChooser`?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 4.6: Indovina il numero (⋆)

Un programma senza finestra principale, fatto solo di finestre di dialogo: estrae un numero da 1 a 100 e chiede all'utente di indovinarlo, rispondendo "troppo alto" o "troppo basso" con un messaggio di tipo informazione; un input non numerico produce un messaggio d'errore e non conta come tentativo. Indovinato il numero, dice in quanti tentativi e chiede con una conferma se giocare ancora. Annullare in qualunque momento termina il gioco.

Soluzione: progetto `sol-d05-c04`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 4.4: Senza proprietario, ma sul thread giusto

Senza una finestra principale il proprietario dei dialoghi è `null`, ed è uno dei pochi casi in cui va bene. Il gioco però va avviato con `invokeLater` come qualunque programma Swing; e alla fine il programma termina da solo, perché non resta nessuna finestra aperta: i dialoghi di `JOptionPane` si chiudono da sé.

Esercizio 4.7: Il login (⋆⋆)

Scrivere una finestra di dialogo modale `DialogLogin` con un campo per il nome, un campo per la password (`JPasswordField`) e i pulsanti "Annulla" e "Accedi"; "Accedi" è abilitato solo quando entrambi i campi contengono qualcosa (un `DocumentListener` sui due documenti). Il metodo `chiedi()` restituisce un `Optional<Credenziali>`, con `Credenziali` un record. Un `main` di prova la apre, e dopo la risposta stampa il nome, oppure "annullato".

Soluzione: progetto `sol-d05-c04`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 4.5: La password come stringa

`JPasswordField.getPassword()` restituisce un `char[]` e non una `String`, per poterla cancellare dalla memoria dopo l'uso; `getText()` esiste ma è deprecato. Nel record la password resta un `char[]`, con la consapevolezza che un record confronta gli array per riferimento, non per contenuto.

Esercizio 4.8: Il mini editor (⋆⋆)

Un editor di testo con una `JTextArea` e il menu File: Nuovo, Apri…, Salva, Salva come…, Esci. Il titolo mostra il nome del file e un asterisco se ci sono modifiche (un `DocumentListener` le rileva). Nuovo, Apri ed Esci, con modifiche non salvate, chiedono che cosa fare con "Salva", "Non salvare" e "Annulla"; la X della finestra si comporta come Esci; "Salva come" chiede conferma prima di sostituire un file esistente. Gli errori di lettura e scrittura si mostrano con un messaggio d'errore.

Soluzione: progetto `sol-d05-c04`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 4.9: Trova (⋆⋆⋆)

Aggiungere al mini editor una finestra di dialogo *non modale* "Trova", aperta da un pulsante o da una voce di menu: un campo di testo, una casella "Maiuscole e minuscole" e i pulsanti "Trova successivo" e "Chiudi". Mentre è aperta l'utente può continuare a scrivere nell'editor; "Trova successivo" seleziona nella `JTextArea` l'occorrenza successiva a partire dal cursore, ricominciando dall'inizio quando arriva in fondo, e segnala con un messaggio quando non ce ne sono. Riaprendo "Trova" compare la stessa finestra, con l'ultimo testo cercato.

Suggerimento: una finestra non modale non restituisce un valore, perché il metodo che la mostra torna subito; agisce sull'editor attraverso un riferimento che riceve nel costruttore. Qui riusarla, con `setVisible(false)`, è proprio ciò che si vuole.

Soluzione: progetto `sol-d05-c04`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 4.1. La condizione cancella il file per ogni risposta che non sia "No": con "Sì", ma anche quando l'utente chiude la finestra con la X o con Esc, che danno `CLOSED_OPTION`. Chi chiude una domanda senza rispondere non ha acconsentito. La correzione procede solo con la risposta esplicita: `if (r == JOptionPane.YES_OPTION) file.delete();`. In generale, le azioni distruttive si eseguono per una risposta precisa, e tutto il resto vale come "annulla".

Risposta 4.2. Tre cose. Se l'utente annulla, `showInputDialog` restituisce `null`, e `parseInt(null)` lancia `NumberFormatException`. Se scrive qualcosa che non è un numero ("sei", "6 lati", una stringa vuota) l'eccezione è la stessa. Se scrive un numero che non ha senso (0, −3), il frammento lo accetta. In tutti i casi l'eccezione, non catturata, finisce nel thread degli eventi: il listener si interrompe, la console mostra lo stack, e l'utente non vede niente. La versione corretta è un ciclo: si chiede; se la risposta è `null` si esce; altrimenti si prova `Integer.parseInt(s.strip())` in un `try`, si controlla l'intervallo, e in caso di errore si mostra un messaggio di tipo errore e si richiede.

Risposta 4.3. L'etichetta mostra "Prima": il listener è sospeso dentro `showMessageDialog` e non ha ancora eseguito la terza istruzione. "Dopo" compare solo quando il messaggio viene chiuso. La finestra principale si ridisegna normalmente, perché mentre il dialogo modale è aperto il thread degli eventi esegue un ciclo secondario che continua a distribuire gli eventi, ridisegni compresi; è bloccato solo l'input verso la finestra principale. È la differenza fra una finestra modale e un listener che fa un lavoro lungo: in quel caso la finestra non si ridisegnerebbe.

Risposta 4.4. La seconda volta `chiedi()` restituisce di nuovo le proprietà confermate la prima volta, anche se l'utente ha annullato: `risultato` è un campo dell'istanza, è stato impostato dal primo OK e nessuno l'ha rimesso a `null`. I componenti, alla seconda apertura, mostrano i valori lasciati alla chiusura precedente, non quelli attuali della Lavagna, che potrebbero essere cambiati nel frattempo (per esempio aprendo un altro file). Si risolve creando una finestra nuova a ogni apertura; oppure, se la si vuole riusare, azzerando `risultato` e reimpostando i componenti in `chiedi()`.

Risposta 4.5. Con `EXIT_ON_CLOSE` la finestra avvisa i listener e poi chiama `System.exit`. Il listener chiama `esci()`, che mostra la domanda e aspetta; l'utente preme "Annulla", `puoAbbandonare` restituisce `false`, `esci()` non fa niente e torna; a quel punto la finestra esegue comunque `System.exit`, e il programma termina con le modifiche perse. Con "Salva" e poi "Annulla" nel chooser succede lo stesso: nessun salvataggio, e uscita. La domanda c'è, ma la risposta non conta.

Capitolo 5

Il componente che si disegna

Nella Lavagna del capitolo 1 il listener del mouse prendeva il contesto grafico del pannello e ci disegnava sopra; al primo ridisegno il rettangolo spariva. Da allora la finestra ha imparato a comporsi, a reagire e a chiedere, ma l'area bianca è rimasta vuota. Questo capitolo risponde alla seconda domanda della dispensa, dove si disegna perché il disegno resti. La risposta è un componente che sa disegnarsi da solo, ridefinendo un metodo che Swing chiama ogni volta che serve. Si vedrà chi chiama quel metodo e quando, come si ragiona nel sistema di coordinate dei pixel, e con quali istruzioni si disegnano linee, rettangoli, ovali, archi, poligoni e testo. Alla fine la Lavagna disegna davvero, e il disegno non sparisce più.

5.1 Dove si disegna

Ogni componente Swing si disegna da sé: un pulsante sa disegnare il suo bordo e il suo testo, una lista le sue righe. Lo fa in un metodo, `paintComponent(Graphics g)`, che Swing chiama quando il componente deve comparire sullo schermo, passandogli un *contesto grafico*: l'oggetto `Graphics`, con i metodi per disegnare nella superficie del componente. Per avere un componente che disegna ciò che vogliamo noi basta estendere un componente e ridefinire quel metodo.

Definizione 5.1: Contesto grafico

Un *contesto grafico* è un oggetto di tipo `java.awt.Graphics` che rappresenta una superficie su cui disegnare (una zona dello schermo, un'immagine, una pagina da stampare) insieme allo *stato* del disegno: il colore corrente, il font corrente, la regione entro cui il disegno è visibile. I suoi metodi (`drawLine`, `fillRect`, `drawString`, ...) disegnano sulla superficie usando quello stato.

Il primo pannello disegna una cornice e le due diagonali, e lo fa a partire dalle dimensioni che il pannello ha nel momento in cui viene disegnato:

```
public class PnlCroce extends JPanel {

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);                    // lo sfondo
        int w = getWidth(), h = getHeight();
        g.setColor(Color.BLUE);
        g.drawRect(10, 10, w - 21, h - 21);         // la cornice
        g.drawLine(10, 10, w - 11, h - 11);         // le due diagonali
        g.drawLine(10, h - 11, w - 11, 10);
    }
}
```

Il pannello si usa come qualunque componente: lo si mette al centro di una finestra, e quando la finestra viene ridimensionata la croce si ridisegna adattandosi, perché a ogni chiamata `getWidth()` e `getHeight()` danno le misure di quel momento. Tre dettagli di questo codice valgono per tutti i pannelli che si disegnano, e conviene fissarli subito.

`protected` e `@Override`. Il metodo ridefinito ha la stessa visibilità dell'originale, `protected`: non lo deve chiamare nessun altro, se non Swing. L'annotazione fa controllare al compilatore che il nome e i parametri siano proprio quelli; senza, un `paintcomponent` scritto con la minuscola sarebbe un metodo nuovo che nessuno chiama mai, e il pannello resterebbe vuoto senza un errore.

`super.paintComponent(g)`. La versione di `JPanel` riempie il componente con il colore di sfondo, cancellando ciò che c'era prima. Senza questa chiamata il pannello disegna sopra il disegno precedente: muovendo qualcosa si vedono le sue scie, e nelle zone non disegnate possono comparire residui di altri componenti.

`paintComponent` e non `paint`. `JComponent` ha anche un metodo `paint`, che è quello che Swing chiama davvero: disegna il componente con `paintComponent`, poi il bordo con `paintBorder`, poi i componenti figli con `paintChildren`. Ridefinire `paint` toglie il bordo e i figli, a meno di ricordarsi di chiamare `super.paint`; ridefinire `paintComponent` lascia tutto il resto al suo posto.

5.1.1 `JPanel` o `JComponent`

Si può estendere `JPanel` o direttamente `JComponent`, e la differenza sta proprio in ciò che fa la chiamata a `super`.

	`JPanel`	`JComponent`
`super.paintComponent`	riempie con lo sfondo	non fa niente
opaco (`isOpaque()`)	sì	no
`setBackground`	si vede	non si vede, se non lo si disegna noi
layout predefinito	`FlowLayout`	nessuno

Un `JPanel` è la scelta comoda: lo sfondo lo disegna lui, e il colore si cambia con `setBackground`. Un `JComponent` è più leggero, ma è trasparente: se si vuole uno sfondo, lo si disegna con un `fillRect` su tutta la superficie, e conviene dichiararlo opaco con `setOpaque(true)`, così che Swing non ridisegni inutilmente ciò che sta dietro. In questa dispensa i pannelli che si disegnano estendono `JPanel`.

Attenzione 5.1: Mai `getGraphics()` per disegnare

`getGraphics()` restituisce un contesto grafico del componente in qualunque momento, ed è per questo che il designer, e molti esempi in rete, lo usano nei listener. Ma ciò che si disegna così resta sullo schermo solo finché Swing non ridisegna il componente, e Swing lo ridisegna di continuo, chiamando `paintComponent`: è il disegno che spariva nel capitolo 1. Tutto ciò che deve restare visibile si disegna in `paintComponent`, a partire dallo stato del componente.

5.2 Chi chiama `paintComponent`, e quando

Il programma non chiama mai `paintComponent`. Lo chiama Swing, sul thread degli eventi, in tre situazioni: quando il componente compare la prima volta; quando cambia dimensione; quando una parte di esso torna visibile dopo essere stata coperta, da un'altra finestra o da un menu a tendina. E in una quarta, che dipende dal programma: quando il programma lo chiede con `repaint()`.

Per vederlo basta un pannello che conta le proprie chiamate e lo scrive in console:

```
private int disegni;

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    disegni++;
    String thread = Thread.currentThread().getName();
    System.out.println("disegno " + disegni + " su " + thread);
    // ...
}
```

```
disegno 1 su AWT-EventQueue-0
disegno 2 su AWT-EventQueue-0
disegno 3 su AWT-EventQueue-0
...
disegno 57 su AWT-EventQueue-0
```

La finestra compare: una chiamata. Si ridimensiona trascinando l'angolo per un secondo: qualche decina di chiamate. Si copre e si scopre: altre ancora. Il thread è sempre lo stesso, quello degli eventi, che dipinge fra un evento e l'altro. Il numero di chiamate non si può prevedere, e non deve contare: `paintComponent` deve produrre lo stesso disegno la prima volta e la centesima, perché disegna lo stato, e lo stato fra un ridisegno e l'altro non cambia da solo.

Definizione 5.2: `repaint`

`repaint()` non disegna: chiede a Swing di ridisegnare il componente il prima possibile, mettendo una richiesta nella coda degli eventi, e torna subito. Più richieste per lo stesso componente che arrivano prima del ridisegno vengono fuse in una sola. `repaint(x, y, w, h)` chiede di ridisegnare soltanto un rettangolo del componente.

Che `repaint` sia una richiesta e non un'esecuzione ha due conseguenze. La prima è che si può chiamare senza preoccupazioni: dieci `repaint` di seguito in un listener producono un solo ridisegno, alla fine, quando il listener è tornato e la coda arriva alla richiesta. La seconda è che il disegno non avviene dove sta la chiamata: chi chiama `repaint()` e subito dopo si aspetta di trovare il disegno aggiornato, per esempio per salvarlo in un'immagine, trova quello vecchio.

Da qui le regole di `paintComponent`, che discendono tutte dall'immagine guida: `paintComponent` *legge* lo stato e lo disegna, e basta.

- **Non cambia lo stato.** Un contatore incrementato, una posizione spostata, un elemento aggiunto dentro `paintComponent` cambiano a ogni ridisegno, e i ridisegni non si contano: il disegno cambierebbe coprendo e scoprendo la finestra.

- **Non chiama `repaint`.** Chiedere un ridisegno mentre si ridisegna produce un ciclo senza fine, che tiene la CPU occupata anche quando non succede niente.
- **Non fa lavoro lungo.** `paintComponent` gira sul thread degli eventi, come i listener: finché non torna la finestra non risponde. Calcoli costosi si fanno prima, quando lo stato cambia, e se ne salva il risultato.

5.3 Le coordinate dei pixel

Il contesto grafico che `paintComponent` riceve misura tutto in pixel, a partire dall'angolo in alto a sinistra del componente. L'asse x va verso destra, come in matematica; l'asse y va verso il *basso*, al contrario. Il punto $(0, 0)$ è l'angolo in alto a sinistra, $(w - 1, h - 1)$ il pixel in basso a destra, con w e h dati da `getWidth()` e `getHeight()`.

Le coordinate sono intere, e indicano le linee *fra* i pixel, non i pixel. Un `fillRect(1, 1, 4, 3)` riempie l'area fra le linee $x = 1$ e $x = 5$, $y = 1$ e $y = 4$: esattamente 4 per 3 pixel. Un `drawRect(1, 1, 4, 3)` traccia invece il contorno con una penna larga un pixel che si appoggia sotto e a destra della linea geometrica, e occupa 5 per 4 pixel: uno in più in ciascuna direzione.



Ogni quadretto è un pixel; la linea rossa è il rettangolo geometrico. Il riempimento sta dentro, il contorno sborda di un pixel a destra e in basso.

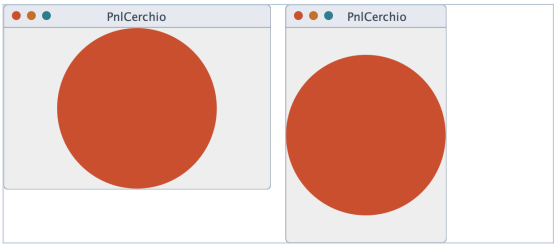
Il pixel in più spiega un'abitudine che sembra pignoleria: il bordo di un pannello di larghezza w si disegna con `drawRect(0, 0, w - 1, h - 1)`. Con `drawRect(0, 0, w, h)` il lato destro e quello inferiore cadrebbero sulle colonne w e sulla riga h, che non appartengono al pannello, e non si vedrebbero.

5.3.1 Le misure si ricalcolano

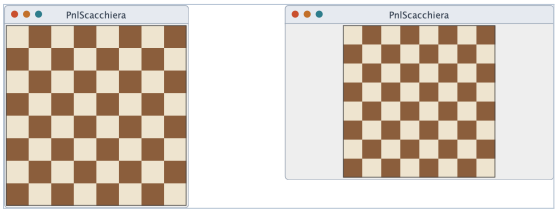
Un pannello non sa quanto sarà grande: lo decide il layout della finestra, e cambia a ogni ridimensionamento. Le misure di ciò che si disegna non si scrivono dunque nel codice né si salvano in campi nel costruttore, dove il pannello non ha ancora una dimensione: si ricalcolano in `paintComponent` a partire da `getWidth()` e `getHeight()`. L'esempio tipico è il cerchio centrato, il più grande che il pannello può contenere:

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);                    // lo sfondo
    int d = Math.min(getWidth(), getHeight());  // il diametro: il lato minore
    int x = (getWidth() - d) / 2;               // l'angolo del quadrato
    int y = (getHeight() - d) / 2;
    g.setColor(new Color(0xC9, 0x4F, 0x2E));
    g.fillOval(x, y, d, d);
}
```

Un cerchio, per `Graphics`, è un'ellisse inscritta in un rettangolo: lo si descrive con l'angolo in alto a sinistra e le due dimensioni del rettangolo, non con il centro e il raggio. Per centrarlo si parte quindi dal centro del pannello, $(w/2, h/2)$, e si toglie mezzo diametro; le due righe del codice fanno lo stesso conto in un modo che evita un arrotondamento in più.



Lo stesso pannello in due finestre di forma diversa: il cerchio segue il lato minore.



La scacchiera resta quadrata e centrata qualunque sia la forma del pannello.

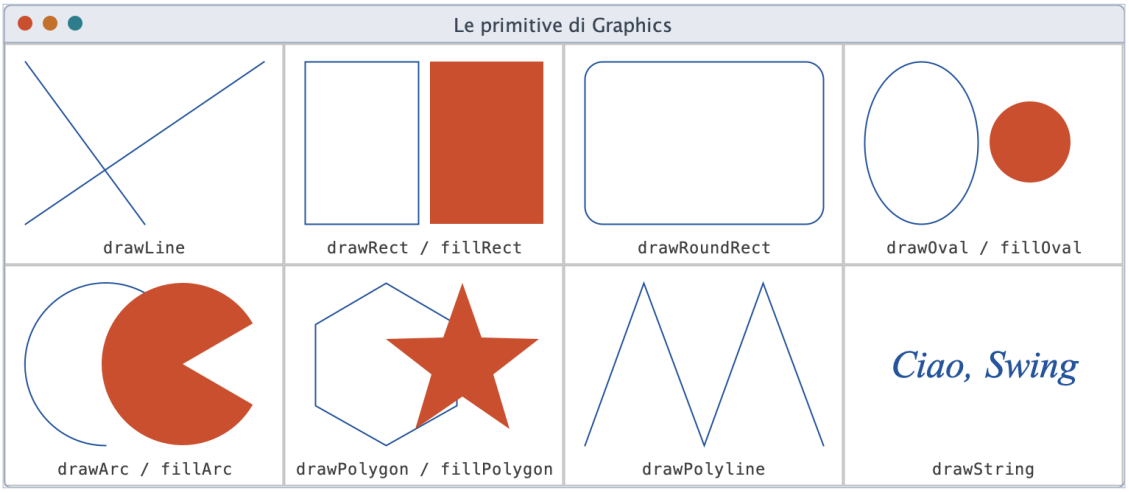
La scacchiera della seconda schermata applica la stessa idea a una griglia: la casella è la più grande che permette a otto caselle di stare nel lato minore, e la scacchiera si centra con il margine che avanza.

```
int casella = Math.min(getWidth(), getHeight()) / N;   // la piu' grande
int x0 = (getWidth() - casella * N) / 2;               // al centro
int y0 = (getHeight() - casella * N) / 2;
for (int riga = 0; riga < N; riga++)
    for (int col = 0; col < N; col++) {
        g.setColor((riga + col) % 2 == 0 ? CHIARA : SCURA);
        g.fillRect(x0 + col * casella, y0 + riga * casella, casella, casella);
    }
```

La divisione intera che calcola la casella non è un dettaglio: con caselle intere le righe della griglia cadono su pixel interi, e le caselle sono tutte uguali. Con una dimensione frazionaria arrotondata a ogni casella, alcune sarebbero larghe un pixel in più delle altre.

5.4 Le primitive di `Graphics`

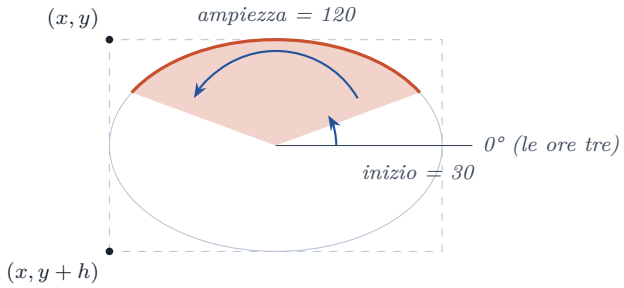
`Graphics` offre un piccolo insieme di metodi per le forme di base, le *primitive*. Quasi tutte esistono in due versioni: `draw...` traccia il contorno, `fill...` riempie l'interno, entrambe con il colore corrente.



Le primitive di `Graphics`, una per riquadro.

Metodo	**Che cosa disegna**
`drawLine(x1, y1, x2, y2)`	il segmento fra due punti
`drawRect`, `fillRect(x, y, w, h)`	il rettangolo con l'angolo in (x, y)
`drawRoundRect`, `fillRoundRect(x, y, w, h, aw, ah)`	con gli angoli arrotondati
`drawOval`, `fillOval(x, y, w, h)`	l'ellisse inscritta nel rettangolo
`drawArc`, `fillArc(x, y, w, h, inizio, ampiezza)`	un arco o un settore dell'ellisse
`drawPolygon`, `fillPolygon(xs, ys, n)`	il poligono chiuso con quei vertici
`drawPolyline(xs, ys, n)`	la spezzata aperta con quei vertici
`drawString(s, x, y)`	il testo, sulla linea di base in y
`clearRect(x, y, w, h)`	il rettangolo riempito con lo sfondo

Rettangoli e ovali si descrivono sempre con il rettangolo che li contiene. Gli archi aggiungono due angoli, in gradi interi: l'angolo di partenza, misurato dalla direzione delle ore tre, e l'ampiezza, positiva in senso *antiorario*. È il verso della matematica, nonostante l'asse y capovolto: la libreria fa la conversione per noi. `fillArc` riempie il settore, la fetta di torta fra l'arco e il centro.



`fillArc(x, y, w, h, 30, 120)`: il settore dell'ellisse inscritta nel rettangolo tratteggiato, da 30° a 150° in senso antiorario.

5.4.1 Poligoni

Un poligono si descrive con i suoi vertici, nell'ordine in cui si percorre il contorno: due array, uno con le x e uno con le y, e il numero di vertici. `drawPolygon` chiude da solo il contorno, tornando dall'ultimo vertice al primo; `drawPolyline` no, e disegna una spezzata aperta. Un triangolo con il vertice in alto:

```
int[] xs = { 100, 150, 50 };
int[] ys = { 20, 110, 110 };
g.fillPolygon(xs, ys, 3);
```

Due array paralleli sono scomodi da costruire quando i vertici si calcolano uno alla volta, e AWT ha una classe apposta, `Polygon`, a cui si aggiungono i vertici con `addPoint`. Un poligono regolare di n lati inscritto in un cerchio di centro (c_x, c_y) e raggio r ha i vertici a intervalli di $2\pi/n$ sulla circonferenza:

```
public static Polygon regolare(int cx, int cy, int r, int n) {
    Polygon p = new Polygon();
    for (int i = 0; i < n; i++) {
        double a = 2 * Math.PI * i / n - Math.PI / 2;   // -90 gradi: in alto
        p.addPoint((int) Math.round(cx + r * Math.cos(a)),
                   (int) Math.round(cy + r * Math.sin(a)));
    }
    return p;
}
```

L'angolo di partenza, $-\pi/2$, mette il primo vertice in alto, ed è l'asse y capovolto a renderlo vero: in matematica $-90°$ è in basso, sullo schermo $\sin(-\pi/2) = -1$ sposta il punto *verso l'alto*. Per la stessa ragione, all'aumentare di i i vertici si susseguono in senso orario. Una stella a n punte è lo stesso ciclo con $2n$ vertici, alternati fra un raggio esterno e uno interno. Il poligono si disegna con `g.drawPolygon(p)` o `g.fillPolygon(p)`, e sa anche dire se contiene un punto, `p.contains(x, y)`: un'informazione che servirà per scegliere le figure con il mouse.

Attenzione 5.2: Il poligono che non si chiude

`drawPolyline` con gli stessi vertici di `drawPolygon` lascia aperto l'ultimo lato, e un triangolo diventa un angolo. Al contrario, una spezzata disegnata con `drawPolygon` si chiude con un segmento che nessuno voleva, dall'ultimo punto al primo. La mano libera della Lavagna è una `drawPolyline`: unire la fine del tratto con il suo inizio sarebbe un errore vistoso.

5.5 Colori e testo

Tutte le primitive usano il colore *corrente* del contesto, che si cambia con `setColor` e resta in vigore per tutto ciò che segue, finché non lo si cambia di nuovo. Un colore è un oggetto `java.awt.Color`: ci sono le costanti (`Color.RED`, `Color.DARK_GRAY`), il costruttore con le tre componenti rossa, verde e blu da 0 a 255, e un quarto parametro, l'*alfa*, per la trasparenza (0 trasparente, 255 opaco). `Color.getHSBColor(tinta, saturazione, luminosità)` lo costruisce a partire dalla tinta, comodo per generare colori diversi in un ciclo, e `brighter()` e `darker()` ne danno una versione più chiara o più scura.

Attenzione 5.3: Il colore di prima

Un metodo di disegno che non imposta il colore usa quello lasciato dall'ultimo che l'ha fatto, e il risultato dipende dall'ordine in cui le cose vengono disegnate. Ogni pezzo di disegno, ogni metodo `disegnaQualcosa`, comincia con il suo `setColor`; lo stesso vale per il font.

Il testo si scrive con `drawString(s, x, y)`, con il font corrente, che si cambia con `setFont(new Font(nome, stile, dimensione))`. Il punto (x, y) non è l'angolo in alto a sinistra del testo, come per un rettangolo: è l'inizio della *linea di base*, la riga immaginaria su cui poggiano le lettere. Le lettere salgono sopra la linea di base per l'*ascendente* e le code di lettere come "g" e "p" scendono sotto per il *discendente*. `drawString(s, 0, 0)` scrive quindi un testo quasi del tutto invisibile, sopra il bordo del componente.



Le misure di un font: il punto passato a `drawString` sta sulla linea di base, a sinistra.

Le misure le conosce un oggetto `FontMetrics`, che si ottiene dal contesto con `g.getFontMetrics()` e si riferisce al font corrente: `stringWidth(s)` è la larghezza in pixel di una stringa, `getAscent()` e `getDescent()` le due misure della figura, `getHeight()` la distanza fra due righe. Con queste si centra un testo su un punto, cosa che serve di continuo: le etichette di un grafico, il numero in una casella, il nome sotto una forma.

```
public static void scriviCentrato(Graphics g, String s, int cx, int cy) {
    FontMetrics fm = g.getFontMetrics();
    int x = cx - fm.stringWidth(s) / 2;                    // a sinistra
    int y = cy + (fm.getAscent() - fm.getDescent()) / 2;   // la linea di base
    g.drawString(s, x, y);
}
```

In orizzontale si toglie mezza larghezza. In verticale si vuole che il centro della parte visibile delle lettere cada su c_y: la linea di base va spostata in basso di mezza altezza delle lettere, e il conto con ascendente e discendente lo fa senza dipendere dal font.

5.6 L'esempio guida: la Lavagna che disegna

Tutto è pronto per dare alla Lavagna l'area che le manca. Il primo passo è decidere lo stato: che cosa deve sapere il pannello per poter ridisegnare tutto, in qualunque momento? Una lista di figure, e per ogni figura la forma, il colore e i punti del gesto con cui è stata tracciata. Una figura non cambia dopo essere stata disegnata, ed è un valore: un record.

```
public record Figura(Forma forma, Color colore, List<Point> punti) {

    public Figura {
        punti = List.copyOf(punti);         // una copia del gesto
    }

    /** Il rettangolo fra il primo e l'ultimo punto, in qualunque direzione. */
    public Rectangle rettangolo() {
        Point a = punti.get(0), b = punti.get(punti.size() - 1);
        return new Rectangle(Math.min(a.x, b.x), Math.min(a.y, b.y),
                Math.abs(b.x - a.x), Math.abs(b.y - a.y));
    }
}
```

Per un rettangolo e un'ellisse contano il primo e l'ultimo punto, gli angoli opposti del rettangolo che li contiene; `rettangolo()` li ordina, perché l'utente può trascinare il mouse verso l'alto o verso sinistra, e `drawRect` con larghezza negativa non disegna niente. Per la mano libera contano tutti i punti. Il costruttore compatto copia la lista: quella che riceve è la lista del gesto in corso, che il pannello svuota e riempie a ogni nuovo gesto.

Il pannello, `PnlLavagna`, tiene la lista delle figure e i punti del gesto in corso. Gli eventi del mouse, che il capitolo 8 tratterà per intero, qui servono in tre: la pressione comincia un gesto, il trascinamento lo allunga, il rilascio lo trasforma in una figura. `MouseAdapter` li raccoglie tutti, e lo stesso oggetto si registra come listener dei pulsanti (`addMouseListener`) e del movimento (`addMouseMotionListener`).

```
MouseAdapter mouse = new MouseAdapter() {
    @Override
    public void mousePressed(MouseEvent e) {
        gesto.clear();
        gesto.add(e.getPoint());
    }

    @Override
    public void mouseDragged(MouseEvent e) {
        gesto.add(e.getPoint());
        repaint();                              // la figura provvisoria
    }

    @Override
    public void mouseReleased(MouseEvent e) {
        if (gesto.size() > 1) {
            figure.add(new Figura(forma, colore, gesto));
            fireStateChanged();                 // le figure sono cambiate
        }
        gesto.clear();
        repaint();
    }
};
addMouseListener(mouse);
addMouseMotionListener(mouse);
```

Nessuno dei tre metodi disegna. Cambiano lo stato (i punti del gesto, la lista delle figure) e chiedono un ridisegno: è l'immagine guida, finalmente rispettata. Quando una figura si aggiunge, il pannello lo fa sapere con un `ChangeEvent` ai suoi listener, con lo schema del componente con i suoi eventi del capitolo 3: la Lavagna lo ascolta per segnare il disegno come modificato e aggiornare il conteggio nella riga di stato.

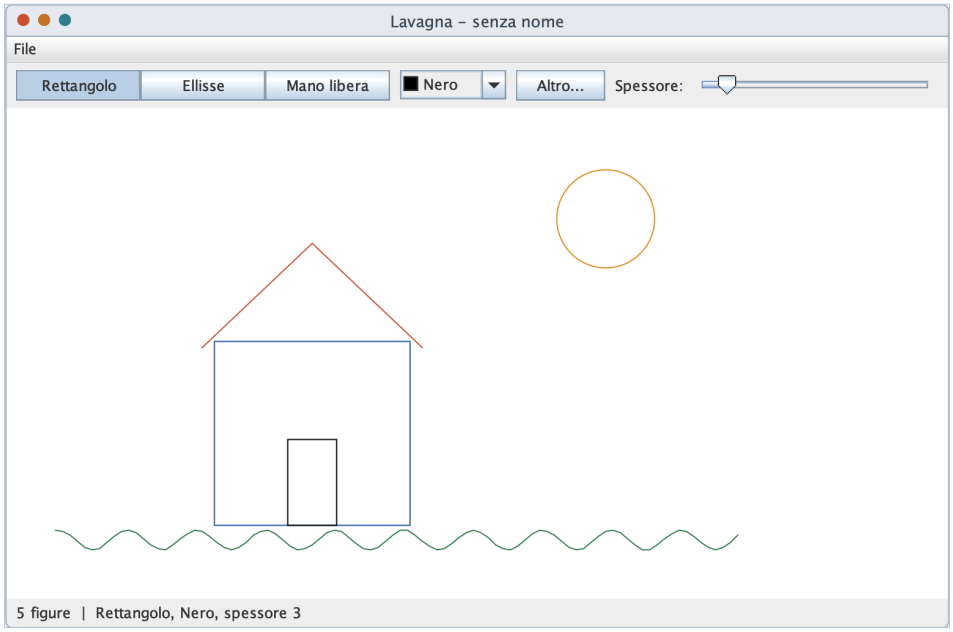
Il disegno è tutto in `paintComponent`: lo sfondo, la griglia se le proprietà la chiedono, le figure, e per ultima la figura che si sta tracciando, che è una figura come le altre costruita con i punti del gesto.

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);                    // lo sfondo
    if (griglia)
        disegnaGriglia(g);
    for (Figura f : figure)
        disegna(g, f);
    if (gesto.size() > 1)
        disegna(g, new Figura(forma, colore, gesto));   // quella in corso
}
```

Ogni figura si disegna con la primitiva che le corrisponde:

```
static void disegna(Graphics g, Figura f) {
    g.setColor(f.colore());
    switch (f.forma()) {
        case RETTANGOLO -> {
            Rectangle r = f.rettangolo();
            g.drawRect(r.x, r.y, r.width, r.height);
        }
        case ELLISSE -> {
            Rectangle r = f.rettangolo();
            g.drawOval(r.x, r.y, r.width, r.height);
        }
        case MANO_LIBERA -> {
            int n = f.punti().size();
            int[] xs = new int[n], ys = new int[n];
            for (int i = 0; i < n; i++) {
                xs[i] = f.punti().get(i).x;
                ys[i] = f.punti().get(i).y;
            }
            g.drawPolyline(xs, ys, n);
        }
    }
}
```

Lo `switch` sull'`enum` è esaustivo: se un giorno si aggiunge una forma, il compilatore segnala che manca il suo caso. Il resto della Lavagna cambia poco. La barra passa al pannello forma e colore scelti, nel metodo `aggiornaStato` che già ascoltava tutti i componenti; il file, oltre alle proprietà, contiene una riga per figura (la forma, il colore come intero RGB, i punti come coppie x, y), e la lettura ricostruisce la lista e la passa al pannello.



La Lavagna con le figure disegnate con le primitive: rettangoli, un'ellisse, due tratti a mano libera.

Ora il disegno si copre, si scopre, si ridimensiona, si salva e si riapre, e non sparisce più. Manca però una cosa che la barra promette: lo slider dello spessore non fa niente. Le primitive di `Graphics` disegnano solo linee larghe un pixel, e non c'è un metodo per cambiarlo. Serve la versione completa del contesto grafico, che è il tema del prossimo capitolo.

Ricetta: Un pannello che si disegna

1. Decidere lo stato: tutto ciò che serve per ridisegnare il pannello da capo, in campi del pannello.
2. Estendere `JPanel` e ridefinire `protected void paintComponent(Graphics g)` con `@Override`; prima istruzione `super.paintComponent(g)`.
3. In `paintComponent` solo lettura: misure da `getWidth()` e `getHeight()`, un `setColor` prima di ogni pezzo, e il disegno dello stato.
4. Chi cambia lo stato (un listener, un metodo pubblico) chiama `repaint()` subito dopo.

Esercizi

Esercizio 5.1: Che cosa si vede (⋆)

Disegnare su carta a quadretti, con un quadretto per pixel, ciò che produce questo `paintComponent` su un pannello bianco di 10 per 8 pixel.

```
super.paintComponent(g);
g.setColor(Color.BLACK);
g.fillRect(1, 1, 5, 4);
g.setColor(Color.WHITE);
g.drawOval(2, 2, 2, 2);
g.setColor(Color.RED);
g.drawLine(0, 7, 9, 7);
g.drawRect(6, 0, 3, 3);
```

Esercizio 5.2: Il bordo che non si vede (⋆)

Un pannello di 200 per 100 pixel disegna il suo bordo con `g.drawRect(0, 0, getWidth(), getHeight())`. Quali lati del rettangolo si vedono, e perché? Qual è la chiamata giusta? E per un bordo spesso 3 pixel, disegnato con tre `drawRect`?

Esercizio 5.3: La scacchiera ferma (⋆)

Una scacchiera calcola la dimensione della casella nel costruttore, `casella = Math.min(getWidth(), getHeight()) / 8`, e la usa in `paintComponent`. Che cosa si vede quando compare la finestra? E quando la si ridimensiona? Perché?

Esercizio 5.4: Senza `super` (⋆⋆)

Un pannello disegna una pallina in una posizione che cambia ogni volta che l'utente fa click, e chiama `repaint()`; in `paintComponent` abbiamo dimenticato `super.paintComponent(g)`. Che cosa vede l'utente dopo cinque click? E se il pannello estendesse `JComponent` invece di `JPanel`, con o senza la chiamata a `super`?

Esercizio 5.5: Il testo in alto a sinistra (⋆)

Che cosa si vede con `g.drawString("Titolo", 0, 0)`? E con `g.drawString("Titolo", 0, g.getFontMetrics().getAscent())`? Come si scrive "Titolo" centrato in orizzontale, a 10 pixel dal bordo superiore?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 5.6: La griglia (⋆)

Un pannello che disegna una griglia di 10 per 10 caselle quadrate, la più grande che ci sta, centrata, con le linee grigie e il bordo esterno nero. Le caselle sulla diagonale principale sono riempite di azzurro. La griglia deve adattarsi a ogni ridimensionamento della finestra.

Soluzione: progetto `sol-d05-c05`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 5.4: Il lato della griglia

Con caselle di c pixel, 10 caselle sono larghe $10c$, e le linee verticali sono 11, da x_0 a $x_0 + 10c$. Un ciclo che ne disegna 10 dimentica il bordo destro; e un lato calcolato come `getWidth() / 10` senza confrontarlo con l'altezza esce dal pannello in una finestra larga e bassa.

Esercizio 5.7: Il quadrante (⋆⋆)

Un pannello che disegna il quadrante di un orologio, centrato e grande quanto il lato minore: il cerchio, 60 tacche (più lunghe e più spesse, con due `drawLine` affiancati, quelle delle ore), i numeri da 1 a 12 centrati vicino alle tacche delle ore, e le tre lancette ferme sulle 10:10:30. Per le posizioni serve la trigonometria del poligono regolare: un angolo misurato dalle ore dodici in senso orario.

Soluzione: progetto `sol-d05-c05`, package `it.unibs.pajc.redolfi.e07`.

Esercizio 5.8: L'istogramma (⋆⋆)

Un pannello che riceve un array di valori interi positivi e un array di etichette e disegna un istogramma: una barra per valore, tutte della stessa larghezza, la più alta che occupa l'altezza disponibile meno lo spazio per il testo; sopra ogni barra il valore, sotto l'etichetta, entrambi centrati con `FontMetrics`. Colori diversi per ogni barra, generati con `Color.getHSBColor`.

Soluzione: progetto `sol-d05-c05`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 5.9: Poligoni e stelle (⋆⋆⋆)

Una finestra con uno `JSpinner` per il numero di lati n (da 3 a 20), una casella "Stella" e un pannello che disegna, centrato e grande quanto il lato minore, il poligono regolare di n lati oppure la stella a n punte, riempiti e con il contorno di un altro colore; i vertici sono segnati con un pallino e numerati. Lo spinner e la casella ridisegnano il pannello attraverso metodi del pannello, che non legge i componenti della finestra.

Soluzione: progetto `sol-d05-c05`, package `it.unibs.pajc.redolfi.e09`.

Attenzione 5.5: Il pannello che legge la finestra

Se `paintComponent` legge il valore dello spinner, il pannello dipende dalla finestra che lo contiene e non si può riusare. Il pannello ha il suo stato (`n`, `stella`) con i metodi per cambiarlo, che chiamano `repaint()`; è la finestra a chiamarli dai suoi listener.

Risposte

Risposta 5.1. Un rettangolo nero pieno di 5 per 4 pixel, dalle colonne 1–5 e dalle righe 1–4. Dentro, `drawOval(2, 2, 2, 2)` traccia il contorno di un'ellisse in un quadrato 2 per 2, che con il pixel in più occupa le colonne e le righe 2–4: un piccolo anello bianco, quasi un quadratino con gli angoli smussati, dentro il nero. In fondo una riga rossa su tutta la riga 7, dalla colonna 0 alla 9 compresa. In alto a destra il contorno rosso di un quadrato che occupa colonne 6–9 e righe 0–3: 4 per 4 pixel per un `drawRect` di lato 3.

Risposta 5.2. Si vedono solo il lato superiore e quello sinistro. Il rettangolo va da 0 a 200 in orizzontale e il contorno occupa la colonna 200, che non esiste in un pannello largo 200 (le colonne vanno da 0 a 199); lo stesso per la riga 100. La chiamata giusta è `g.drawRect(0, 0, getWidth() - 1, getHeight() - 1)`. Per tre pixel di spessore, tre rettangoli uno dentro l'altro: `drawRect(i, i, w - 1 - 2 * i, h - 1 - 2 * i)` per `i` da 0 a 2.

Risposta 5.3. Non si vede niente: nel costruttore il pannello non ha ancora una dimensione, il layout non l'ha ancora disposto, e `getWidth()` e `getHeight()` valgono 0. La casella vale 0 per sempre, e i `fillRect` di larghezza 0 non disegnano niente; ridimensionare non cambia nulla, perché il calcolo non si ripete. Anche se il valore fosse giusto all'inizio, al primo ridimensionamento la scacchiera resterebbe della misura iniziale. Le misure che dipendono dalla dimensione si calcolano in `paintComponent`.

Risposta 5.4. Con un `JPanel` senza `super.paintComponent` lo sfondo non viene ridipinto, e ogni ridisegno disegna la pallina nuova sopra ciò che c'era: dopo cinque click si vedono sei palline, la prima e le cinque nuove, o anche frammenti di altri componenti nelle zone che nessuno ridipinge; coprendo e scoprendo la finestra il risultato cambia ancora. L'aspetto dipende dalle circostanze, segno sicuro di un errore. Con un `JComponent` `super.paintComponent` non disegna comunque niente, e il risultato è lo stesso con o senza la chiamata; lo sfondo va disegnato a mano con un `fillRect` su tutto il componente.

Risposta 5.5. Con $y = 0$ la linea di base coincide con il bordo superiore, e del testo si vedono al più le code delle lettere che scendono sotto la linea di base. Con y uguale all'ascendente le lettere toccano il bordo superiore senza uscire. Centrato a 10 pixel dal bordo: `FontMetrics fm = g.getFontMetrics();` e poi `g.drawString("Titolo", (getWidth() - fm.stringWidth("Titolo")) / 2, 10 + fm.getAscent())`.

Capitolo 6

`Graphics2D`: forme, tratti e colori

Il capitolo 5 ha lasciato la Lavagna con un difetto visibile: lo slider dello spessore non serve a niente, perché le primitive di `Graphics` tracciano soltanto linee larghe un pixel. Non è l'unico limite: le coordinate sono intere, i bordi delle forme oblique sono scalettati, non ci sono sfumature né trasparenze, e una forma esiste solo nel momento in cui la si disegna. Tutte queste cose le offre una seconda versione del contesto grafico, `Graphics2D`, che Swing usa da sempre senza dirlo. Questo capitolo la presenta: le forme come oggetti che si costruiscono, si combinano e si interrogano; il tratto, con il suo spessore e il suo stile; il colore come pittura, con sfumature e trasparenza; e la regola per non lasciare il contesto in disordine. Alla fine le figure della Lavagna sono forme, disegnate con lo spessore scelto nella barra.

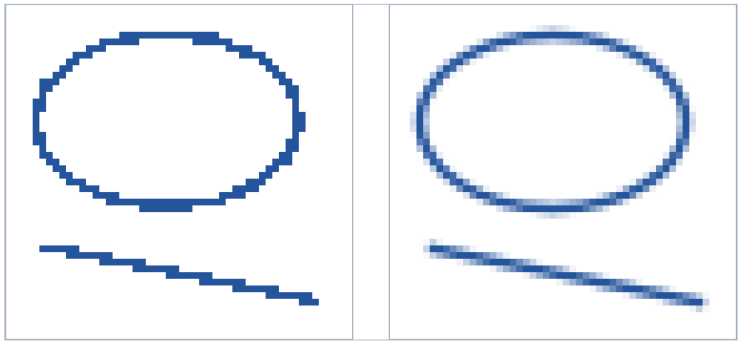
6.1 Perché un secondo contesto

`Graphics` è la classe della prima versione di Java, pensata per gli schermi di allora. Qualche anno dopo la libreria ha ricevuto un sistema grafico completo, chiamato *Java 2D*, che ragiona come un programma di grafica vettoriale: coordinate reali, forme qualsiasi, linee di qualunque spessore, colori sfumati e trasparenti, trasformazioni del sistema di riferimento. Per non rompere i programmi esistenti, Java 2D non ha cambiato `Graphics`: ne ha creato una sottoclasse, `Graphics2D`, e da allora il contesto grafico che Swing passa a `paintComponent` è sempre un `Graphics2D`, anche se il parametro è dichiarato `Graphics`. Per usarne i metodi basta un cast:

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
    // ... da qui si disegna con g2
}
```

Il cast non può fallire in un componente Swing, ed è così comune che molti programmi lo fanno come prima cosa in ogni `paintComponent`. Tutto ciò che si è visto nel capitolo precedente resta valido: `Graphics2D` è un `Graphics`, e `drawRect` o `drawString` funzionano come prima.

La seconda riga dell'esempio attiva l'*antialiasing*. Un bordo obliquo o curvo disegnato su una griglia di pixel diventa una scaletta: ogni pixel è acceso o spento, e i gradini si vedono. Con l'antialiasing i pixel attraversati solo in parte dal bordo prendono un colore intermedio, proporzionale a quanto ne è coperto, e l'occhio vede una linea continua. La differenza si apprezza ingrandendo:



La stessa ellisse e lo stesso segmento, ingranditi sei volte: senza antialiasing (a sinistra) e con (a destra).

L'antialiasing è un *suggerimento* (*rendering hint*): il contesto lo segue se può, e un suggerimento resta in vigore come il colore, finché non lo si cambia. Per il testo ce n'è uno a parte, `KEY_TEXT_ANTIALIASING`, anche se Swing di solito lo imposta già in base alle preferenze del sistema. Costa un po' di tempo di disegno, che su qualunque computer recente non si nota, e si attiva quasi sempre; si lascia spento quando si vogliono pixel netti, per esempio nelle righe di una griglia sottile, che con l'antialiasing diventerebbero grigie e sfocate.

6.2 Le forme come oggetti

Con `Graphics` una forma esiste solo come chiamata: `drawOval(10, 10, 80, 40)` disegna un'ellisse e se ne dimentica. Java 2D rovescia l'idea: una forma è un *oggetto*, che si costruisce una volta, si conserva in un campo, si disegna quando serve e si può interrogare.

Definizione 6.1: `Shape`

`java.awt.Shape` è l'interfaccia delle forme geometriche di Java 2D: una regione del piano delimitata da un contorno. Sa descrivere il proprio contorno, dire se contiene un punto o un rettangolo (`contains`), se interseca un rettangolo (`intersects`) e qual è il rettangolo che la racchiude (`getBounds2D`). `Graphics2D` ne traccia il contorno con `draw(shape)` e ne riempie l'interno con `fill(shape)`.

Le forme di uso comune stanno nel package `java.awt.geom`. Ognuna è una classe astratta con due sottoclassi annidate, `Double` e `Float`, che differiscono solo per il tipo delle coordinate; si usa quasi sempre `Double`.

Classe	Costruttore (versione `Double`)
`Line2D`	`new Line2D.Double(x1, y1, x2, y2)`
`Rectangle2D`	`new Rectangle2D.Double(x, y, w, h)`
`RoundRectangle2D`	`new RoundRectangle2D.Double(x, y, w, h, aw, ah)`
`Ellipse2D`	`new Ellipse2D.Double(x, y, w, h)`
`Arc2D`	`new Arc2D.Double(x, y, w, h, inizio, ampiezza, tipo)`
`Path2D`	`new Path2D.Double()`, poi `moveTo`, `lineTo`, ...
`Area`	`new Area(shape)`, poi `add`, `subtract`, ...

I parametri sono quelli delle primitive del capitolo precedente, ma reali: un'ellisse può avere il centro in $(100.5, 40.25)$, e con l'antialiasing la differenza di mezzo pixel si vede. `Arc2D` ha in più il tipo: `OPEN` è il solo arco, `CHORD` lo chiude con una corda, `PIE` con due raggi, come la fetta di `fillArc`.

La forma come oggetto cambia il modo di scrivere il disegno. Il cerchio centrato del capitolo 5 si costruisce una volta per chiamata, con il suo centro e il suo raggio, e poi si riempie e si contorna senza ripetere i numeri:

```
double r = Math.min(getWidth(), getHeight()) / 2.0 - 10;
Shape cerchio = new Ellipse2D.Double(getWidth() / 2.0 - r, getHeight() / 2.0 - r,
        2 * r, 2 * r);
g2.setColor(new Color(0xF4, 0xC5, 0xB4));
g2.fill(cerchio);                       // l'interno
g2.setColor(new Color(0xC9, 0x4F, 0x2E));
g2.draw(cerchio);                       // e il contorno, della stessa forma
```

E una forma può rispondere a domande. Se il programma tiene in un campo le forme che ha disegnato, sapere su quale di esse l'utente ha fatto click è una chiamata a `contains(e.getPoint())`; sapere se due oggetti di un gioco si toccano è un `intersects`. Lo stesso oggetto serve al disegno e alla geometria, ed è questa la ragione per cui le figure della Lavagna diventeranno forme.

6.3 Percorsi e aree

Le forme della tabella coprono i casi semplici. Per tutto il resto ci sono due strumenti: i percorsi, che descrivono un contorno passo per passo, e le aree, che combinano forme già esistenti.

6.3.1 Percorsi

Un `Path2D` si costruisce come si disegnerebbe a mano con una penna: `moveTo` alza la penna e la porta in un punto, `lineTo` traccia un segmento fino a un altro punto, `closePath` torna al punto di partenza chiudendo il contorno. È il `Polygon` del capitolo precedente, ma con coordinate reali e con la possibilità di fare più contorni separati nello stesso percorso. La stella a n punte diventa:

```
public static Path2D stella(double cx, double cy,
                            double rEst, double rInt, int n) {
    Path2D p = new Path2D.Double();
    for (int i = 0; i < 2 * n; i++) {
        double a = Math.PI * i / n - Math.PI / 2;
        double r = i % 2 == 0 ? rEst : rInt;
        double x = cx + r * Math.cos(a), y = cy + r * Math.sin(a);
        if (i == 0)
            p.moveTo(x, y);                     // il primo: si alza la penna
        else
            p.lineTo(x, y);
    }
    p.closePath();                              // torna al primo vertice
    return p;
}
```

I percorsi sanno anche curvare. `quadTo(cx, cy, x, y)` traccia una curva fino a (x, y) attirata dal *punto di controllo* (c_x, c_y), che la curva non tocca; `curveTo` fa lo stesso con due

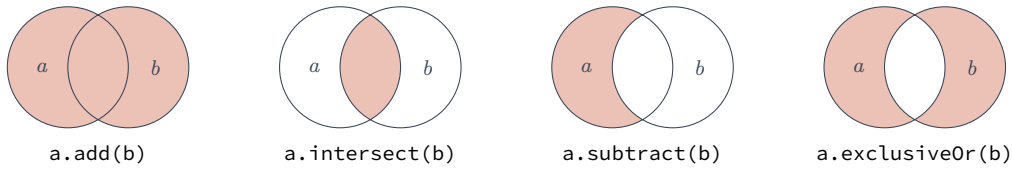
punti di controllo, uno vicino all'inizio e uno vicino alla fine, e permette curve a S. Sono le *curve di Bézier* dei programmi di grafica, e con due di esse, simmetriche, si fa un cuore:

```
Path2D p = new Path2D.Double();
p.moveTo(cx, cy + r);                                    // la punta, in basso
p.curveTo(cx - 1.6 * r, cy - 0.2 * r, cx - 0.6 * r, cy - 1.3 * r,   // controlli
          cx, cy - 0.45 * r);                                        // l'incavo
p.curveTo(cx + 0.6 * r, cy - 1.3 * r, cx + 1.6 * r, cy - 0.2 * r,
          cx, cy + r);                                               // la punta
p.closePath();
```

La prima curva sale a sinistra fino all'incavo in alto, la seconda scende a destra fino alla punta; i punti di controllo, fuori dalla forma, ne decidono la rotondità. Trovare i numeri giusti per una curva è questione di prove, e un buon metodo è disegnare anche i punti di controllo, con dei pallini, finché la forma non convince.

6.3.2 Aree

Molte forme si descrivono meglio come combinazioni di forme semplici che come contorni: un anello è un cerchio meno un cerchio, una luna è un cerchio meno un cerchio spostato, una chiave inglese è un rettangolo più due cerchi meno due rettangoli. `Area` trasforma una forma qualunque in una regione del piano su cui si fanno le operazioni degli insiemi.

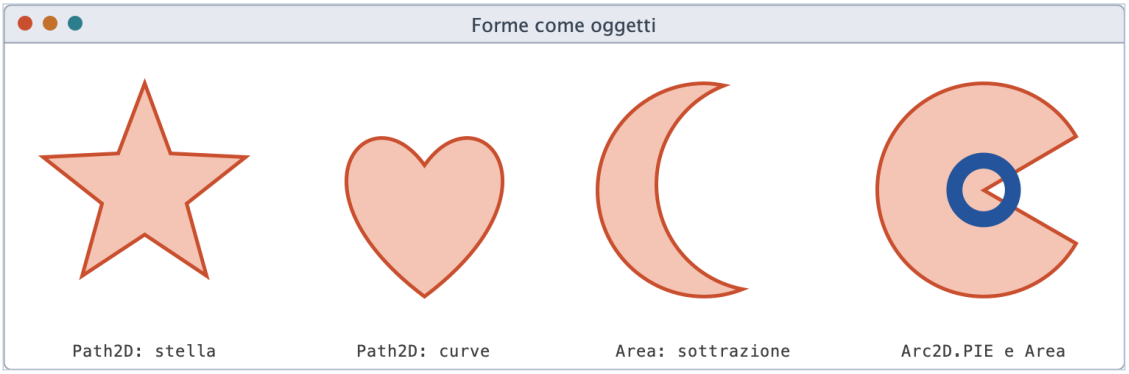


Le quattro operazioni di `Area` su due cerchi: unione, intersezione, differenza, differenza simmetrica. In colore il risultato in `a`.

Le operazioni modificano l'area su cui sono chiamate. La luna si ottiene così:

```
Area luna = new Area(new Ellipse2D.Double(cx - r, cy - r, 2 * r, 2 * r));
Shape morso = new Ellipse2D.Double(cx - 0.45 * r, cy - 1.05 * r, 2 * r, 2 * r);
luna.subtract(new Area(morso));
g2.fill(luna);
```

Un'`Area` è una `Shape`, e si disegna, si riempie e si interroga come le altre. Le operazioni non sono gratuite, perché calcolano il contorno del risultato, e conviene farle una volta e conservare il risultato, non ripeterle a ogni ridisegno.



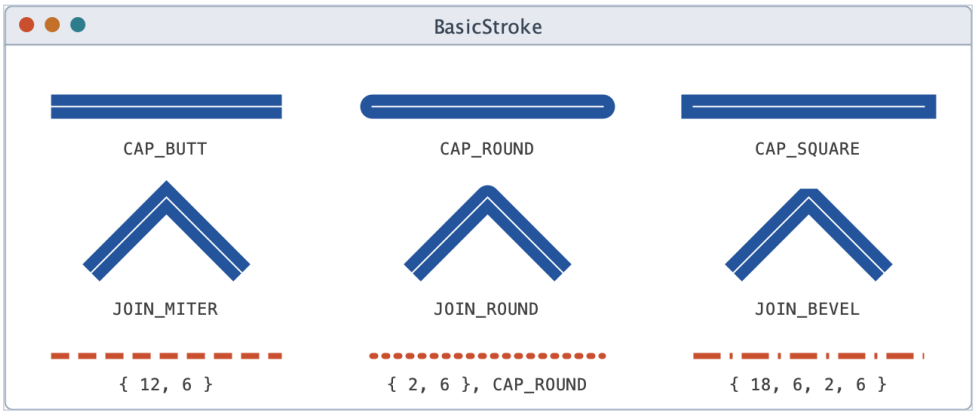
Forme come oggetti: una stella e un cuore con `Path2D`, una luna per sottrazione, un settore con un anello.

6.4 Il tratto

Con `draw` il contorno di una forma viene tracciato con il *tratto* corrente, un oggetto `Stroke` che il contesto conserva come il colore e che si cambia con `setStroke`. Il tratto di Java 2D è `BasicStroke`, e ha cinque caratteristiche, tutte nel costruttore: lo spessore, la forma degli estremi, la forma delle giunzioni, un limite per gli spigoli acuti e l'eventuale tratteggio.

```
g2.setStroke(new BasicStroke(6));                  // solo lo spessore
g2.setStroke(new BasicStroke(6, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
float[] trattini = { 12, 6 };                      // 12 pieni, 6 vuoti, ...
g2.setStroke(new BasicStroke(4, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER,
        10, trattini, 0));                         // limite spigoli, fase
```

Gli *estremi* sono le terminazioni di una linea aperta: `CAP_BUTT` la taglia esattamente dove finisce, `CAP_ROUND` la arrotonda con un semicerchio, `CAP_SQUARE` la prolunga di mezzo spessore. Le *giunzioni* sono gli angoli fra due segmenti consecutivi: `JOIN_MITER` li fa a punta, `JOIN_ROUND` li arrotonda, `JOIN_BEVEL` li smussa. Il tratteggio è un array di lunghezze che si alternano, pieno e vuoto, e la *fase* dice da che punto dell'array si comincia: farla crescere nel tempo fa scorrere i trattini, l'effetto delle "formiche in marcia" delle selezioni.



Estremi, giunzioni e tratteggi; la linea bianca sottile è il contorno geometrico.

La schermata mostra un fatto che conviene tenere a mente: il tratto è *centrato* sul contorno geometrico. Metà dello spessore sta da una parte, metà dall'altra. Un rettangolo grande quanto il pannello, contornato con un tratto di 10 pixel, perde la metà esterna di ogni lato, che cade fuori dal pannello e non si vede; per vederlo intero lo si costruisce rientrato di mezzo spessore.

Attenzione 6.1: Il tratto che resta

Come il colore, il tratto impostato con `setStroke` resta in vigore per tutto ciò che segue, compreso ciò che disegna Swing dopo `paintComponent` con lo stesso contesto. Un pannello che imposta un tratto di 8 pixel per l'ultima figura e lo lascia lì può ritrovarsi con il bordo del pannello, disegnato subito dopo, spesso 8 pixel. Il rimedio, valido per tutto lo stato del contesto, è nella sezione 6.6.

6.5 Il colore come pittura

In Java 2D il colore è un caso particolare di qualcosa di più generale, la *pittura* (`Paint`): la regola che decide il colore di ogni pixel che una forma riempie o un tratto attraversa. Un `Color` è la pittura più semplice, lo stesso colore ovunque; `setPaint` accetta anche pitture che cambiano colore da un punto all'altro.

```
// dal blu in alto a sinistra al rosa in basso a destra
g2.setPaint(new GradientPaint(20, 20, BLU, 200, 160, ROSA));
g2.fill(new Rectangle2D.Double(20, 20, 180, 140));

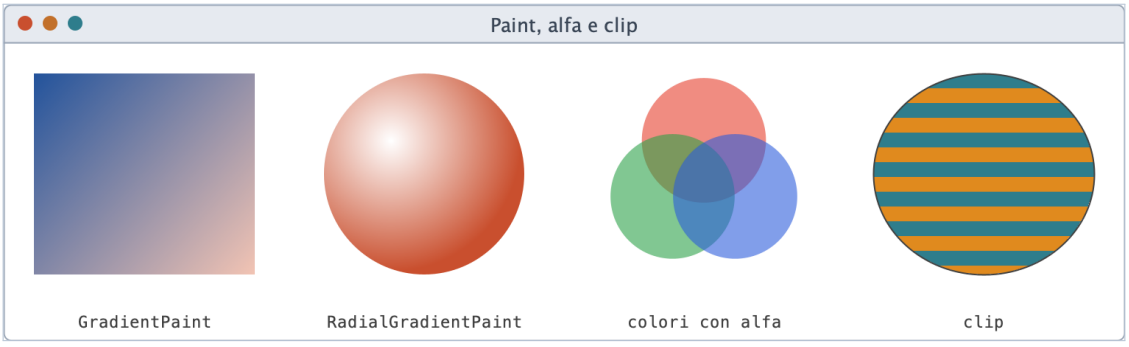
// dal bianco al centro (spostato in alto a sinistra) al rosso sul bordo
g2.setPaint(new RadialGradientPaint(centroLuce, raggio, new float[] { 0f, 1f },
        new Color[] { Color.WHITE, ROSSO }));
g2.fill(cerchio);
```

`GradientPaint` sfuma linearmente fra due colori assegnati a due punti; oltre i due punti il colore resta quello dell'estremo. `RadialGradientPaint` sfuma a cerchi concentrici, e con il centro della sfumatura spostato rispetto al centro della forma dà l'illusione di una sfera illuminata.

La *trasparenza* si ottiene con il quarto componente di un colore, l'alfa: con `new Color(230, 60, 40, 150)` il rosso copre per 150 parti su 255, e il resto lascia vedere ciò che c'è sotto. Per rendere trasparente tutto ciò che si disegna, immagini comprese, si cambia invece la regola di composizione del contesto:

```
g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f));
```

Infine il *clip*, la regione entro cui il disegno è visibile. Il contesto che `paintComponent` riceve ha già un clip, il rettangolo del componente da ridisegnare; `g2.clip(shape)` lo restringe all'intersezione con una forma qualunque, e tutto ciò che si disegna dopo resta dentro quella forma. È il modo più semplice di riempire una forma con un motivo, o di disegnare solo dentro una finestra rotonda.



Le pitture: una sfumatura lineare, una radiale, tre cerchi con alfa, righe colorate tagliate da un clip ellittico.

6.6 Il contesto da non sporcare

Il contesto grafico ha uno stato, ormai lungo: colore o pittura, font, tratto, suggerimenti, composizione, clip, e nel prossimo capitolo il sistema di riferimento. Ogni `set...` cambia lo stato per tutto ciò che segue. E ciò che segue non è solo il resto del nostro `paintComponent`: Swing usa lo stesso contesto, dopo di noi, per disegnare il bordo del componente. La documentazione di `paintComponent` è esplicita: il metodo non deve fare cambiamenti permanenti al contesto che riceve, e se ne ha bisogno conviene che ne crei uno nuovo.

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g.create();    // una copia, con lo stesso stato
    try {
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setStroke(new BasicStroke(8));
        // ... il disegno, con tutte le impostazioni che servono
    } finally {
        g2.dispose();                           // la copia si butta
    }
}
```

`create()` restituisce un contesto nuovo, che disegna sulla stessa superficie e parte dallo stesso stato dell'originale; i cambiamenti fatti sulla copia non toccano l'originale. `dispose()` libera la copia quando non serve più: è una risorsa del sistema grafico, e va rilasciata come un file, per questo sta in un `finally`. Lo stesso schema serve anche *dentro* un `paintComponent`, per isolare il disegno di un singolo oggetto che cambia molte impostazioni: una copia per l'oggetto, e alla fine il contesto principale è com'era. Nel capitolo 7, con le trasformazioni, diventerà indispensabile.

Osservazione 6.1

`dispose` si chiama solo sui contesti creati da noi con `create()`. Il contesto `g` ricevuto come parametro appartiene a Swing, che lo usa ancora dopo `paintComponent`: chiamarne `dispose` è un errore che, a seconda del sistema, fa sparire il bordo o i componenti figli.

6.7 L'esempio guida: figure con lo spessore

Le figure della Lavagna diventano forme. Il record conserva ciò che serve a ricostruirle (il tipo, il colore, lo spessore e i punti del gesto) e calcola la forma geometrica quando gli viene chiesta. Il campo che nel capitolo precedente si chiamava `forma` diventa `tipo`, perché ora la forma è la `Shape`:

```
public record Figura(Forma tipo, Color colore, float spessore,
                     List<Point> punti) {

    public Shape forma() {
        Point a = punti.get(0), b = punti.get(punti.size() - 1);
        double x = Math.min(a.x, b.x), y = Math.min(a.y, b.y);
        double w = Math.abs(b.x - a.x), h = Math.abs(b.y - a.y);
        return switch (tipo) {
            case RETTANGOLO -> new Rectangle2D.Double(x, y, w, h);
            case ELLISSE -> new Ellipse2D.Double(x, y, w, h);
            case MANO_LIBERA -> {
                Path2D p = new Path2D.Double();
                p.moveTo(a.x, a.y);
                for (Point q : punti.subList(1, punti.size()))
                    p.lineTo(q.x, q.y);
                yield p;
            }
        };
    }
}
```

La forma non si memorizza: è un dato *derivato*, che si ricava dai punti, e salvarla accanto ai punti significherebbe tenere due versioni della stessa informazione, con il rischio che un giorno non coincidano. Il costruttore compatto, che copia la lista dei punti, è quello del capitolo precedente.

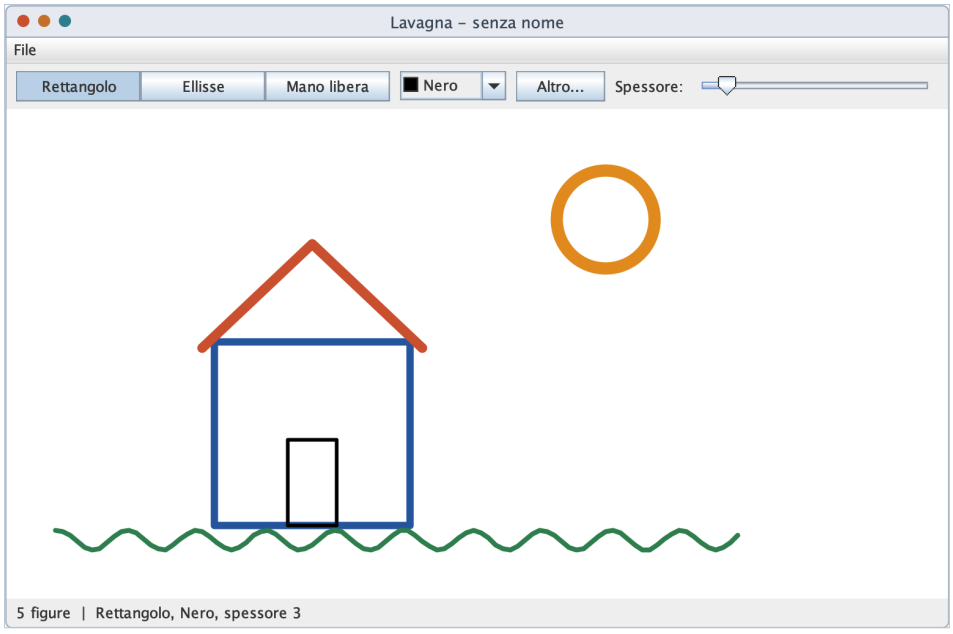
Il disegno di una figura si riduce a tre righe, e lo `switch` sulle primitive sparisce: ogni figura sa qual è la sua forma, e il pannello la traccia con il suo colore e il suo tratto, a estremi e giunzioni arrotondati, come una penna vera.

```
static void disegna(Graphics2D g2, Figura f) {
    g2.setColor(f.colore());
    g2.setStroke(new BasicStroke(f.spessore(), BasicStroke.CAP_ROUND,
            BasicStroke.JOIN_ROUND));
    g2.draw(f.forma());
}
```

`paintComponent` lavora su una copia del contesto, con l'antialiasing, e per il resto è quello del capitolo 5:

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);                    // lo sfondo
    Graphics2D g2 = (Graphics2D) g.create();    // le impostazioni restano qui
    try {
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        if (griglia)
            disegnaGriglia(g2);
        for (Figura f : figure)
            disegna(g2, f);
        if (gesto.size() > 1)
            disegna(g2, new Figura(forma, colore, spessore, gesto));
    } finally {
        g2.dispose();
    }
}
```

Lo spessore arriva dalla barra come la forma e il colore: `aggiornaStato` lo passa al pannello con `setSpessore(sldSpessore.getValue())`. Il file registra anche lo spessore di ogni figura, e le righe salvate nel capitolo precedente non si leggono più: un problema reale dei formati di file che cambiano, che qui si risolve cambiando l'intestazione in "LAVAGNA 2".



La Lavagna con le figure di Java 2D: ogni figura ha il suo spessore, e i bordi sono smussati dall'antialiasing.

Le forme portano con sé un'ultima possibilità, che il capitolo 8 userà per cancellare una figura con il tasto destro: trovare la figura sotto il puntatore. Per un'ellisse piena basterebbe `contains`, ma le figure della Lavagna sono contorni, e un click al centro di un rettangolo vuoto non lo tocca. Il tratto stesso sa costruire la forma della linea che traccia: `createStrokedShape` restituisce la `Shape` occupata dal contorno disegnato, e un tratto un po' più spesso di quello vero lascia all'utente qualche pixel di tolleranza.

```
public Figura figuraIn(Point p) {
    for (int i = figure.size() - 1; i >= 0; i--) {   // dall'ultima, la piu' in
    alto
        Figura f = figure.get(i);
        Stroke largo = new BasicStroke(f.spessore() + 6);
        Shape bordo = largo.createStrokedShape(f.forma());
        if (bordo.contains(p))
            return f;
    }
    return null;
}
```

Ricetta: Disegnare con `Graphics2D`

1. In `paintComponent`, dopo `super.paintComponent(g)`, una copia: `Graphics2D g2 = (Graphics2D) g.create()`, e `g2.dispose()` in un `finally`.
2. L'antialiasing con `setRenderingHint`, salvo dove servono pixel netti.
3. Le forme come oggetti di `java.awt.geom`, costruite dai dati; `Path2D` per i contorni, `Area` per le combinazioni, calcolate una volta sola se costose.
4. Per ogni forma colore o pittura, tratto, e poi `fill` o `draw`.

Esercizi

Esercizio 6.1: Il bordo spesso (⋆)

Un pannello con un bordo di linea, assegnato con `setBorder`, disegna in `paintComponent`, sul contesto ricevuto e senza copiarlo, un'ultima figura con `g2.setStroke(new BasicStroke(8))`. Che cosa si nota nel bordo del pannello? Come si corregge, in due modi?

Esercizio 6.2: Mezzo spessore (⋆)

Un pannello di 200 per 100 pixel traccia `new Rectangle2D.Double(0, 0, 200, 100)` con un tratto di 10 pixel. Quanto è spesso ciascun lato visibile? Quale rettangolo bisogna costruire perché il contorno si veda tutto, spesso 10, a filo del bordo?

Esercizio 6.3: Tre cerchi (⋆)

`a`, `b` e `c` sono tre `Area` costruite da tre cerchi uguali i cui centri formano un triangolo equilatero, sovrapposti come nel diagramma di Venn della schermata delle pitture. Descrivere la regione che resta in `a` dopo `a.add(b); a.intersect(c);` e dopo, partendo da capo, `a.intersect(b); a.subtract(c);`.

Esercizio 6.4: Dentro o sopra (⋆⋆)

Una figura della Lavagna è un rettangolo vuoto di 200 per 100 pixel, con il contorno spesso 3. L'utente fa click al centro del rettangolo. Che cosa restituisce `f.forma().contains(p)`? E `figuraIn(p)`? Quale delle due è la risposta giusta per una lavagna, e quale sarebbe giusta per un programma in cui le figure sono piene?

Esercizio 6.5: L'ordine dei trasparenti (⋆)

Su uno sfondo bianco si riempie un quadrato blu opaco e sopra, parzialmente sovrapposto, un quadrato rosso con alfa 128. In un secondo pannello si fa il contrario: prima il rosso con alfa 128, poi il blu opaco. La zona di sovrapposizione ha lo stesso colore nei due pannelli? Perché?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 6.6: Il tratteggio (⋆)

Una finestra con tre slider (spessore da 1 a 20, lunghezza dei trattini da 2 a 40, fase da 0 a 40) e una casella "estremi arrotondati", e un pannello che traccia con quel tratto un rettangolo dagli angoli arrotondati, un cerchio e una spezzata a zig-zag. Il pannello ha i suoi metodi per ricevere i valori, e si ridisegna a ogni cambiamento.

Soluzione: progetto `sol-d05-c06`, package `it.unibs.pajc.redolfi.e06`.

Esercizio 6.7: La casetta (⋆⋆)

Un pannello che disegna, centrata e scalata sul lato minore, una casetta come pittogramma pieno, di un solo colore: il corpo è un rettangolo con la porta e due finestre *sottratte* (si vede lo sfondo attraverso), il tetto è un triangolo costruito con `Path2D` e unito al corpo, il comignolo un rettangolo unito al tetto. Tutta la casetta è una sola `Area`, costruita fuori da `paintComponent` quando cambia la dimensione del pannello, e riempita con una sfumatura verticale.
Suggerimento: un `ComponentListener` con `componentResized` avvisa quando il pannello cambia dimensione; in alternativa si ricostruisce l'area in `paintComponent` solo se larghezza o altezza sono diverse da quelle dell'ultima volta.

Soluzione: progetto `sol-d05-c06`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 6.2: Le operazioni a ogni ridisegno

Costruire l'`Area` della casetta dentro `paintComponent`, a ogni chiamata, funziona, ma ricalcola le stesse operazioni durante ogni ridimensionamento e ogni copertura della finestra. Su una forma semplice non si nota; su una forma con molte operazioni il ridisegno rallenta. Il risultato si conserva e si ricalcola solo quando cambiano i dati da cui dipende.

Esercizio 6.8: Il cerchio cromatico (⋆⋆)

Un pannello che disegna un cerchio cromatico: n settori (`Arc2D` di tipo `PIE`) di uguale ampiezza, ciascuno riempito con la tinta corrispondente alla sua posizione (`Color.getHSBColor(i / (float) n, 1, 1)`), e al centro un cerchio bianco che lo trasforma in una corona. Uno slider sceglie n da 3 a 72; il cerchio è centrato e grande quanto il lato minore, con l'antialiasing.

Soluzione: progetto `sol-d05-c06`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 6.9: L'ingranaggio (⋆⋆⋆)

Un pannello che disegna un ingranaggio con n denti (da 6 a 40, scelto con uno spinner): il profilo è un unico `Path2D` che alterna tratti sul raggio esterno, dove stanno i denti, e tratti sul raggio interno, calcolati con la trigonometria; al centro un foro tondo sottratto con `Area`. L'ingranaggio è riempito con una sfumatura radiale, contornato con un tratto di 2 pixel a giunzioni arrotondate, centrato e grande quanto il lato minore.
Suggerimento: ogni dente occupa un angolo di $2\pi/n$; dividerlo in quattro parti (salita, cima, discesa, fondo) dà i quattro vertici per dente.

Soluzione: progetto `sol-d05-c06`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 6.1. Il bordo, che Swing disegna dopo `paintComponent` con lo stesso contesto, viene tracciato con il tratto lasciato impostato, spesso 8 invece di 1: il bordo del pannello diventa una cornice grossa, e lo diventa anche quello di eventuali componenti figli. Primo rimedio: lavorare su una copia, `(Graphics2D) g.create()` con `dispose()` in un `finally`, così nessuna impostazione esce da `paintComponent`. Secondo, più fragile: salvare il tratto all'inizio con `getStroke()` e ripristinarlo alla fine con `setStroke`; funziona finché ci si ricorda di tutte le impostazioni cambiate.

Risposta 6.2. Il tratto è centrato sul contorno: metà dei 10 pixel cade fuori dal pannello e non si vede, e ogni lato appare spesso 5 pixel. Per vederlo intero si rientra di mezzo spessore su ogni lato: `new Rectangle2D.Double(5, 5, 190, 90)`. Il contorno occupa allora la fascia da 0 a 10 pixel dal bordo, tutta dentro il pannello.

Risposta 6.3. Dopo `a.add(b)` l'area è l'unione dei primi due cerchi; l'intersezione con `c` tiene la parte di quell'unione che sta anche in `c`: le due lenti fra `c` e ciascuno degli altri due cerchi, unite (comprendono la zona centrale comune a tutti e tre). Nel secondo caso `a.intersect(b)` è la lente fra i primi due cerchi; sottrarre `c` toglie la parte centrale comune a tutti e tre e lascia la porzione di lente fuori da `c`: nel diagramma di Venn, la zona che appartiene ad `a` e `b` ma non a `c`.

Risposta 6.4. `contains` restituisce `true`: un `Rectangle2D` contiene tutti i punti del suo interno, anche se il disegno ne traccia solo il contorno. `figuraIn` restituisce `null`, perché la forma del tratto, anche ingrossata di 6 pixel, è una cornice larga pochi pixel attorno al contorno, e il centro ne resta fuori. Per una lavagna, dove le figure sono linee, la risposta giusta è la seconda: si cancella una figura toccandone il tratto. Con figure piene, come le forme di un programma di disegno vettoriale, sarebbe giusto `contains`.

Risposta 6.5. No. Nel primo pannello il rosso semitrasparente sta sopra il blu, e la sovrapposizione è una media fra rosso e blu, un viola. Nel secondo il blu opaco arriva per ultimo e copre completamente ciò che c'è sotto: la sovrapposizione è blu pieno. La composizione `SRC_OVER` mescola il colore nuovo con quello già presente secondo l'alfa del nuovo: l'ordine conta, come quando si sovrappongono fogli colorati su un tavolo.

Capitolo 7

Coordinate e trasformazioni

Finora ogni disegno è stato pensato in pixel, con l'origine in alto a sinistra e l'asse y verso il basso. È il sistema del componente, ma raramente è quello del problema. Il grafico di una funzione ragiona in unità matematiche, con l'origine dove capita e la y verso l'alto; un sistema solare in milioni di chilometri; una lavagna che si può ingrandire in coordinate del disegno, che non cambiano quando cambia lo zoom. Questo capitolo mostra due modi di passare da un sistema all'altro: scrivere le funzioni di conversione, e chiedere al contesto grafico di *trasformare* il proprio sistema di riferimento, con traslazioni, rotazioni e cambi di scala. Il secondo modo è quello che rende semplici le cose difficili, e ha regole precise, che si vedono una alla volta. Alla fine la Lavagna ha l'origine al centro, una griglia vera e lo zoom.

7.1 Due sistemi di riferimento

Il grafico di $f(x) = \sin(x)/x$ fra -15 e 15 si disegna in un pannello di qualche centinaio di pixel. Il punto $(0, 1)$ della funzione deve finire da qualche parte al centro del pannello, il punto $(-15, 0)$ sul bordo sinistro, e i valori positivi di y sopra, non sotto. Ci sono quindi due sistemi di riferimento: quello del problema, che chiameremo il *mondo*, e quello dei pixel. Disegnare vuol dire portare ogni punto del mondo nel suo pixel.

La conversione è una proporzione per ciascun asse. Se il mondo va da $x_{\min}$ a $x_{\max}$ e il pannello ha larghezza w, il punto x del mondo cade alla colonna

$$p_x = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \cdot w;$$

per l'asse verticale si fa lo stesso partendo dall'alto, perché la y dei pixel cresce verso il basso:

$$p_y = \frac{y_{\max} - y}{y_{\max} - y_{\min}} \cdot h.$$

In codice sono due metodi, a cui conviene affiancare subito l'inverso, che serve per sapere a quale punto del mondo corrisponde il pixel sotto il mouse. I nomi sono una convenzione che useremo sempre: `worldToScreen` per la strada dal mondo ai pixel, `screenToWorld` per quella opposta, con `X` o `Y` in fondo quando convertono un solo asse.

```
private double worldToScreenX(double x) {
    return SINISTRA + (x - xMin) / (xMax - xMin) * larghezza();
}

private double worldToScreenY(double y) {
    return SOPRA + (yMax - y) / (yMax - yMin) * altezza();    // y verso l'alto
}

private double screenToWorldX(int pixel) {
    return xMin + (double) (pixel - SINISTRA) / larghezza() * (xMax - xMin);
}
```

`SINISTRA` e `SOPRA` sono i margini lasciati per le etichette degli assi, e `larghezza()` e `altezza()` la parte di pannello che resta. Con queste funzioni il disegno si scrive nel mondo e si converte punto per punto, come il segmento da (x_1, y_1) a (x_2, y_2):

```
g2.draw(new Line2D.Double(worldToScreenX(x1), worldToScreenY(y1),
                          worldToScreenX(x2), worldToScreenY(y2)));
```

È un metodo sicuro e trasparente, e per un grafico basta. Diventa scomodo quando le forme sono molte o complicate: ogni coordinata di ogni `Path2D` va convertita a mano, le dimensioni (il raggio di un cerchio, la larghezza di un rettangolo) vanno scalate a parte, e una rotazione richiede la trigonometria su ogni vertice. Per questi casi c'è un'altra strada.

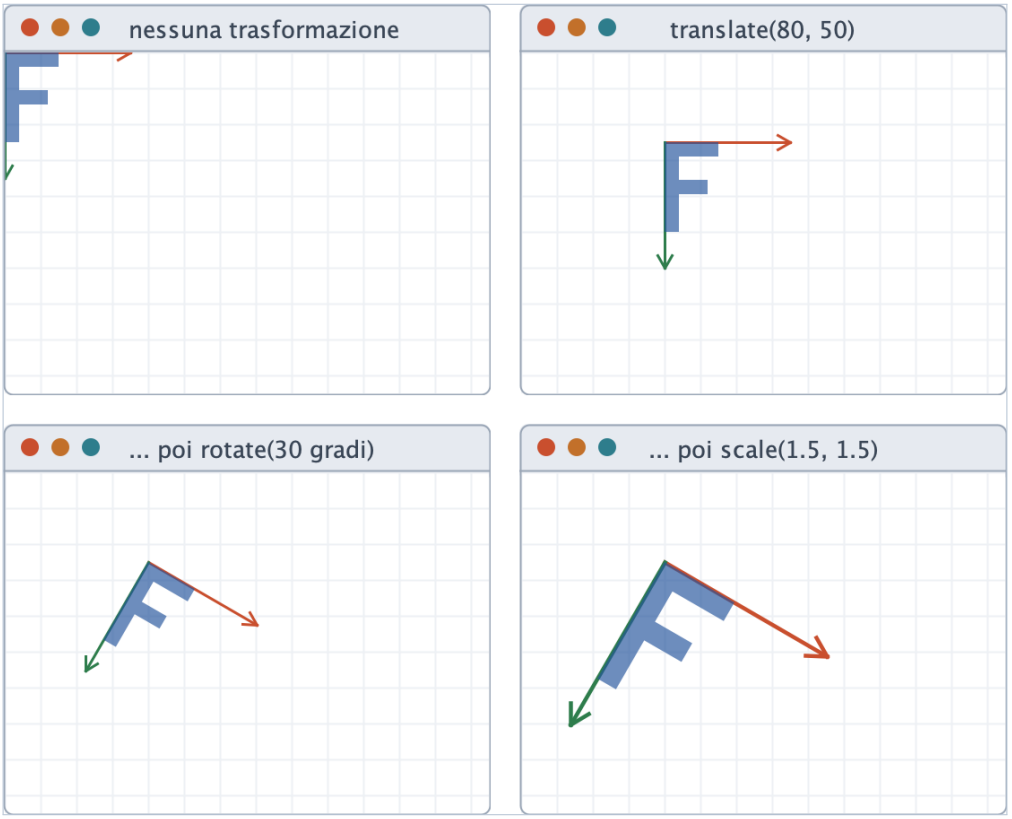
7.2 Trasformare il sistema

`Graphics2D` conserva, nel suo stato, una *trasformazione*: la regola con cui le coordinate passate ai metodi di disegno diventano pixel. All'inizio è l'identità, e $(10, 20)$ è il pixel $(10, 20)$. Tre metodi la cambiano, e vanno capiti con precisione: *non spostano le figure, spostano il sistema di riferimento*. Tutto ciò che si disegna dopo la chiamata usa il sistema nuovo.

Definizione 7.1: Trasformazioni del contesto

`translate(dx, dy)` sposta l'origine del sistema corrente nel punto (d_x, d_y) del sistema corrente. `scale(sx, sy)` cambia l'unità di misura: una unità lungo x diventa s_x unità del sistema precedente, e un fattore negativo capovolge l'asse. `rotate(theta)` ruota gli assi attorno all'origine corrente di `theta` *radianti*; con l'asse y verso il basso un angolo positivo ruota in senso *orario*. Le trasformazioni si *compongono*: ciascuna agisce sul sistema lasciato dalle precedenti.

Le quattro finestre della schermata disegnano la stessa "F", con le stesse coordinate (un'asta da $(0, 0)$ a $(0, 50)$ con due bracci verso destra), dopo trasformazioni diverse. Le due frecce sono gli assi del sistema corrente, x in rosso e y in verde; la griglia di sfondo è quella dei pixel, ogni 20.

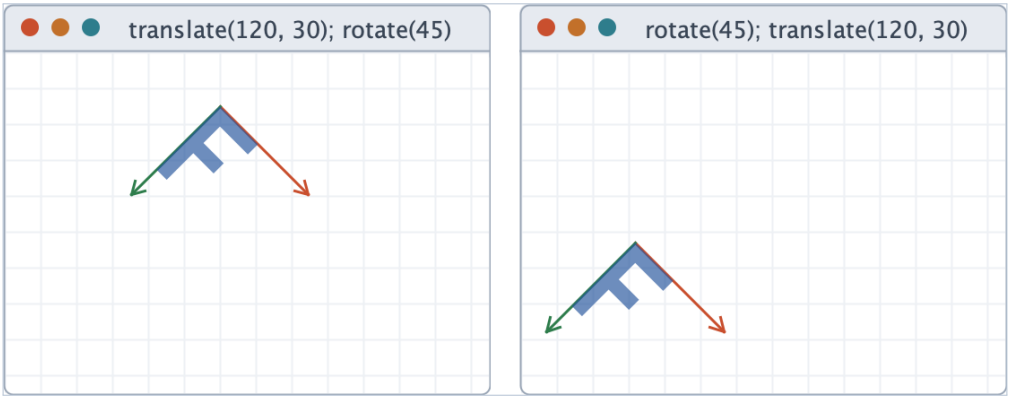


La stessa "F" disegnata dopo trasformazioni crescenti: ogni chiamata parte dal sistema lasciato dalla precedente.

Senza trasformazioni la F sta nell'angolo. `translate(80, 50)` porta l'origine in $(80, 50)$, e la F con lei. `rotate` aggiunto dopo ruota gli assi di 30 gradi attorno alla *nuova* origine, e la F si inclina. `scale(1.5, 1.5)` aggiunto dopo ancora allunga le unità degli assi ruotati: la F è più grande e resta inclinata. Nessuna coordinata della F è cambiata.

7.2.1 L'ordine conta

Che ogni trasformazione agisca sul sistema lasciato dalle precedenti ha una conseguenza che è il punto più importante del capitolo: l'ordine conta. Traslare e poi ruotare non è la stessa cosa che ruotare e poi traslare.



Le stesse due trasformazioni in ordine diverso: a sinistra la F ruota attorno al punto in cui è stata portata, a destra viene portata lungo gli assi già ruotati.

A sinistra la traslazione porta l'origine in $(120, 30)$ e la rotazione fa girare la F attorno a quel punto. A destra la rotazione avviene per prima, attorno all'angolo del pannello, e la

traslazione di $(120, 30)$ si fa lungo gli assi *già ruotati*: 120 unità nella direzione della freccia rossa inclinata e 30 in quella della verde. La F finisce più in basso e più a sinistra. Il modo più sicuro di prevedere il risultato è proprio questo: seguire le chiamate nell'ordine, disegnando a ogni passo i due assi del sistema corrente.

Ricetta: Prevedere un disegno con le trasformazioni

1. Disegnare gli assi del sistema iniziale: origine in alto a sinistra, x a destra, y in basso.
2. Per ogni `translate(dx, dy)`: spostare l'origine di d_x lungo l'asse x corrente e di d_y lungo l'asse y corrente.
3. Per ogni `rotate(a)`: ruotare entrambi gli assi attorno all'origine corrente, in senso orario se $a > 0$.
4. Per ogni `scale(sx, sy)`: annotare la nuova lunghezza dell'unità su ciascun asse, e capovolgere l'asse se il fattore è negativo.
5. Disegnare la figura nel sistema finale, con le sue coordinate.

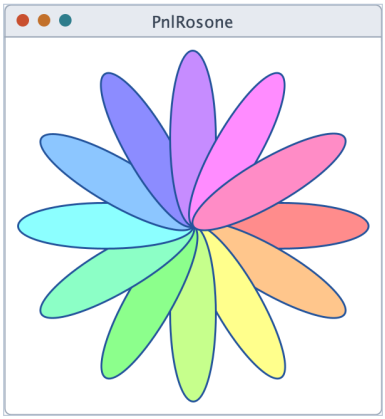
Osservazione 7.1

Dietro i tre metodi c'è un solo oggetto, `java.awt.geom.AffineTransform`: una matrice che rappresenta qualunque combinazione di traslazioni, rotazioni, scale e inclinazioni. `g2.getTransform()` restituisce quella corrente, `g2.transform(t)` la compone con un'altra, e un `AffineTransform` si può usare anche fuori dal disegno: trasforma punti (`transform`), forme (`createTransformedShape`) e sa calcolare la propria inversa (`inverseTransform`), che riporta i pixel nel mondo.

7.3 Disegnare attorno all'origine

Le trasformazioni cambiano il modo di costruire le forme. Invece di calcolare dove va ogni vertice, si disegna la forma nella posizione più comoda, quasi sempre centrata sull'origine, e la si porta al suo posto trasformando il sistema. Un rosone di dodici petali è un solo petalo, un'ellisse che parte dall'origine verso destra, disegnato dodici volte ruotando il sistema di un dodicesimo di giro:

```
g2.translate(getWidth() / 2.0, getHeight() / 2.0);         // l'origine al centro
Ellipse2D petalo = new Ellipse2D.Double(0, -r * 0.13, r, r * 0.26);
for (int i = 0; i < N; i++) {
    g2.setColor(Color.getHSBColor(i / (float) N, 0.45f, 1f));
    g2.fill(petalo);
    g2.rotate(2 * Math.PI / N);                            // il prossimo petalo
}
```



Un petalo, dodici rotazioni.

Il petalo non sa niente di angoli: è sempre lo stesso oggetto, e la trigonometria la fa il contesto. Le rotazioni si accumulano, ed è esattamente ciò che si vuole qui: dopo i giri il sistema è ruotato di i/N di giro.

Altre volte l'accumulo è un problema. Un pannello che disegna più oggetti, ciascuno con la sua posizione e la sua rotazione, deve tornare al sistema di partenza fra un oggetto e l'altro. Il modo più semplice è il contesto copiato del capitolo 6: una copia per ogni oggetto, trasformata a piacere e buttata alla fine.

```
for (Pianeta p : pianeti) {
    Graphics2D gp = (Graphics2D) g2.create();   // parte dal sistema di g2
    gp.rotate(p.angolo());                      // l'orbita
    gp.translate(p.distanza(), 0);              // la posizione sull'orbita
    double r = p.raggio();
    gp.fill(new Ellipse2D.Double(-r, -r, 2 * r, 2 * r)); // centrato sull'origine
    gp.dispose();                               // g2 e' com'era
}
```

L'alternativa è salvare la trasformazione prima e rimetterla dopo:

```
AffineTransform salvata = g2.getTransform();
// ... trasformazioni e disegno ...
g2.setTransform(salvata);
```

Funziona allo stesso modo per la trasformazione, ma ripristina solo quella; la copia ripristina tutto lo stato, colore e tratto compresi.

Attenzione 7.1: `setTransform` con una trasformazione nuova

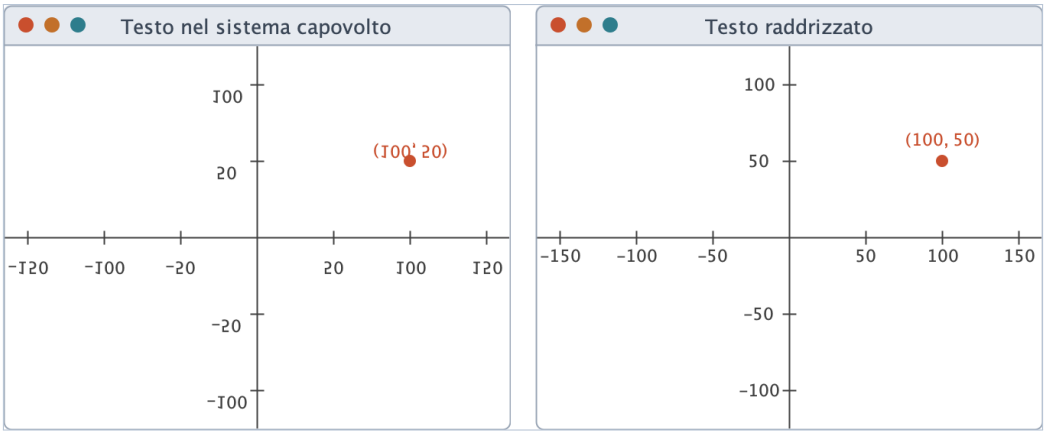
`g2.setTransform(new AffineTransform())` non riporta il sistema "all'inizio del componente": lo riporta all'identità della *superficie* su cui Swing sta disegnando, che può essere la finestra intera, con l'origine nell'angolo della finestra e non del pannello. Il disegno finisce nel posto sbagliato, o fuori dal pannello. Si ripristina solo una trasformazione salvata prima con `getTransform()`, mai una costruita da zero.

7.4 Il piano cartesiano

Il caso più frequente di cambio di sistema è il piano cartesiano: origine al centro del pannello e asse y verso l'alto. Con le trasformazioni sono due righe, una traslazione e un cambio di scala con il fattore verticale negativo, che capovolge l'asse:

```
g2.translate(getWidth() / 2, getHeight() / 2);  // l'origine al centro
g2.scale(1, -1);                                // e la y verso l'alto
g2.drawLine(-getWidth() / 2, 0, getWidth() / 2, 0);    // asse x
g2.drawLine(0, -getHeight() / 2, 0, getHeight() / 2);  // asse y
for (int x = 50; x < getWidth() / 2; x += 50) {        // tacche ogni 50 pixel
    g2.drawLine(x, -4, x, 4);
    g2.drawLine(-x, -4, -x, 4);
}
```

Gli assi e le tacche escono come in un quaderno di matematica, e un punto disegnato in $(100, 50)$ sta in alto a destra. Il guaio si vede appena si aggiungono le etichette: anche il testo viene capovolto, e i numeri si leggono allo specchio.



Gli stessi assi: a sinistra le etichette scritte nel sistema capovolto, a destra raddrizzate.

Il testo è un disegno come gli altri, e subisce la trasformazione come gli altri. Per raddrizzarlo si applica, solo per il testo, la trasformazione opposta: si porta l'origine nel punto dell'etichetta e si capovolge di nuovo l'asse, su una copia del contesto.

```
private void etichetta(Graphics2D g2, String s, int x, int y) {
    Graphics2D t = (Graphics2D) g2.create();
    t.translate(x, y);                  // l'origine sull'etichetta
    t.scale(1, -1);                     // si raddrizza il testo
    t.drawString(s, -t.getFontMetrics().stringWidth(s) / 2, 0);
    t.dispose();
}
```

C'è un secondo effetto collaterale, meno visibile, del cambio di scala: *scala anche il tratto*. Con `scale(20, 20)`, comodo per un mondo misurato in unità piccole, una linea di spessore 1 diventa larga 20 pixel. Ci sono due rimedi. Il primo è compensare, impostando un tratto di spessore $1/20$, che dopo la scala torna a un pixel; funziona finché la scala è uguale sui due assi. Il secondo, più pulito, è non scalare il contesto ma le forme: si costruisce la trasformazione dal mondo ai pixel come oggetto, si trasforma la forma, e la si disegna nel sistema dei pixel, con il tratto normale.

```
AffineTransform mondo = new AffineTransform();
mondo.translate(getWidth() / 2.0, getHeight() / 2.0);
mondo.scale(20, -20);                        // 20 pixel per unita', y in su
Shape curva = mondo.createTransformedShape(curvaNelMondo);
g2.setStroke(new BasicStroke(2));            // 2 pixel, a ogni scala
g2.draw(curva);
```

Attenzione 7.2: Le divisioni intere

`getWidth() / 2` è una divisione intera: con una larghezza dispari l'origine cade mezzo pixel a sinistra del centro, e in un disegno simmetrico la differenza si vede. Per le trasformazioni si usa `getWidth() / 2.0`, che è un `double`; lo stesso vale per i rapporti di scala, dove `lato / 100` con due interi può valere zero.

7.5 Il grafico di una funzione

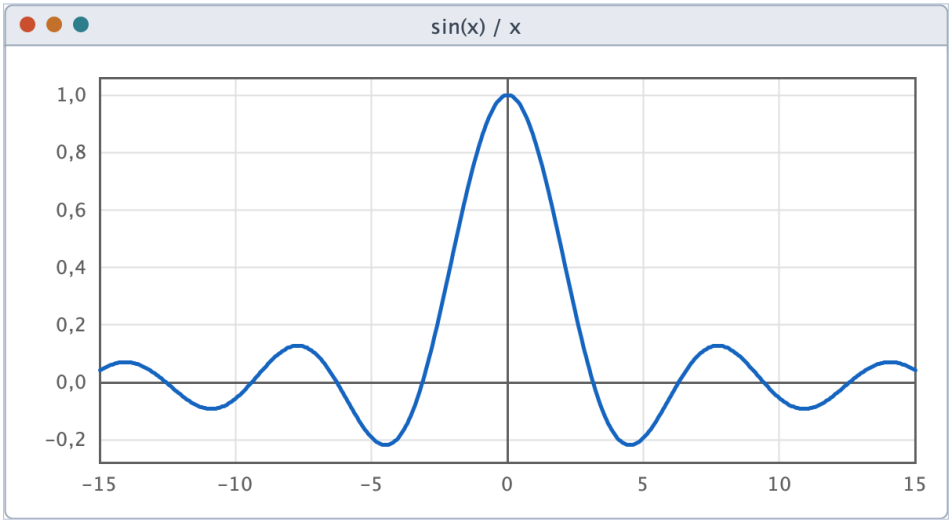
Tutto ciò che serve per disegnare il grafico di una funzione è ora a disposizione. La funzione arriva come `DoubleUnaryOperator`, l'interfaccia funzionale da `double` a `double` (*Dal nome alla funzione*), così che il pannello possa disegnare qualunque funzione gli si passi con una lambda: `pnlPlot.setFunzione(x -> Math.sin(x) / x)`.

Il disegno procede in tre passi. Il primo *campiona* la funzione: un valore per ogni colonna di pixel, perché più fitto non si vedrebbe e meno fitto perderebbe dettagli. Dai campioni si ricava l'intervallo verticale, il minimo e il massimo, con un piccolo margine, così che la curva occupi tutto il pannello qualunque sia la funzione.

```
double[] ys = new double[w + 1];
double min = Double.POSITIVE_INFINITY, max = Double.NEGATIVE_INFINITY;
for (int i = 0; i <= w; i++) {
    ys[i] = funzione.applyAsDouble(screenToWorldX(SINISTRA + i));
    if (Double.isFinite(ys[i])) {
        min = Math.min(min, ys[i]);
        max = Math.max(max, ys[i]);
    }
}
```

Il secondo passo disegna la griglia e gli assi, con le righe su valori "tondi" (multipli di 1, 2 o 5 per una potenza di dieci, scelti in modo da averne una decina) e le etichette calcolate con le funzioni di conversione. Il terzo disegna la curva come un `Path2D` che unisce i campioni. Alcuni campioni possono non essere numeri: $\sin(0)/0$ è `NaN`, $\tan(\pi/2)$ è enorme o infinito. Un punto non finito *interrompe* la curva, invece di sporcarla con un segmento verso l'infinito:

```
Path2D curva = new Path2D.Double();
boolean pennaGiu = false;
for (int i = 0; i <= w; i++) {
    if (!Double.isFinite(ys[i])) {
        pennaGiu = false;                       // si stacca la penna
        continue;
    }
    double x = SINISTRA + i, y = worldToScreenY(ys[i]);
    if (pennaGiu)
        curva.lineTo(x, y);
    else
        curva.moveTo(x, y);
    pennaGiu = true;
}
```



Il grafico di $\sin(x)/x$: in $x = 0$ il campione è `NaN`, e la curva si interrompe per un pixel.

Il grafico usa le funzioni di conversione e non le trasformazioni, ed è una scelta: le etichette, le linee della griglia e il tratto della curva devono restare in pixel, e l'unico calcolo nel mondo è il campionamento. Le due strade non si escludono, e un programma sceglie per ogni pezzo quella che lo rende più semplice.

7.6 L'esempio guida: la Lavagna con lo zoom

Fin qui la Lavagna ha registrato i punti dei gesti in pixel del pannello. Funziona finché la vista non cambia; ma se si vuole ingrandire il disegno, i pixel non sono più la misura giusta: la stessa figura occupa pixel diversi a zoom diversi. Le figure devono quindi stare in un sistema loro, le *coordinate del disegno*, con l'origine al centro dell'area; la *vista* è la trasformazione che le porta sullo schermo, e lo zoom cambia la vista, non il disegno.

```
/** Dal disegno allo schermo: l'origine al centro dell'area, poi lo zoom. */
public AffineTransform worldToScreen() {
    AffineTransform t = new AffineTransform();
    t.translate(getWidth() / 2.0, getHeight() / 2.0);
    t.scale(zoom, zoom);
    return t;
}

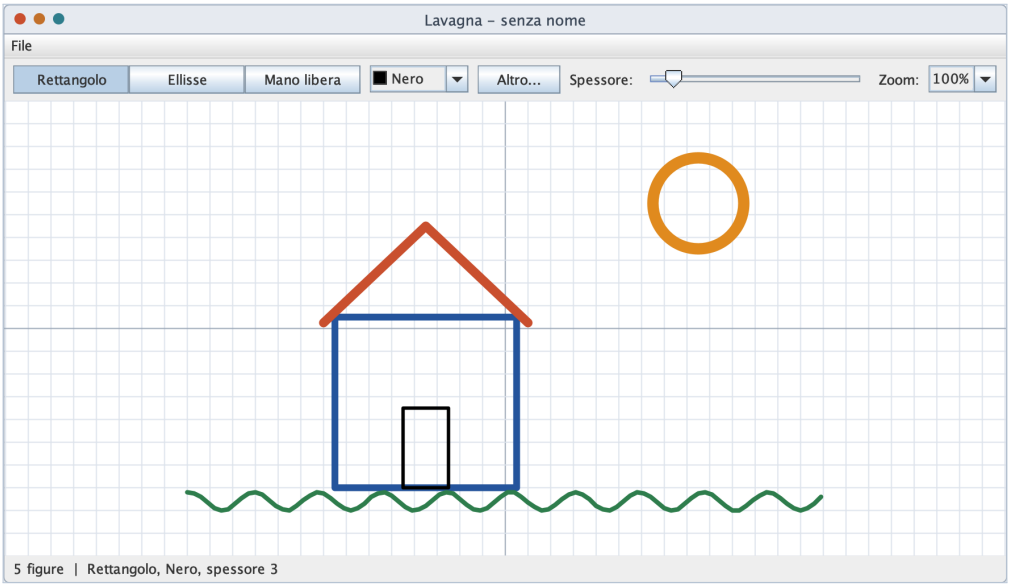
/** Dallo schermo al disegno: la trasformazione inversa. */
public Point2D screenToWorld(Point p) {
    try {
        return worldToScreen().inverseTransform(p, null);
    } catch (NoninvertibleTransformException e) {
        throw new IllegalStateException(e);     // con zoom > 0 non succede
    }
}
```

Le due direzioni sono le due metà del lavoro. In `paintComponent`, dopo l'antialiasing, una sola riga, `g2.transform(worldToScreen())`, e tutto ciò che segue si disegna nelle coordinate del disegno: le figure non cambiano di una virgola. Nel listener del mouse ogni punto passa per `screenToWorld` prima di entrare nel gesto, così che anche le figure nuove nascano nel sistema giusto. I punti diventano `Point2D`, reali, perché a zoom 400% un pixel vale un quarto di unità.

La griglia ora appartiene al disegno: le sue righe stanno sui multipli del passo nelle coordinate del disegno, e quella sullo zero, più scura, segna gli assi. Per disegnare solo le righe visibili si riportano nel disegno i due angoli del pannello e si disegnano le righe fra l'uno e l'altro. Lo spessore delle righe merita un'attenzione: nel sistema ingrandito un tratto di 1 sarebbe largo quanto lo zoom, e la griglia diventerebbe pesante. Si compensa con un tratto di $1/\mathit{zoom}$:

```
private void disegnaGriglia(Graphics2D g2) {
    Point2D a = screenToWorld(new Point(0, 0));
    Point2D b = screenToWorld(new Point(getWidth(), getHeight()));
    g2.setStroke(new BasicStroke((float) (1 / zoom)));   // un pixel
    double x0 = Math.floor(a.getX() / passo) * passo;     // il primo multiplo
    for (double x = x0; x <= b.getX(); x += passo) {
        g2.setColor(x == 0 ? ASSE : RIGA);
        g2.draw(new Line2D.Double(x, a.getY(), x, b.getY()));
    }
    // ... e lo stesso per le righe orizzontali
}
```

Lo spessore delle figure, invece, *deve* crescere con lo zoom: è una proprietà del disegno, come la loro dimensione, e un tratto di 4 a zoom 200% è giusto che sia largo 8 pixel. Nella barra una tendina sceglie lo zoom fra 50%, 100%, 200% e 400%; il file registra i punti come numeri reali, e la sua intestazione passa a "LAVAGNA 3".



La Lavagna con la griglia e l'origine al centro: le righe più scure sono gli assi del disegno.

Esercizi

Esercizio 7.1: Che cosa si vede (⋆)

Per ciascun frammento, eseguito in un pannello di 200 per 200 pixel con il colore nero, dire dove compare la figura e con che dimensioni, in pixel del pannello.

```
// (a)
g2.translate(100, 100);
g2.fillOval(-10, -10, 20, 20);
// (b)
g2.scale(2, 2);
g2.translate(10, 10);
g2.fillRect(0, 0, 20, 10);
// (c)
g2.translate(10, 10);
g2.scale(2, 2);
g2.fillRect(0, 0, 20, 10);
```

Esercizio 7.2: Il verso della rotazione (⋆)

Dove compare il segmento `g2.drawLine(0, 0, 50, 0)` dopo ciascuna delle due sequenze? Perché i due risultati sono diversi?

```
// (a)
g2.translate(100, 100);
g2.rotate(Math.PI / 2);
// (b)
g2.translate(100, 100);
g2.scale(1, -1);
g2.rotate(Math.PI / 2);
```

Esercizio 7.3: La rotazione che si accumula (⋆)

Un pannello disegna tre quadrati, ciascuno ruotato di 20 gradi attorno al proprio centro, in tre posizioni diverse; per ognuno chiama `translate` al centro del quadrato, `rotate`, e poi disegna il quadrato centrato sull'origine. Il primo esce giusto, gli altri due no. Che cosa si vede, e come si corregge?

Esercizio 7.4: La griglia pesante (⋆)

Un grafico usa `scale(40, -40)` per misurare il mondo in unità e disegna la griglia con `g2.draw(new Line2D.Double(...))` e il tratto predefinito. Quanto sono larghe le righe? Indicare due modi di averle larghe un pixel.

Esercizio 7.5: Il click nel disegno (⋆⋆)

Nella Lavagna di questo capitolo, a zoom 200% in un'area di 800 per 600 pixel, l'utente fa click nel pixel $(500, 200)$. Quali sono le coordinate del punto nel disegno? E se il programma dimenticasse di convertire il punto con `screenToWorld` e lo usasse com'è, dove comparirebbe la figura?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 7.6: Gli assi con le tacche (⋆)

Un pannello che disegna gli assi cartesiani con l'origine al centro, una freccia in fondo a ciascun asse (verso destra e verso l'alto), una tacca ogni 50 pixel con il valore scritto accanto, dritto, e il nome degli assi "x" e "y" vicino alle frecce. Le tacche devono essere simmetriche e fermarsi prima delle frecce, qualunque sia la dimensione del pannello.

Soluzione: progetto `sol-d05-c07`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 7.3: Le tacche contate a mano

Un numero fisso di tacche ("ce ne sono 4 per lato") funziona per una sola dimensione del pannello. Il ciclo va da 50 fino a metà della larghezza meno lo spazio della freccia, a passi di 50, e disegna ogni tacca in $+x$ e in $-x$.

Esercizio 7.7: Il campo di forze (⋆⋆)

Un pannello che disegna, centrata, una figura fatta di un solo elemento ripetuto n volte attorno al centro: un triangolo isoscele con la base vicino al centro e il vertice verso l'esterno, colorato con una tinta che cambia con l'indice. Il triangolo si costruisce una sola volta, attorno all'origine; la ripetizione la fanno le rotazioni. Uno slider sceglie n da 3 a 60, e una casella alterna i triangoli fra due lunghezze.

Soluzione: progetto `sol-d05-c07`, package `it.unibs.pajc.redolfi.e07`.

Esercizio 7.8: Il sistema solare (⋆⋆)

Un pannello che disegna un sistema solare fermo: il Sole al centro, tre pianeti su orbite circolari (tracciate in grigio) a distanze e angoli diversi, e una luna attorno al terzo pianeta. Ogni corpo si disegna con una copia del contesto: rotazione dell'orbita, traslazione sulla distanza, e per la luna una seconda rotazione e traslazione a partire dal sistema del suo pianeta. Le distanze sono in unità del sistema solare, e una scala adatta il tutto al lato minore del pannello.

Soluzione: progetto `sol-d05-c07`, package `it.unibs.pajc.redolfi.e08`.

Attenzione 7.4: La luna che gira attorno al Sole

Se la luna si disegna partendo dal contesto del Sole, invece che da quello del suo pianeta, le sue rotazioni avvengono attorno al centro del pannello: la luna finisce su un'orbita sua attorno al Sole. Il contesto della luna nasce dalla copia del pianeta, quando quella ha già fatto le sue trasformazioni.

Esercizio 7.9: Il grafico con lo zoom (⋆⋆⋆)

Estendere il pannello del grafico perché disegni *due* funzioni, con colori diversi e una legenda in alto a destra (un trattino del colore e il nome), e una barra con i pulsanti "Zoom +", "Zoom −", "←" e "→", che restringono, allargano e spostano l'intervallo delle x attorno al suo centro. La scala verticale si adatta a entrambe le funzioni insieme.

Soluzione: progetto `sol-d05-c07`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 7.1. (a) Un cerchio di diametro 20 centrato in $(100, 100)$: il centro del pannello. (b) Dopo `scale(2, 2)` ogni unità vale 2 pixel, e `translate(10, 10)` sposta l'origine di 10 unità, cioè in $(20, 20)$; il rettangolo va da $(20, 20)$ e misura 40×20 pixel. (c) La traslazione avviene prima della scala, in pixel: l'origine è in $(10, 10)$, e il rettangolo, scalato, va da $(10, 10)$ e misura ancora 40×20. Le dimensioni sono uguali, la posizione no: in (b) anche la traslazione è stata scalata.

Risposta 7.2. Nel primo caso l'asse x ruota di 90 gradi in senso orario, cioè punta verso il basso: il segmento va da $(100, 100)$ a $(100, 150)$. Nel secondo l'asse y è stato capovolto prima della rotazione, e nel sistema capovolto un angolo positivo gira in senso antiorario, come in matematica: l'asse x punta verso l'alto e il segmento va da $(100, 100)$ a $(100, 50)$. Il verso di `rotate` non è "orario" in assoluto: è da x verso y, e dipende da dove punta l'asse y.

Risposta 7.3. Le trasformazioni si accumulano: il secondo `translate` parte dal sistema del primo quadrato, già traslato e ruotato, e la seconda posizione viene misurata lungo assi inclinati di 20 gradi; il terzo quadrato è ruotato di 60 gradi in totale ed è ancora più fuori posto. Si corregge disegnando ogni quadrato su una copia del contesto, `Graphics2D gq = (Graphics2D) g2.create()`, trasformata e poi chiusa con `dispose()`; oppure salvando la trasformazione prima del primo quadrato e ripristinandola dopo ciascuno.

Risposta 7.4. Il tratto predefinito è largo 1 unità, e con `scale(40, -40)` un'unità vale 40 pixel: le righe sono larghe 40 pixel, e la griglia diventa una scacchiera di bande. Primo modo: un tratto di spessore $1/40$, `new BasicStroke(1f / 40)`, che dopo la scala torna a un pixel. Secondo modo: non scalare il contesto, ma costruire un `AffineTransform` dal mondo ai pixel e disegnare le righe trasformate con `createTransformedShape`, con il tratto normale nel sistema dei pixel.

Risposta 7.5. La vista porta l'origine al centro, $(400, 300)$, e moltiplica per 2. L'inversa sottrae il centro e divide per 2: $((500 - 400)/2, (200 - 300)/2) = (50, -50)$. Senza conversione il punto $(500, 200)$ verrebbe preso come coordinate del disegno, e la vista lo porterebbe in $(400 + 2 \cdot 500, 300 + 2 \cdot 200) = (1400, 700)$: fuori dall'area. L'utente non vedrebbe nascere niente sotto il mouse.

Capitolo 8

Mouse, tastiera e tempo

Nei capitoli precedenti il mouse è servito alla Lavagna in tre eventi, presi così come venivano. Questo capitolo prende sul serio l'interazione con un pannello che si disegna da solo, dove non c'è nessun componente a tradurre i gesti dell'utente in eventi semantici: tutti gli eventi del mouse e che cosa portano con sé, la rotellina, la tastiera con il problema del fuoco, le scorciatoie fatte bene. Poi il tempo: un pannello che cambia anche quando l'utente non fa niente, con `javax.swing.Timer`, che è il modo di animare in Swing. Alla fine la Lavagna è completa: cancella con il tasto destro, annulla con Ctrl+Z, ingrandisce con la rotellina attorno al puntatore, mostra le coordinate e riproduce il disegno una figura alla volta.

8.1 Il mouse

Il mouse genera eventi di basso livello di tre famiglie, ciascuna con il suo listener. `MouseListener` riceve ciò che accade ai pulsanti e i passaggi del puntatore sul bordo del componente; `MouseMotionListener` il movimento; `MouseWheelListener` la rotellina.

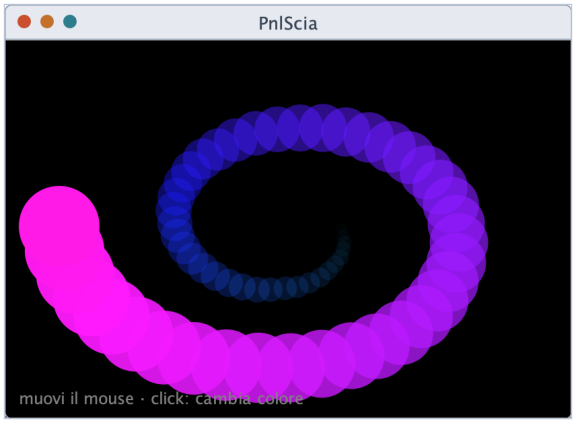
Listener	Metodo	Quando
`MouseListener`	`mousePressed`	un pulsante viene premuto
	`mouseReleased`	un pulsante viene rilasciato
	`mouseClicked`	premuto e rilasciato nello stesso punto
	`mouseEntered`, `mouseExited`	il puntatore entra, esce
`MouseMotionListener`	`mouseMoved`	il puntatore si muove, senza pulsanti
	`mouseDragged`	si muove con un pulsante premuto
`MouseWheelListener`	`mouseWheelMoved`	la rotellina gira

Tutti ricevono un `MouseEvent`, che dice dove (`getX()`, `getY()`, `getPoint()`, in pixel del componente), con quale pulsante (`SwingUtilities.isLeftMouseButton(e)`, `isRightMouseButton`, `isMiddleMouseButton`), quante volte di seguito (`getClickCount()`, 2 per un doppio click) e con quali tasti modificatori premuti (`isShiftDown()`, `isControlDown()`). La rotellina riceve un `MouseWheelEvent`, una sottoclasse con `getWheelRotation()`: negativo verso l'alto, positivo verso il basso.

Scrivere tre listener con dieci metodi per usarne tre sarebbe tedioso, e qui torna utile `MouseAdapter`: implementa tutte e tre le interfacce con metodi vuoti. Si estende con una classe anonima ridefinendo i metodi che servono, e lo stesso oggetto si registra con i tre metodi `add`. La scia del mouse, il primo esempio d'interazione, tiene in una coda gli ultimi sessanta punti toccati dal puntatore e li disegna come cerchi sempre più grandi e opachi:

```
MouseAdapter listener = new MouseAdapter() {
    @Override
    public void mouseMoved(MouseEvent e) {
        aggiungi(e.getPoint());
    }

    @Override
    public void mouseExited(MouseEvent e) {
        scia.clear();                       // fuori: niente scia
        repaint();
    }
};
addMouseListener(listener);                       // mouseExited sta qui...
addMouseMotionListener(listener);                 // ...mouseMoved qui
```



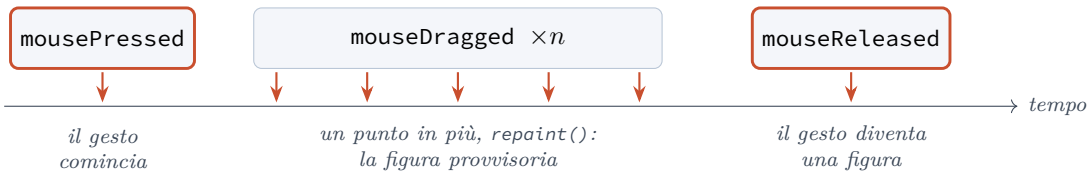
La scia del mouse: gli eventi aggiungono punti a una coda, `paintComponent` li disegna.

Attenzione 8.1: Il trascinamento sta in un altro listener

`mouseDragged` e `mouseMoved` sono metodi di `MouseMotionListener`: un `MouseAdapter` registrato solo con `addMouseListener` non li riceve mai, e il codice che li ridefinisce compila senza errori e non fa niente. Ogni metodo dell'adapter funziona solo se l'adapter è registrato presso il listener a cui il metodo appartiene; nel dubbio, si registra con tutti e tre. Non esiste, infine, un "MouseClickListener": il click è un metodo di `MouseListener`.

8.2 Un gesto in tre eventi

Tracciare una figura è un gesto che dura nel tempo, e il pannello lo riceve a pezzi: una pressione, una serie di trascinamenti, un rilascio. Fra un pezzo e l'altro il programma torna al ciclo degli eventi, e il gesto in corso deve stare nello stato del pannello, come nella Lavagna dal capitolo 5: la pressione inizia la lista dei punti, il trascinamento la allunga e chiede un ridisegno, il rilascio la trasforma in una figura.



Un gesto visto dal pannello: tre tipi di evento, e fra l'uno e l'altro lo stato del gesto resta nei campi.

Il trascinamento ha una proprietà comoda: una volta premuto il pulsante sul pannello, gli eventi continuano ad arrivare a quel pannello anche se il puntatore esce dai suoi bordi, fino al rilascio. Un gesto iniziato nell'area non si perde se l'utente esagera con il mouse; le coordinate, semplicemente, escono dall'area.

Il tasto destro, nella Lavagna, cancella la figura sotto il puntatore. È un'altra interpretazione della pressione, che si riconosce all'inizio di `mousePressed` e non avvia nessun gesto; il trascinamento successivo trova la lista dei punti vuota e non fa niente.

```
@Override
public void mousePressed(MouseEvent e) {
    if (SwingUtilities.isRightMouseButton(e)) {
        Figura f = figuraIn(e.getPoint());      // il tratto ingrossato
        if (f != null) {
            figure.remove(f);
            fireStateChanged();
            repaint();
        }
        return;                                 // nessun gesto
    }
    gesto.clear();
    gesto.add(screenToWorld(e.getPoint()));
}
```

Osservazione 8.1

`mouseClicked` sembra il posto naturale per "fare qualcosa con un click", ma arriva solo se pressione e rilascio avvengono nello stesso punto: un utente che muove il mouse di un pixel mentre clicca non genera il click. Per le azioni che devono essere sicure (una selezione, una cancellazione) `mousePressed` è più affidabile; `mouseClicked` serve quando si vuole distinguere un click da un trascinamento, o per il doppio click con `getClickCount()`.

8.3 La rotellina e le proprietà che avvisano

Nella Lavagna del capitolo 7 lo zoom ingrandiva attorno al centro dell'area. Con la rotellina ci si aspetta un'altra cosa: che ingrandisca attorno al puntatore, così che il punto sotto il mouse resti dov'è e il resto si allarghi attorno. Serve che la vista, oltre allo zoom, abbia uno spostamento in pixel, `panX` e `panY`, che si aggiunge alla traslazione verso il centro dell'area.

Il conto è una condizione da imporre. Sia q il pixel del puntatore e p il punto del disegno sotto di lui, calcolato con lo zoom vecchio. Dopo il cambio di zoom la vista deve portare p ancora in q: $w/2 + pan_x + z \cdot p_x = q_x$, da cui si ricava pan_x, e lo stesso per y.

```
public void zoomAttorno(double nuovo, Point fulcro) {
    Point2D p = screenToWorld(fulcro);                // con lo zoom vecchio
    double vecchio = zoom;
    zoom = nuovo;
    panX = fulcro.x - getWidth() / 2.0 - zoom * p.getX();  // vista(p) = fulcro
    panY = fulcro.y - getHeight() / 2.0 - zoom * p.getY();
    firePropertyChange("zoom", vecchio, zoom);
    repaint();
}
```

Il listener della rotellina raddoppia o dimezza lo zoom, entro un minimo e un massimo, e chiama `zoomAttorno` con il punto dell'evento. La tendina dello zoom chiama lo stesso metodo con il centro dell'area.

La penultima riga merita un discorso a parte. Ora lo zoom cambia in due modi, dalla tendina e dalla rotellina, e la tendina deve mostrare il valore giusto anche quando lo cambia la rotellina: il pannello deve avvisare. Si potrebbe aggiungere un altro tipo di evento, come il `ChangeEvent` delle figure; ma ogni `JComponent` ha già un meccanismo pronto per questo, le *proprietà*. `firePropertyChange(nome, vecchio, nuovo)` avvisa tutti i listener registrati per quel nome con `addPropertyChangeListener(nome, l)`, e non fa niente se il valore vecchio e il nuovo sono uguali. La Lavagna ascolta due proprietà del pannello:

```
pnlLavagna.addPropertyChangeListener("zoom", e ->
        cmbZoom.setSelectedItem(Math.round(pnlLavagna.getZoom() * 100) + "%"));
pnlLavagna.addPropertyChangeListener("puntatore", e -> aggiornaStato());
```

La seconda proprietà, `puntatore`, è il punto del disegno sotto il mouse, che il pannello aggiorna in `mouseMoved` e `mouseDragged` e azzera in `mouseExited`; la riga di stato lo mostra come coordinate. Che la tendina, quando la rotellina la aggiorna, generi a sua volta un evento e richiami `setZoom` con lo stesso valore non è un problema: il valore non cambia, e `firePropertyChange` non avvisa nessuno. È il controllo "nessun cambiamento, nessun evento" del capitolo 3, che qui la libreria fa da sé.

8.4 La tastiera

La tastiera ha un problema che il mouse non ha: a chi vanno i tasti? Il mouse indica da solo il componente sotto il puntatore; i tasti vanno al componente che ha il *fuoco* (*focus*), uno solo per finestra, quello in cui sta il cursore o che ha il bordo evidenziato. Un pannello che si disegna non riceve il fuoco, se non lo si chiede: va dichiarato con `setFocusable(true)`, e il fuoco va preso con `requestFocusInWindow()`, tipicamente quando l'utente ci fa click sopra.

Definizione 8.1: Eventi della tastiera

Un `KeyListener` riceve tre eventi dal componente che ha il fuoco. `keyPressed` e `keyReleased` parlano di *tasti*: `e.getKeyCode()` restituisce un codice `KeyEvent.VK_...` (`VK_LEFT`, `VK_SPACE`, `VK_A`) che identifica il tasto fisico. `keyTyped` parla di *caratteri*: `e.getKeyChar()` restituisce il carattere prodotto, che dipende dalla tastiera, dalle maiuscole e dalla lingua.

Per un gioco o un pannello che si muove con le frecce servono `keyPressed` e `keyReleased`; per accettare del testo, `keyTyped`, perché il "+" sta su tasti diversi in tastiere diverse. Tenendo

premuto un tasto, il sistema ripete `keyPressed` a intervalli regolari, dopo una pausa iniziale: è la *ripetizione automatica*, utile per scrivere e scomoda per muovere qualcosa, perché il movimento parte, si ferma mezzo secondo e poi riparte a scatti. Inoltre la ripetizione riguarda solo l'ultimo tasto premuto: con due frecce premute insieme, se ne ripete una sola.

Il rimedio è separare il *sapere* quali tasti sono premuti dal *muovere*. Gli eventi della tastiera tengono aggiornato l'insieme dei tasti premuti in questo momento; un timer, che vedremo fra poco, a intervalli regolari legge l'insieme e sposta l'oggetto.

```
private final Set<Integer> premuti = new HashSet<>();

addKeyListener(new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
        premuti.add(e.getKeyCode());            // ripetuto? add non cambia niente
    }

    @Override
    public void keyReleased(KeyEvent e) {
        premuti.remove(e.getKeyCode());
    }
});
```

Con due frecce premute l'oggetto si muove in diagonale, e il movimento è continuo, perché dipende dal timer e non dalla ripetizione.

Attenzione 8.2: Il tasto che resta premuto

Se il componente perde il fuoco mentre un tasto è premuto (l'utente passa a un'altra finestra), il `keyReleased` di quel tasto non arriva mai, e per il programma il tasto resta premuto per sempre: l'oggetto continua a muoversi da solo. Un `FocusListener` che svuota l'insieme in `focusLost` risolve il problema.

8.5 Le scorciatoie

Ctrl+Z per annullare non è un tasto del pannello: deve funzionare ovunque sia il fuoco nella finestra, anche se l'utente ha appena toccato lo slider. Un `KeyListener` sul pannello non basta, perché riceverebbe il tasto solo quando ha il fuoco. Swing offre due meccanismi migliori.

Il primo, per le azioni che stanno in un menu, è l'*acceleratore* della voce: la voce di menu ascolta la combinazione in tutta la finestra e, quando arriva, si attiva come se fosse stata scelta con il mouse. Il menu mostra la combinazione accanto alla voce, e l'utente la impara da lì.

```
int menu = Toolkit.getDefaultToolkit().getMenuShortcutKeyMaskEx();  // Ctrl o Cmd
JMenuItem annulla = voce("Annulla", pnlLavagna::annullaUltima);
annulla.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Z, menu));
```

`getMenuShortcutKeyMaskEx()` restituisce il modificatore delle scorciatoie del sistema: Ctrl su Windows e Linux, Cmd su macOS. Scrivere `CTRL_DOWN_MASK` darebbe agli utenti di macOS una scorciatoia diversa da quella di tutti gli altri programmi.

Il secondo meccanismo, per i tasti che non corrispondono a una voce di menu, sono le mappe di un componente: l'`InputMap` associa un tasto a un nome, l'`ActionMap` associa il nome

a un'azione. Con la condizione `WHEN_IN_FOCUSED_WINDOW` il tasto funziona finché la finestra ha il fuoco, ovunque sia. Nella Lavagna Esc annulla il gesto in corso:

```
KeyStroke esc = KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0);
getInputMap(WHEN_IN_FOCUSED_WINDOW).put(esc, "annullaGesto");
getActionMap().put("annullaGesto", new AbstractAction() {
    @Override
    public void actionPerformed(ActionEvent e) {
        gesto.clear();
        repaint();
    }
});
```

È lo stesso meccanismo che si è usato nel capitolo 4 per chiudere una finestra di dialogo con Esc, attraverso `registerKeyboardAction`, che è una scorciatoia per queste due righe. Un'`Action`, di cui `AbstractAction` è l'implementazione di comodo, è un `ActionListener` con in più un nome, un'icona e uno stato abilitato: la stessa azione si può assegnare a un tasto, a una voce di menu e a un pulsante, e disabilitarla disabilita tutti e tre.

8.6 Il tempo: `javax.swing.Timer`

Tutto ciò che si è visto finora accade quando l'utente fa qualcosa. Un'animazione, un orologio, un gioco cambiano anche quando l'utente sta fermo: hanno bisogno che qualcuno li avvisi a intervalli regolari. Il modo di farlo in Swing è `javax.swing.Timer`.

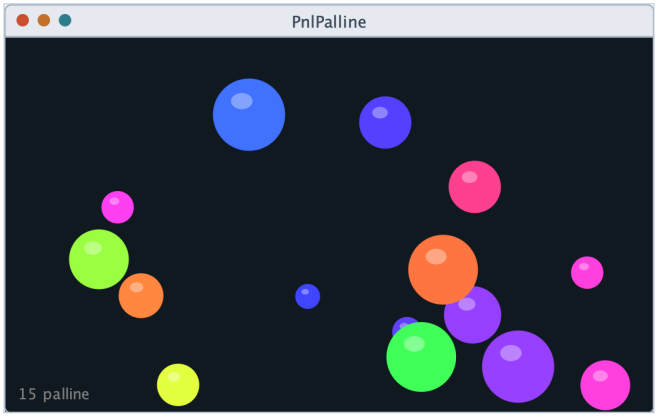
Definizione 8.2: `javax.swing.Timer`

Un `javax.swing.Timer` chiama un `ActionListener` ogni tot millisecondi, sul thread degli eventi di Swing, come qualunque altro evento. Si costruisce con l'intervallo e il listener, `new Timer(16, e -> stepNext())`, e si comanda con `start()`, `stop()` e `isRunning()`; `setRepeats(false)` lo fa scattare una volta sola.

Che il listener giri sul thread degli eventi è il punto decisivo. Il passo dell'animazione può leggere e scrivere lo stato del pannello e chiamare `repaint()` senza nessuna sincronizzazione, perché il mouse, la tastiera, il disegno e il timer stanno tutti sullo stesso thread e non possono mai accavallarsi. Lo schema di un'animazione è sempre quello dell'immagine guida: il timer è un evento come gli altri, cambia lo stato e chiede un ridisegno. Il metodo che fa avanzare lo stato di un passo lo chiameremo sempre `stepNext`, in ogni programma in cui passa il tempo:

```
private final Timer timer = new Timer(16, e -> stepNext());   // circa 60 al
    secondo

private void stepNext() {
    for (Pallina p : palline) {
        p.x += p.vx;                              // si muove...
        p.y += p.vy;
        if (p.x - p.r < 0 || p.x + p.r > getWidth())
            p.vx = -p.vx;                         // ...e rimbalza sui bordi
        if (p.y - p.r < 0 || p.y + p.r > getHeight())
            p.vy = -p.vy;
    }
    repaint();
}
```



Palline che rimbalzano: ogni 16 ms il timer sposta lo stato e chiede un ridisegno.

Il passo deve essere breve, per la stessa ragione dei listener: finché `stepNext()` non torna, la finestra non si ridisegna e non risponde. Se dura più dell'intervallo, il timer non accumula i ritardi: fonde gli scatti arretrati in uno solo, e l'animazione rallenta invece di intasare la coda. Per le animazioni che devono avere una velocità reale, indipendente da quanto spesso il timer riesce a scattare, lo spostamento si calcola dal tempo trascorso dall'ultimo passo (`System.nanoTime()`) invece che con un passo fisso.

Attenzione 8.3: I timer sbagliati

`java.util.Timer` e uno `ScheduledExecutorService` chiamano il compito da un *altro* thread, che non può toccare i componenti né lo stato che `paintComponent` legge senza sincronizzazione (*Thread e concorrenza*). E un ciclo con `Thread.sleep(16)` dentro un listener blocca il thread degli eventi per tutta la durata dell'animazione: la finestra resta congelata e mostra solo l'ultimo fotogramma. Per animare un'interfaccia Swing si usa `javax.swing.Timer`, e il compilatore non aiuta: le tre classi si chiamano `Timer` e compilano tutte.

8.7 L'esempio guida: la Lavagna completa

La Lavagna riceve le ultime funzioni. Il tasto destro, la rotellina e le coordinate nella riga di stato si sono già visti. Il menu "Modifica" ha due voci con il loro acceleratore: "Annulla" toglie l'ultima figura, "Riproduci" ridisegna il disegno da capo, una figura ogni 400 millisecondi.

La riproduzione è un'animazione, e si costruisce come le altre: uno stato, un passo e un timer. Lo stato è il numero di figure da mostrare, `visibili`, che vale -1 quando si mostrano tutte; `paintComponent` disegna solo le prime `visibili`:

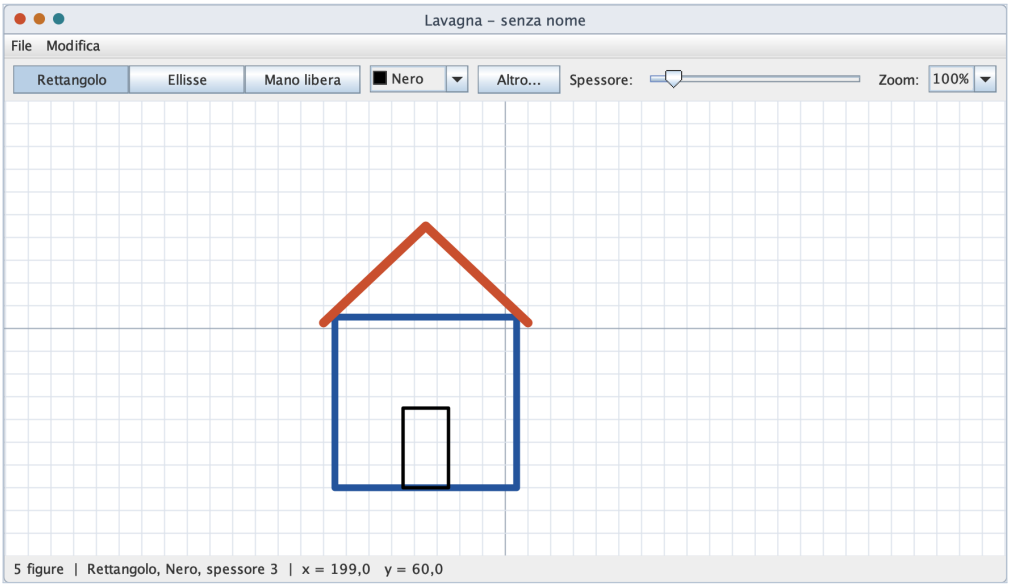
```
private int visibili = -1;                  // quante se ne vedono; -1 tutte
private final Timer timer = new Timer(400, e -> stepNext());

public void riproduci() {
    if (figure.isEmpty() || timer.isRunning())
        return;
    visibili = 0;
    timer.start();
    repaint();
}

private void stepNext() {
    visibili++;
    if (visibili >= figure.size()) {
        timer.stop();
        visibili = -1;                          // di nuovo tutte
    }
    repaint();
}
```

In `paintComponent` il ciclo sulle figure diventa `for (Figura f : figure.subList(0, n))`, con `n` uguale a `visibili` o, se è -1, al numero di figure. Durante la riproduzione il mouse non deve aggiungere niente, e `mousePressed` comincia con `if (timer.isRunning()) return;`; "Annulla" controlla lo stesso.

"Annulla" toglie l'ultima figura della lista e avvisa, come la cancellazione con il tasto destro. È un annullamento semplice: annulla solo le aggiunte, e una figura cancellata con il tasto destro non torna indietro. Un annullamento completo tiene una pila delle operazioni fatte, ciascuna capace di disfarsi; Swing ha un'infrastruttura apposta, `javax.swing.undo`, e la palestra ne propone una versione fatta a mano.



La Lavagna durante la riproduzione: tre figure su cinque, e nella riga di stato le coordinate del puntatore.

Ricetta: Animare un pannello

1. Lo stato dell'animazione in campi del pannello (posizioni, velocità, un contatore), e `paintComponent` che lo disegna.
2. Un metodo `stepNext()` che fa avanzare lo stato di un intervallo e chiama `repaint()`; breve, senza attese.
3. Un `javax.swing.Timer` con l'intervallo e il passo: `new Timer(16, e -> stepNext())`.
4. `start()` e `stop()` dai comandi dell'utente o dal passo stesso, quando l'animazione finisce.

Esercizi

Esercizio 8.1: La sequenza degli eventi (⋆)

Un pannello ha un `MouseAdapter` registrato con `addMouseListener` e `addMouseMotionListener`, che stampa il nome di ogni metodo chiamato. Che cosa stampa (a) per un click fermo; (b) per una pressione, uno spostamento di 30 pixel e un rilascio; (c) per un doppio click? E se l'adapter fosse registrato solo con `addMouseListener`?

Esercizio 8.2: I tasti che non arrivano (⋆)

Un pannello che si disegna ha un `KeyListener` che sposta un cerchio con le frecce, ma premendo le frecce non succede niente. Elencare le cause possibili e, per ciascuna, il rimedio.

Esercizio 8.3: Il passo troppo lungo (⋆)

Un `javax.swing.Timer` ha un intervallo di 100 ms, e il suo passo impiega 300 ms. Quante volte al secondo viene eseguito il passo? Che cosa vede l'utente se prova a ridimensionare la finestra o a premere un pulsante? Che cosa cambierebbe con un `java.util.Timer`?

Esercizio 8.4: L'animazione con `sleep` (⋆⋆)

Il listener di un pulsante "Anima" esegue un ciclo di 100 iterazioni: sposta una pallina di 3 pixel, chiama `repaint()` e poi `Thread.sleep(20)`. Che cosa vede l'utente durante i due secondi del ciclo, e alla fine? Perché i cento `repaint()` non producono cento ridisegni?

Esercizio 8.5: Lo zoom attorno al puntatore (⋆⋆)

In un'area di 800 per 600 pixel, con zoom 1 e nessuno spostamento ($pan = 0$), l'utente gira la rotellina con il puntatore nel pixel $(600, 300)$, e lo zoom passa a 2. Quale punto del disegno sta sotto il puntatore prima? Quanto valgono pan_x e pan_y dopo? Dove finisce sullo schermo l'origine del disegno?

I problemi che seguono chiedono un programma completo, in ordine crescente di difficoltà. Le soluzioni sono nel repository delle soluzioni del corso, nel progetto e nel package indicati.

Esercizio 8.6: Il cerchio che segue (⋆)

Un pannello con un cerchio che segue il puntatore del mouse, centrato su di lui; un click cambia il colore del cerchio (passando alla tinta successiva su dodici), la rotellina ne cambia il raggio fra 5 e 100 pixel, e quando il puntatore esce dal pannello il cerchio sparisce. Nella riga di stato le coordinate del puntatore e il raggio.

Soluzione: progetto `sol-d05-c08`, package `it.unibs.pajc.redolfi.e06`.

Esercizio 8.7: Spostare le figure (⋆⋆)

Un pannello con una decina di forme piene (rettangoli ed ellissi di colori diversi, in posizioni casuali). Premendo il tasto sinistro su una forma la si afferra (la più in alto, se si sovrappongono) e la si porta in primo piano; trascinando la si sposta, tenendo fermo il punto in cui era stata presa; il rilascio la lascia. La forma afferrata ha il contorno evidenziato. Le forme si spostano con un `AffineTransform` di traslazione applicato con `createTransformedShape`.

Soluzione: progetto `sol-d05-c08`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 8.4: La forma che salta

Se nel trascinamento la forma viene spostata in modo che il suo angolo coincida con il puntatore, al primo movimento "salta" sotto il mouse. Si salva lo scarto fra il punto di presa e la posizione della forma alla pressione, e a ogni trascinamento si sposta della differenza fra il punto nuovo e quello precedente.

Esercizio 8.8: Il gioco della vita (⋆⋆⋆)

Il *gioco della vita* di Conway su una griglia di 60 per 40 celle, toroidale (il bordo destro confina con il sinistro, quello inferiore con il superiore). Il pannello disegna la griglia, centrata e con le celle quadrate; un click accende o spegne una cella; un pulsante avvia e ferma un `javax.swing.Timer` che a ogni scatto calcola la generazione successiva; un altro svuota la griglia, un terzo la riempie a caso; uno slider regola la velocità. La regola di nascita e sopravvivenza è una lambda, così che se ne possano provare di diverse. *Suggerimento: la generazione nuova si calcola in una matrice nuova, leggendo solo la vecchia; aggiornare le celle sul posto fa contare i vicini già cambiati.*

Soluzione: progetto `sol-d05-c08`, package `it.unibs.pajc.redolfi.e08`.

Esercizio 8.9: Il planetario (⋆⋆⋆)

Il sistema solare del capitolo 7 in movimento: ogni pianeta ha una velocità angolare sua (i più lontani più lenti), la luna gira attorno al suo pianeta, e un `javax.swing.Timer` fa avanzare gli angoli. Lo spostamento si calcola dal tempo trascorso, così che la velocità non dipenda dalla frequenza del timer; la barra spaziatrice mette in pausa e riprende, "+" e "−" accelerano e rallentano il tempo.

Soluzione: progetto `sol-d05-c08`, package `it.unibs.pajc.redolfi.e09`.

Risposte

Risposta 8.1. (a) `mousePressed`, `mouseReleased`, `mouseClicked`, preceduti da `mouseMoved` (e `mouseEntered`) mentre il puntatore arriva sul pannello. (b) `mousePressed`, una serie di `mouseDragged`, `mouseReleased`, e niente `mouseClicked`, perché pressione e rilascio sono in punti diversi. (c) Due volte la sequenza di (a); nel secondo `mouseClicked` `getClickCount()` vale 2. Registrato solo con `addMouseListener`, l'adapter non riceve più `mouseMoved` e `mouseDragged`, che appartengono a `MouseMotionListener`: nel caso (b) si stamperebbero solo pressione e rilascio.

Risposta 8.2. Il pannello non può ricevere il fuoco: si chiama `setFocusable(true)`. Può riceverlo ma non l'ha: un altro componente della finestra, per esempio un pulsante, ha il fuoco; si chiama `requestFocusInWindow()` sul pannello quando compare e quando l'utente ci fa click. Il listener è registrato su un altro componente, per esempio sulla finestra invece che sul pannello. Il codice confronta `getKeyChar()` con le frecce, che non producono caratteri: le frecce si riconoscono in `keyPressed` con `getKeyCode()` e `VK_LEFT`. Per tasti che devono funzionare comunque, la soluzione robusta è un'`InputMap` con `WHEN_IN_FOCUSED_WINDOW`.

Risposta 8.3. Circa tre volte al secondo: il timer non può scattare mentre il passo precedente è in corso, perché tutto avviene sullo stesso thread, e fonde gli scatti arretrati invece di accumularli. Il thread degli eventi è occupato dal passo tre decimi di secondo ogni trecento millisecondi, cioè sempre: la finestra risponde a scatti, ridimensionarla è faticoso, un pulsante reagisce con ritardo. Con un `java.util.Timer` il passo girerebbe su un altro thread, e la finestra resterebbe reattiva; ma il passo non potrebbe più toccare lo stato letto da `paintComponent` senza sincronizzazione, né i componenti. La soluzione è rendere il passo breve, o spostare il calcolo pesante fuori dal thread degli eventi e riportarne il risultato (*Thread e concorrenza*).

Risposta 8.4. L'utente non vede nessun movimento: per due secondi la finestra è congelata, e alla fine la pallina compare direttamente nella posizione finale. Il ciclo gira sul thread degli eventi, lo stesso che dovrebbe ridisegnare; `repaint()` mette una richiesta nella coda, ma la coda non viene servita finché il listener non torna, e a quel punto le cento richieste sono fuse in un solo ridisegno, con lo stato finale. L'animazione va costruita con un `javax.swing.Timer`, un passo per scatto.

Risposta 8.5. Prima la vista è la traslazione di $(400, 300)$, e il punto del disegno sotto $(600, 300)$ è $(200, 0)$. Dopo: $pan_x = 600 - 400 - 2 \cdot 200 = -200$, $pan_y = 300 - 300 - 2 \cdot 0 = 0$. La vista porta l'origine del disegno in $(400 - 200, 300 + 0) = (200, 300)$: l'origine si è spostata a sinistra, e il punto $(200, 0)$ è ancora in $200 + 2 \cdot 200 = 600$, sotto il puntatore.

Capitolo 9

Palestra

Nessuna teoria nuova. Otto programmi in ordine crescente di difficoltà, ciascuno con l'indicazione di che cosa allena, che arrivano a chiudere il problema del capitolo 1. I primi due sono nello stile delle prove scritte: disegnare a mano la finestra prodotta da un frammento di codice, scrivere un `paintComponent` su carta. I tre seguenti sono piccoli programmi completi, ciascuno centrato su una parte della dispensa: componenti ed eventi, finestre di dialogo e immagini, trasformazioni. Gli ultimi tre sono applicazioni vere, da progettare prima di scrivere: un gioco da tavolo, un gioco in tempo reale e la Lavagna con le funzioni che le mancano. Dopo ogni enunciato, l'errore che si vede fare più spesso su quel tipo di problema; le soluzioni, con i commenti sulle scelte, sono nel repository delle soluzioni del corso, progetto `sol-d05-c09`. Il consiglio vale soprattutto qui: prima un disegno della finestra e dello stato, poi il codice.

Problemi

Esercizio 9.1: Le finestre da disegnare (⋆)

Allena: layout predefiniti, `BorderLayout` e `FlowLayout`, `pack`. Per ciascuno dei tre frammenti, che costruiscono il contenuto di una `JFrame` poi mostrata con `pack()`, disegnare a mano la finestra, indicando quali componenti si vedono e dove, e che cosa succede allargandola. Poi scrivere un programma che apre le tre finestre affiancate, per controllare.

```
// (a)
add(new JButton("Uno"));
add(new JButton("Due"), BorderLayout.NORTH);
add(new JButton("Tre"));
// (b)
JPanel p = new JPanel();
p.add(new JButton("Uno"));
p.add(new JButton("Due"));
add(p, BorderLayout.SOUTH);
add(new JTextArea(5, 20));
// (c)
JPanel p = new JPanel(new GridLayout(0, 2));
for (int i = 1; i <= 5; i++)
    p.add(new JButton("B" + i));
add(p, BorderLayout.WEST);
add(new JLabel("centro", SwingConstants.CENTER));
```

Soluzione: progetto `sol-d05-c09`, package `it.unibs.pajc.redolfi.e01`.

Attenzione 9.1: I default dei due contenitori

Il content pane di una `JFrame` ha un `BorderLayout`, un `JPanel` un `FlowLayout`: confonderli è l'errore più frequente in questo tipo di domanda. E in un `BorderLayout` ogni `add` senza regione va in `CENTER` e sostituisce il precedente: in (a) "Uno" non si vede.

Esercizio 9.2: Cerchio e assi (⋆)

Allena: `paintComponent`, misure calcolate, il piano cartesiano. Scrivere su carta, e poi in un programma, il `paintComponent` di un pannello che disegna gli assi cartesiani con l'origine al centro, con una tacca ogni 50 pixel su entrambi gli assi, e il cerchio pieno centrato nell'origine con il diametro pari al lato minore del pannello, in un colore semitrasparente perché gli assi si vedano attraverso. Il disegno deve restare corretto a ogni ridimensionamento.

Soluzione: progetto `sol-d05-c09`, package `it.unibs.pajc.redolfi.e02`.

Attenzione 9.2: Il cerchio descritto dal centro

`fillOval(cx, cy, d, d)` disegna un cerchio con l'*angolo* del quadrato contenitore in (c_x, c_y), non il centro: il cerchio esce spostato in basso a destra di un raggio. Il cerchio si descrive con il quadrato che lo contiene, da $(c_x - r, c_y - r)$; e il raggio è metà del lato *minore*, non della larghezza.

Esercizio 9.3: Il convertitore (⋆⋆)

Allena: oggetti nelle tendine, un listener per molti componenti, messaggi d'errore. Una finestra che converte lunghezze: un campo di testo per il valore, due tendine con le unità (metri, chilometri, centimetri, miglia, piedi, pollici), un pulsante "↔" che scambia le due unità e un'etichetta grande con il risultato, aggiornato a ogni cambiamento di valore o di unità. Le unità sono record con il nome e il fattore verso il metro. Un valore che non è un numero non produce un'eccezione: l'etichetta mostra un trattino, e alla pressione di Invio nel campo compare un messaggio d'errore.

Soluzione: progetto `sol-d05-c09`, package `it.unibs.pajc.redolfi.e03`.

Attenzione 9.3: La conversione in due passi

Scrivere una tabella di fattori per ogni coppia di unità (36 numeri, e una nuova riga e una nuova colonna per ogni unità in più) è il modo più lungo e più fragile. Ogni conversione passa per il metro: si moltiplica per il fattore dell'unità di partenza e si divide per quello dell'unità di arrivo.

Esercizio 9.4: Il visualizzatore di immagini (⋆⋆)

Allena: `JFileChooser` per le cartelle, modelli e renderer, un pannello che si adatta. Una finestra con un menu "Apri cartella…", che sceglie una cartella con un `JFileChooser`; a sinistra la lista delle immagini della cartella (file `.png`, `.jpg`, `.gif`), con un renderer che mostra il nome e, in grigio, la dimensione in kilobyte; al centro un pannello che disegna l'immagine selezionata, la più grande possibile mantenendo le proporzioni e centrata, su sfondo scuro. Le immagini si leggono con `ImageIO.read(file)` e si disegnano con `g.drawImage(img, x, y, w, h, null)`; un file che non si riesce a leggere produce un messaggio d'errore, non un'eccezione.

Soluzione: progetto `sol-d05-c09`, package `it.unibs.pajc.redolfi.e04`.

Attenzione 9.4: Le proporzioni

Scalare l'immagine alla larghezza e all'altezza del pannello la deforma. Il fattore di scala è uno solo, il minore fra $w_{pannello}/w_{immagine}$ e $h_{pannello}/h_{immagine}$, e si applica alle due dimensioni; il margine che avanza in una delle due direzioni serve a centrarla. E l'immagine si legge una volta, alla selezione, non a ogni `paintComponent`.

Esercizio 9.5: L'omino articolato (⋆⋆)

Allena: trasformazioni annidate, `create` e `dispose`. Un pannello che disegna un omino stilizzato centrato e scalato sul lato minore: busto, testa, due braccia e due gambe, ciascun arto fatto di due segmenti (braccio e avambraccio, coscia e gamba) disegnati come rettangoli arrotondati. Quattro slider regolano l'angolo della spalla e del gomito destri, dell'anca e del ginocchio sinistri. Ogni segmento si disegna nel sistema del segmento a cui è attaccato: l'avambraccio ruota attorno al gomito, che sta in fondo al braccio, che ruota attorno alla spalla.

Soluzione: progetto `sol-d05-c09`, package `it.unibs.pajc.redolfi.e05`.

Attenzione 9.5: L'avambraccio che non segue il braccio

Se l'avambraccio si disegna ripartendo dal sistema del busto, con il suo angolo e la posizione del gomito calcolata a mano, ruotando la spalla l'avambraccio resta dov'era. Il contesto dell'avambraccio nasce dalla copia di quello del braccio, dopo la traslazione fino al gomito: le rotazioni si sommano da sole.

Esercizio 9.6: La dama (⋆⋆⋆)

Allena: stato separato dal disegno, conversione fra pixel e caselle, eventi del mouse. Una scacchiera da dama 8 per 8, centrata e adattata alla finestra, con le dodici pedine per parte sulle caselle scure. Lo stato è una matrice di caselle (vuota, bianca, nera) e il turno; il pannello lo disegna e non ne conserva altro. Il primo click su una pedina del giocatore di turno la seleziona (evidenziata); il click su una casella scura vuota in diagonale in avanti la sposta, e passa il turno; la presa (salto di una pedina avversaria in diagonale) toglie la pedina saltata. Una riga di stato dice di chi è il turno. Damone e prese multiple sono un'estensione facoltativa.

Soluzione: progetto `sol-d05-c09`, package `it.unibs.pajc.redolfi.e06`.

Attenzione 9.6: La casella sotto il mouse

Il click arriva in pixel, e la casella si ottiene con la stessa geometria usata per disegnare: `colonna = (x - x0) / lato`. Se il calcolo del lato e del margine è ripetuto, uguale, in `paintComponent` e nel listener, il primo ritocco a uno dei due li rende diversi; sta in un metodo solo, usato da entrambi.

Esercizio 9.7: La navicella (⋆⋆⋆)

Allena: tastiera con più tasti premuti, timer, forme e collisioni. Un piccolo gioco: una navicella triangolare al centro, che le frecce sinistra e destra fanno ruotare e la freccia in alto spinge nella direzione in cui punta (con un po' d'inerzia); uscendo da un bordo si rientra dal bordo opposto. Alcuni asteroidi, cerchi di dimensioni diverse, attraversano lo schermo a velocità costante con lo stesso rientro. Lo stato avanza con un `javax.swing.Timer`, i tasti si registrano in un insieme di tasti premuti; quando la forma della navicella tocca un asteroide il gioco si ferma e una finestra di dialogo dice quanti secondi è durata la partita e chiede se ricominciare.
Suggerimento: la forma della navicella nella sua posizione è un `Path2D` trasformato con un `AffineTransform` (traslazione e rotazione); due forme si toccano se l'intersezione delle loro `Area` non è vuota.

Soluzione: progetto `sol-d05-c09`, package `it.unibs.pajc.redolfi.e07`.

Attenzione 9.7: La finestra di dialogo dentro il passo del timer

La domanda "ricominciare?" è modale, e il listener del timer resta fermo dentro `showConfirmDialog`; ma il ciclo secondario continua a distribuire eventi, compresi i nuovi scatti del timer, che trovano ancora la collisione e aprono un'altra domanda. Il timer va fermato *prima* di aprire la finestra.

Esercizio 9.8: La Lavagna completa (⋆⋆⋆)

Allena: tutto; chiude il problema del capitolo 1. Partendo dalla Lavagna del capitolo 8, aggiungere le funzioni che le mancano per essere uno strumento vero:

- con Maiuscole premuto, trascinare una figura la sposta invece di tracciarne una nuova; la figura sotto il puntatore viene evidenziata con un contorno tratteggiato che scorre (le "formiche in marcia", con un timer e la fase del tratteggio);
- "Annulla" e "Ripeti" (Ctrl+Y) completi: annullano e ripetono le aggiunte, le cancellazioni con il tasto destro e gli spostamenti, in qualunque ordine;
- "Esporta immagine…" salva il disegno in un file PNG con le dimensioni dell'area, senza la griglia;
- "Nuovo" svuota il disegno, chiedendo prima di perdere le modifiche, e il titolo, il conteggio delle figure e l'abilitazione di "Annulla" e "Ripeti" restano sempre coerenti.

Suggerimento: ogni modifica è un oggetto con due metodi, `fai()` *e* `disfa()`*, e due pile, delle modifiche fatte e di quelle disfatte; un'operazione nuova svuota la seconda. Per l'immagine, un* `BufferedImage` *grande quanto l'area, il suo* `createGraphics()`*, e lo stesso codice di disegno di* `paintComponent`*.*

Soluzione: progetto `sol-d05-c09`, package `it.unibs.pajc.redolfi.e08`.

Attenzione 9.8: Le pile che non si svuotano

Dopo un "Annulla" l'utente traccia una figura nuova: se la pila delle modifiche disfatte non viene svuotata, un "Ripeti" successivo rifà un'operazione che si riferisce a uno stato che non esiste più, e il disegno si corrompe. Ogni operazione nuova, che non sia un "Ripeti", svuota la pila del ripeti.

Capitolo 10

Dove andare adesso

La Lavagna è finita, e con lei il percorso della dispensa: comporre una finestra, farla reagire, interrogare l'utente, disegnare, cambiare sistema di riferimento, animare. Ogni argomento ha un seguito, e qui se ne indicano alcuni, con un assaggio ciascuno, per decidere dove andare.

10.1 Separare il Model dalla View

La Lavagna ha cominciato a separare i dati dal loro aspetto: le figure sono record, il pannello le disegna, il file le salva. Ma la lista delle figure vive ancora dentro il pannello, e la finestra la raggiunge attraverso di lui. In un'applicazione grande si va fino in fondo: un *Model* che contiene i dati e le regole e avvisa chi lo osserva, una o più *View* che lo disegnano, e un *Controller* che traduce le azioni dell'utente in operazioni sul Model. È l'architettura MVC, che Swing stesso usa in piccolo in ogni componente con il suo modello, e che `java.beans.PropertyChangeSupport` rende semplice da scrivere:

```
public class ModelloLavagna {
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);
    private final List<Figura> figure = new ArrayList<>();

    public void aggiungi(Figura f) {
        figure.add(f);
        pcs.firePropertyChange("figure", null, List.copyOf(figure));
    }

    public void addPropertyChangeListener(PropertyChangeListener l) {
        pcs.addPropertyChangeListener(l);
    }
}
```

Con un Model così, due View della stessa Lavagna (l'area grande e una miniatura) restano allineate senza conoscersi, e il Model si prova senza aprire nessuna finestra.

10.2 La grafica per pixel

Tutto ciò che si è disegnato è grafica vettoriale: forme descritte con la geometria. Molti problemi chiedono invece di decidere il colore di ogni pixel, uno per uno: un frattale come l'insieme di Mandelbrot, un filtro su una fotografia, una mappa di calore. Lo strumento è `java.awt.image.BufferedImage`, un'immagine in memoria con `setRGB(x, y, colore)` e `getRGB(x, y)`, che `paintComponent` disegna con `drawImage`. Calcolare milioni di pixel richiede tempo, e qui la grafica incontra i thread: l'immagine si calcola fuori dal thread degli eventi, a strisce in parallelo, e si mostra man mano che è pronta.

10.3 Tabelle, alberi e testo

Swing ha componenti per i dati strutturati che questa dispensa ha lasciato fuori. `JTable` mostra una griglia di righe e colonne, con il suo modello (`AbstractTableModel`) e i suoi renderer, ordinamento e filtri compresi; `JTree` una gerarchia, come le cartelle di un disco. `JTextPane` e `JEditorPane` mostrano testo con stili, colori e immagini, e i loro `Document` permettono di evidenziare la sintassi o di controllare ciò che si scrive. Lo schema è sempre quello del capitolo 3: un modello con i dati, un componente che lo osserva, renderer che decidono l'aspetto.

10.4 L'aspetto

Le schermate di questa dispensa usano l'aspetto Metal, quello predefinito. Una riga all'inizio del `main`, con `UIManager.setLookAndFeel`, lo cambia per tutta l'applicazione: l'aspetto "di sistema", il cui nome restituisce `UIManager.getSystemLookAndFeelClassName()`, imita il sistema operativo, e librerie esterne come FlatLaf offrono aspetti moderni, chiari e scuri, che rendono un programma Swing indistinguibile da un'applicazione recente. Il codice non cambia, ed è un'altra conseguenza della separazione fra componenti e loro aspetto.

10.5 Accessibilità e internazionalizzazione

Un'interfaccia si usa anche senza mouse, con un lettore di schermo, in un'altra lingua. Swing aiuta in tutti e tre i casi: la navigazione con Tab fra i componenti, le scorciatoie e i mnemonici (`setMnemonic`), le descrizioni per i lettori di schermo, che si assegnano attraverso `getAccessibleContext()`, i testi in file di risorse (`ResourceBundle`) invece che nel codice. Sono abitudini che costano poco se si prendono all'inizio, e molto se si aggiungono alla fine.

10.6 Oltre Swing

Swing non è l'unico modo di fare interfacce in Java. JavaFX è la libreria più recente, con un grafo della scena al posto dei componenti, proprietà osservabili ovunque e la possibilità di descrivere le finestre in un file a parte; le idee di fondo (eventi, listener, un solo thread per l'interfaccia, Model e View separati) sono le stesse di questa dispensa. Lo stesso vale, con nomi diversi, per le interfacce web e per quelle dei telefoni: chi ha capito il ciclo evento, stato, disegno lo ritrova ovunque.

10.7 Nelle altre dispense

Thread e concorrenza spiega che cosa c'è sotto la coda degli eventi, perché un listener non deve mai durare a lungo e come eseguire un lavoro lungo con `SwingWorker`, riportandone i risultati e l'avanzamento nell'interfaccia. *Dal nome alla funzione* riprende le lambda e i method reference che in questa dispensa sono diventati listener. *Java Fast Forward* contiene i record, le collezioni e la lettura e scrittura dei file usati dalla Lavagna.

10.8 Riferimenti

- Oracle, *The Java Tutorials*: “Creating a GUI with Swing” (componenti, layout, eventi) e “2D Graphics” (`Graphics2D`, forme, tratti, trasformazioni). Datati in alcune parti, ma ancora la guida più completa.
- La documentazione delle API dei package `javax.swing`, `java.awt` e `java.awt.geom`, in particolare le pagine di `JComponent`, `Graphics2D`, `BasicStroke` e `AffineTransform`.
- M. Loy, R. Eckstein, D. Wood, J. Elliott, B. Cole, *Java Swing*, O’Reilly: il riferimento sistematico sui componenti.
- C. Haase, R. Guy, *Filthy Rich Clients*, Addison-Wesley: animazioni ed effetti grafici in Swing, scritto da due sviluppatori della libreria.
- J. Bloch, *Effective Java*: per le scelte di progetto (record, immutabilità, interfacce) che la Lavagna ha usato lungo la strada.