GRAFICA BI-DIMENSIONALE IN JAVA

Disegnare in un componente java

Ogni componente Java possiede il metodo:

- `void paint(Graphics g)` che ridisegna il componente

usato dal sistema per ridisegnare il componente.

IN AWT, il metodo `paint` e' "monolitico".

In Swing, il metodo `paint` e' implementato invocando nell'ordine i seguenti tre metodi:

- `paintComponent` disegna il componente
- `paintBorder` disegna i bordi
- `paintChildren` disegna ricorsivamente i figli del componente (se e' un contenitore, i figli sono i componenti che contiene)

Il secondo dei tre metodi e' dovuto al fatto che in Swing un componente puo' avere bordi.

L'ultimo dei tre metodi e' reso necessario dal fatto che componenti Swing sono lightweight.

- un componente heavyweight (come quelli AWT) ha la sua propria finestra, percio' ciascun componente disegna la sua finestra
- un componente lightweight non ha finestra propria, ma e' ospitato in una sotto-area di un suo antenato heavyweight (es. il frame che sta alla radice della gerarchia di contenimento),
  percio' e' questo antenato che si incarica di disegnare ricorsivamente tutti i suoi discendenti

Personalizzare il disegno

Per personalizzare la grafica di un componente, occorre definire una sottoclasse con una nuova implementazione del metodo `paint` (in AWT) o `paintComponent` (in Swing).

Scegliere la classe di componente adatta (es `Button`), in modo da ereditare tutti i comportamenti previsti da tale classe. Esempi:

- Per bottone la capacita' di catturare `ActionEvent`
- Se non sono necessari comportamenti particolari si puo' usare un semplice pannello

Ridefinire il metodo `paint` o `paintComponent` inserendovi il codice per compiere il disegnamento voluto. Chiamare sempre `super.paint` o `super.paintComponent` all'inizio in modo da effettuare anche tutto il redisegmanento previsto di default (es. pulitura dello sfondo).

Classi per disegnare

Il metodo `paint` / `paintComponent` ha come argomento un oggetto di classe `Graphics` che viene passato automaticamente dal sistema.

```
void paint/paintComponent (Graphics g)
{
  ....
}
```

Tutto il disegno si fa chiamando metodi della classe Graphics sull'oggetto `g`.

La classe Graphics contiene:

- informazioni sullo **stato** del sistema grafico (attributi pittorici, trasformazioni)
- **primitive** per disegnare (varie forme geometriche)
- metodi per agire sullo stato (leggere e impostare i valori)

In realta' l'oggetto `g` che viene passato dal sistema a `paint` non e' semplicemente di classe Graphics, ma in particolare e' della sottoclasse Graphics2D.

- Classe Graphics contiene funzionalita' grafiche piu' semplici, in genere sufficienti a disegnare l'aspetto dei componenti java
- Classe Graphics2D contiene funzionalita' grafiche piu' avanzate

Essendo di classe Graphics2D (sottoclasse), `g` e' anche di classe Graphics (superclasse), e normalmente si usa come se fosse di classe Graphics
Se voglio vederlo come oggetto di classe Graphics2D (per poter vedere le funzionalita' avanzate), devo fare una conversione esplicita (cast):

```
Graphics2D g2 = (Graphics2D)g;
```
dopo di che in `g2` ho sempre `g` ma visto come oggetto di classe Graphics2D

Noi vedremo le funzionalita' di Graphics2D.

Ingredienti per disegnare

Quello che si disegna dipende da:

- primitive geometriche: geometria degli oggetti che disegno
- attributi pittorici: loro apparenza grafica (colore, tipo di tratto...)
- trasformazioni di coordinate: agiscono sulla geometria

- altri parametri di stato del sistema grafico: perimetro di clipping, modo di composizione del colore...

Sistemi di coordinate e trasformazioni

- **Device space** = sistema di coordinate fisico del componente su cui disegno, ancorato allo schermo ed espresso in pixel
- **User space** = sistema di coordinate logico in cui il programmatore definisce le primitive
- **Trasformazioni** = servono per passare da user space a device space

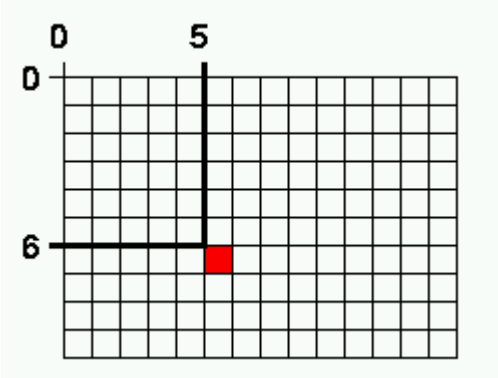
Esempio:
Posso esprimere le mie primitive (linee, poligoni) in un sistema di coordinate numeri reali tra -1 ed 1, e poi visualizzarle in una finestra di dimensioni che voglio. Se la finestra e' di 300x200 pixel, cio' significa che i pixel dell'immagine raster generata saranno in un sistema di coordinate intere x tra 0 e 300, y tra 0 e 200. Nel definire le primitive non mi devo preoccupare di quanto sara' grande la finestra di visualizzazione, a questo ci penseranno le trasformazioni.

Device space

Ogni componente ha un sistema di coordinate intere (in pixel) con origine `(0,0)` in alto a sinistra e punto in basso a destra di coordinate `(d.width-1,d.height-1)` dove `Dimension d = getSize()` sono le dimensioni in pixel del componente. Le coordinate x crescono da sinistra a destra, le coordinate y crescono dall'alto verso il basso.

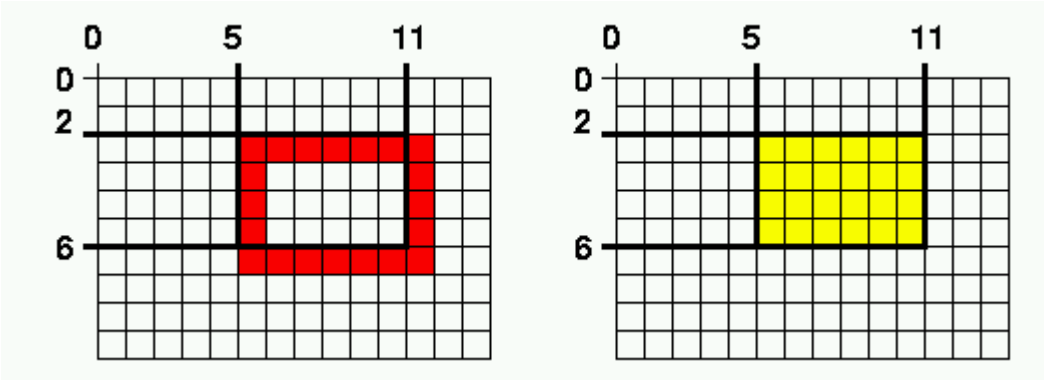
In Swing devo tenere conto che parte del rettangolo della componente puo' essere occupata dal bordo. `Insets i = getInsets()` ritorna informazioni sullo spessore del bordo.
La parte di rettangolo libera dal bordo ha angolo in alto a sinistra `(i.left,i.top)` e in basso a destra `(d.width-i.right-1,d.height-i.bottom-1)`.

Le coordinate sono infinitamente sottili e collocate tra un pixel e l'altro. La punta scrivente traccia la linea sul pixel immediatamente a destra e in basso.



Cio' significa che, se traccio un rettangolo pieno e il suo contorno vuoto dando le stesse coordinate, il contorno si estende una riga di pixel in piu' in basso e una colonna di pixel

a destra rispetto al rettangolo pieno.



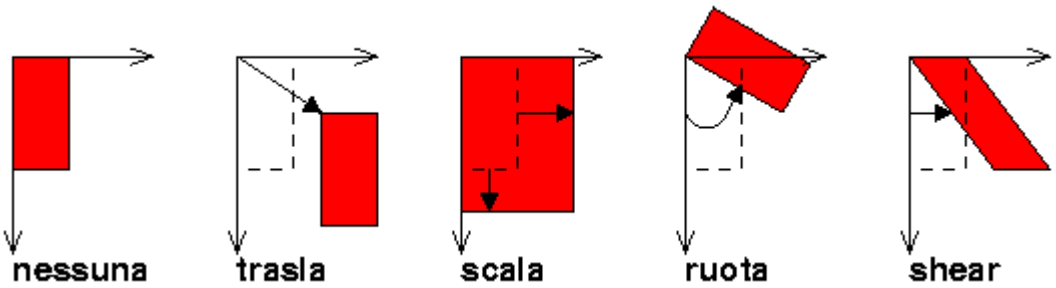
User space

Sistema di coordinate logico in cui il programmatore esprime le primitive da disegnare.

Una **trasformazione affine** determina come portare le primitive da User space a Device space.

Di default la trasformazione e' l'identita' (nessuna trasformazione, i due spazi di coordinate coincidono).
Quindi se la finestra e' di 300x200 pixel io devo definire primitive che abbiano coordinate entro questi limiti, le (parti di) primitive che cadono fuori non saranno visibili. Se l'utente cambia dimensioni alla finestra, la scena puo' rimenere decentrata e/o in parte tagliata fuori.

Posso invece impostare trasformazioni personalizzate: traslazioni, rotazioni, scalature e shear.
In questo modo definisco le primitive con coordinate entro limiti che decido io, poi affido alle trasformazioni il compito di "farle stare" nella finestra, in base alle dimensioni correnti di questa.



Nota: Graphics2D ha trasformazioni e quindi distinzione tra device space e user space. Invece Graphics non ha trasformazioni e devo esprimere le primitive direttamente in device space.

Primitive e attributi

Primitive ("che cosa" posso disegnare):

- Primitive 1-dimensionali (linee): nomi dei metodi che le disegnano iniziano con "draw"
- Primitive 2-dimensionali (aree piene): nomi dei metodi che le disegnano iniziano con "fill"
- Stringhe di caratteri
- Immagini

Vedremo dopo in dettaglio le primitive.

Attributi pittorici ("come" lo posso disegnare):

- COLOR = colore con cui disegno
- STROKE = stile del tratto (spessore, tratteggio, modo di unire gli angoli) per le primitive 1-dimensionali
- PAINT = trama di riempimento (uniforme, sfumata, con tessitura) per le primitive 2-dimensionali
- FONT per le stringhe

Modo di composizione del colore

Quando due primitive vanno a sovrapporsi sulla stessa area della finestra, quale delle due "vince"? Lo stabilisce il modo di composizione del colore:

- la primitiva disegnata per ultima copre la precedente (default)
- la primitiva disegnata per prima copre la seguente
- di ciascuna delle due e' disegnata solo la parte esterna all'altra, mentre la parte intersezione non e' disegnata affatto
- altre modalita'...

Clipping

Clip = tagliare via (qui: dal disegno). Non disegnare quello che cade fuori da una certa area specificata da un **perimetro di clipping** (**clipping path**).

Di norma il perimetro di clipping e' la finestra: cio' che ha coordinate che eccedono i limiti delle coordinate di device e' tagliato via e non si vede.

Posso specificare perimetri di clipping (clipping path) aggiuntivi. Allora anche cio' che cade dentro la finestra ma fuori dal perimetro di clipping sara' tagliato.

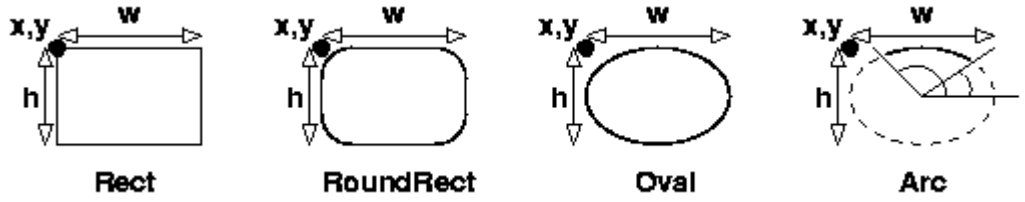
Primitive disponibili

- Metodi della classe Graphics2D per tracciare (draw) o riempire (fill) direttamente una figura geometrica dati dei parametri numerici che la descrivono.
  Molti di questi metodi sono comuni alla superclasse Graphics.
- Classe Shape apposita per rappresentare figure geometriche, che si possono poi disegnare chiamando metodi (draw o fill) della classe Graphics2D.

Disegno diretto

Metodi per disegnare direttamente una figura. I parametri sono interi esprimenti numero di pixel, cioe' le primitive vanno fornite direttamente in device space.

- `clearRect(x,y, width,height)`: pulisce il rettangolo indicato riempiendolo col colore di sfondo.
- `void draw/fillRect(x,y, width,height)`: Disegna **rettangolo** vuoto o pieno.
- `void draw/fillRoundRect(x,y, width,height, arcWidth,arcHeight)`:
  Disegna **rettangolo con angoli smussati**, vuoto o pieno.
- `void draw/fill3DRect(x,y, width,height, arcWidth,arcHeight)`:
  Disegna **rettangolo con effetto di rilievo 3D**, vuoto o pieno.
- `draw/fillOval(x,y, width,height)`: Disegna **ellisse**, vuoto o pieno.
- `draw/fillArc(x,y, width,height, startAngle,arcAngle)`: Disegna **arco** circolare o ellittico, vuoto o pieno. Gli angoli sono in gradi, angolo positivo significa rotazione in senso antiorario, negativo in senso orario. Centro dell'arco e' il centro del rettangolo di origine (x,y) e dimensioni width,
  height.



- `drawLine(x1,y1, x2,y2)`: Disegna **segmento di retta** tra (x1,y1) e (x2,y2).
- `draw/fillPolygon(arrayX, arrayY, n)`: Disegna **poligono** con `n` vertici definiti dai due array di x e y.
- `drawPolyline(arrayX, arrayY, n)`: Disegna **spezzata poligonale** con `n` vertici definiti dai due array di x e y.
- `drawString(stringa,x,y)`: Disegna **stringa** iniziando alle coordinate (x,y). La stringa si estende a destra di x e sopra y, cioe' nelle ascisse maggiori di x e nelle ordinate MINORI di y.
- `drawImage(Image, x,y, ImageObserver)`: Disegna **immagine** alla posizione (x,y). Come image observer si passa il componente stesso (this).
- `drawImage(Image, x,y, width,height, ImageObserver)`: disegna **immagine** alla posizione (x,y), **scalata** alla larghezza ed altezza indicate.
  Es. per disegnare immagine che occupa tutta la componente:
- `Dimension d = getSize();`
- `g.drawImage (image, 0,0, d.width,d.height, this);`

Attributi disponibili

Colore

Il colore con cui disegno si legge con `Color getColor()` e si imposta con `setColor(Color c)`.

Un colore si crea specificando le componenti RGB con `new Color(r,g,b)` dove r,g,b sono interi tra 0 e 255 oppure float tra 0.0 e 1.0. La classe Color ha anche costanti per i colori piu' comuni: Color.black, Color.white, Color.red, Color.green...

Tratto

Il tratto e' il modo di tracciamento delle linee. Si legge con `Stroke getStroke()` e si imposta con `setStroke(Stroke s)`.

Il tratto e' implementato dalla classe BasicStroke, e puo' essere:

- uniforme, che si crea con `new BasicStroke(spessore)` dove lo spessore e' un float
- tratteggiato...

Posso specificare anche il modo di gestire gli angoli (appuntiti o arrotondati) ed altri dettagli...

Trama

La trama e' il modo di riempimento delle aree. Si legge con `Paint getPaint()` e si imposta con `setPaint(Paint p)`.

La trama p puo' essere:

- un colore (trama uniforme), classe `Color` gia' vista
- un colore sfumato, classe `GradientPaint`, che si crea specificando due colori diversi c1 e c2 e due punti p1 e p2: il colore sara' c1 in p1, c2 in p2, e sfumato tra i due nei punti intermedi
- una tessitura, classe `TexturePaint`, che si crea specificando un'immagine e un rettangolo di ancoraggio: l'immagine sara' collocata inizialmente nel rettangolo e poi replicata tante volte quante occorre

Font

La font e' il tipo di carattere usato per tracciare le stringhe. Si legge e si imposta con `Font getFont()` e `setFont(Font f)`.

Una font si crea con: `new Font (nome, stile, grandezza)` dove

- il nome e' una stringa che identifica la font, es: "Courier"
- lo stile e' un intero che puo' essere una delle costanti Font.PLAIN, Font.BOLD, Font.ITALIC o un "or" tra queste (es. sia bold che italic)

- la grandezza e' espressa in numero di pt, es: 10, 12, 24

Il default nella mia versione e' "Dialog", PLAIN, 12 pt. Per avere tutte le font disponibili sulla macchina:

```
GraphicsEnvironment genv =
        GraphicsEnvironment.getLocalGraphicsEnvironment();
Font[] f = genv.getAllFonts();
```

Esempio

Pannello con grafica personalizzato: ExDraw1.java.
La funzione paintComponent riempe un rettangolo sfumato, vi disegna dentro un fumetto con scritto grande "Ciao!", traccia con tratto spesso tre linee alla base del fumetto.

Classi per figure geometriche (Shape)

La classe `Shape` fornisce nelle sue sottoclassi vari tipi di forme geometriche che posso disegnare usando i metodi della classe Graphics2D:

- `draw(Shape s)`: traccia il contorno
- `fill(Shape s)`: riempe l'interno

Sottoclassi di shape:

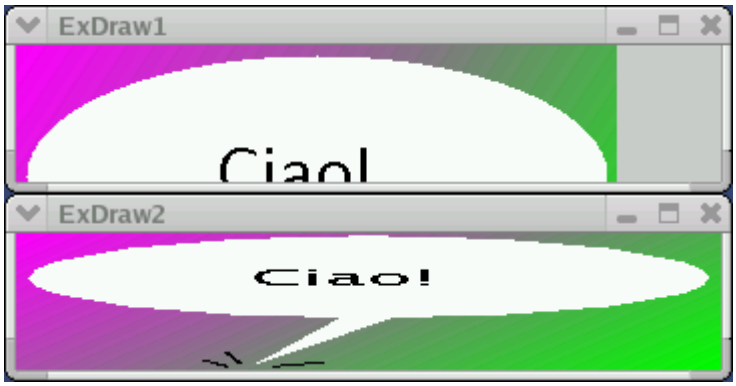
- Figure dotate di un rettangolo che le contiene: `Arc2D`, `Ellispe2D`, `Rectangle2D`, `RoundRectangle2D`
- Segmenti di varia forma: `Line2D` (di retta), QuadCurve2D (di curva quadrica), `CubicCurve2D` (di curva cubica)
- Linea spezzata formata da segmenti di tipo vario: `GeneralPath`
- Figura 2D generale su cui posso eseguire operazioni insiemistiche: Area

Le shape sono contenute nel package java.awt.geom, che occorre percio' importare.

Alcune delle figure di cui sopra corrispondono a funzioni draw/fill gia' viste (es: drawShape di un Rectangle2D equivale a drawRect).
Altre sono nuove. In piu' e' possibile esprimere le coordinate con float e double.

La possibilita' di disegnare le shape esiste solo in Graphics2D, non in Graphics. Le coordinate possono essere numeri reali perche' Graphics2D ammette coordinate logiche diverse da quelle fisiche (cioe' dai pixel).

L'esempio ExDraw1.java visto prima disegna sempre la scena in 300x200 pixel. Invece l'esempio ExDraw2.java disegna la scena sempre in tutta la finestra, anche quando l'utente la deforma. Fa uso di trasformazioni che vedremo dopo.



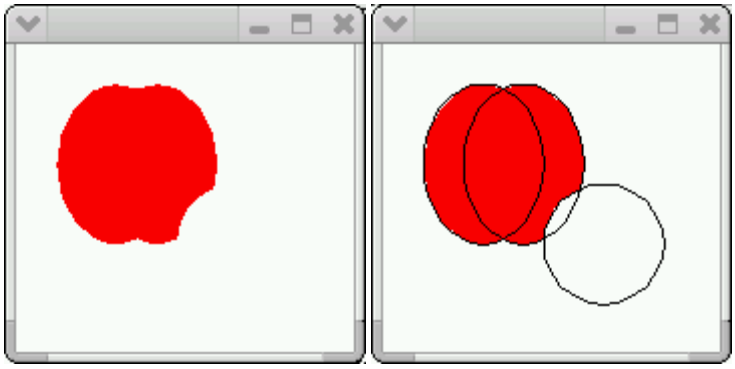
Classe Area

La classe `Area` prevede le operazioni add, subtract, intersect, exclusiveOr con argomento un'altra area. Ogni shape puo' essere trasformata in un'area. Posso costruire una forma complessa aggiungendo, sottraendo, intersecando e facendo l'OR esclusivo di forme semplici.

Esempio:
Mela con morso ottenuta unendo due ellissi e sottraendone un altro: MelaMorsa.java

```
Ellipse2D.Double sinistra, destra, morso;
Area a;
sinistra = new Ellipse2D.Double();
sinistra.setFrame(20, 20, 60, 80);
destra = new Ellipse2D.Double();
destra.setFrame(40, 20, 60, 80);
morso = new Ellipse2D.Double();
morso.setFrame(80, 70, 60, 60);
a = new Area();
a.add( new Area(sinistra) );
a.add( new Area(destra) );
a.subtract( new Area(morso) );
```

Cosa si vede Cosa c'e sotto



Trasformazioni

Graphics2D contiene una trasformazione affine che e' usata per traslare, ruotare, scalare e deformare le primitive durante la loro traduzione da user space a device space.

La trasformazione e' un oggetto di classe AffineTransform. Internamente e' implementata come una matrice 3x3 (in coordinate omogenee).

Posso creare un una nuova trasformazione con:

- `AffineTransform.getTranslateInstance(spostamentoX, spostamentoY)`
- `AffineTransform.getRotateInstance(angolo in radianti)`
- `AffineTransform.getScaleInstance(fattoreX, fattoreY)`
- `AffineTransform.getShearInstance(fattoreX, fattoreY)`

e poi impostarla con il metodo di Graphics2D `setTransform`.

Oppure modificare la trasformazione corrente concatenandone un'altra con i metodi di Graphics2D:

- `translate(spostamentoX, spostamentoY)`
- `rotate(angolo in radianti)`
- `scale(fattoreX, fattoreY)`
- `shear(fattoreX,fattoreY)`

La rotazione e' attorno all'origine, la scalatura tiene come punto fermo l'origine.

Posso ottenere qualsiasi trasformazione componendo queste.

Importante: la composizione delle matrici avviene moltiplicando la nuova matrice a destra di quella corrente, quindi PRIMA viene eseguita la trasformazione corrispondente alla NUOVA matrice e DOPO la trasformazione corrispondente alla VECCHIA matrice.

In pratica le trasformazioni sono eseguite IN ORDINE INVERSO a come le specifico nel codice.

Esempio:
Parto dalla matrice identita' I.

`g2.translate(...);` matrice = I T = T dove T = matrice di traslazione
`g2.rotate(...);` matrice = T R dove R = matrice di rotazione
Risultato: nell'ordine prima ruoto poi traslo i punti

Esempio di trasfomazione composta

Rotazione attorno a un punto C = (xC,yC) diverso da origine:

1. traslo di (-xC,-yC) per portare C nell'origine del nuovo riferimento
2. ruoto dell'angolo alpha voluto attorno a origine
3. traslo di (xC,yC) per portare C alla posizione originale

Nel codice:

3. g2.translate(xC,yC);
2. g2.rotate(ang);
1. g2.translate(-xC,-yC);

Accortezza

La trasformazione va cambiata solo momentaneamente durante l'esecuzione di paint, alla fine del codice di paint bisogna ripristinare la trasformazione che c'era prima. Altrimenti si potrebbero avere effetti inaspettati. Per fare cio':

- all'inizio di paint salvare la trasformazione in una variabile chiamando getTransform
- poi modificarla e disegnare
- alla fine di paint chiamare setTransform con la trasformazione salvata

Animazione

Animazione = cambiamento dinamico della grafica mostrata su un componente.

Ad intervalli regolari (gestiti con un timer) oppure in seguito ad azioni dell'utente si ridisegna il componente cambiandone la grafica.

Forzare il redisegnamento

Di norma il metodo `paint` e' chiamato dal sistema, attraverso canali suoi, quando il sistema ritiene che il componente vada ridisegnato (es. quando la finestra che lo contiene e' mappata o torna visibile dopo essere stata oscurata da altre, quando viene ridimensionata...).

L'animazione rende necessario ridisegnare anche in casi diversi da quelli previsti dal sistema.

Il programma NON DEVE chiamare direttamente `paint`! Per provocare da programma il ridisegnamento del componente bisogna usare il metodo:

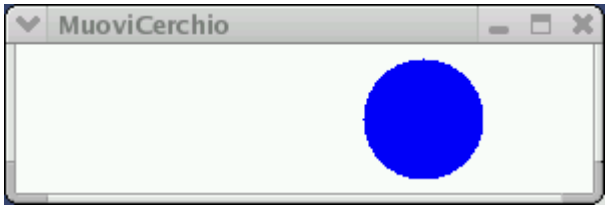
- `void repaint()` che mette in coda una richiesta di redisegnamento per il componente

Siccome la gestione della coda e' asincrona, puo' capitare che piu' chiamate a `repaint` vengano collassate in una sola chiamata a `paint`.

L'animazione si ottiene cambiando alcuni parametri usati dentro la funzione paint e poi invocando repaint.

Esempio

Pannello con cerchio che scorre avanti e indietro per un pannello: MuoviCerchio.java
Una variabile contiene l'ascissa corrente ed e' usata da paint per disegnare il cerchio.
C'e' un timer che ogni tanti millisecondi cambia il valore della variabile e invoca repaint.



Interazione con l'utente

Vi sono due tipi di azioni interessanti che l'utente puo' compiere sull'area grafica:

- Azioni sullo spazio indipendentemente dalla presenza di primitive. Esempio: Cliccare in un punto, il programma poi compiera' qualche operazione con le coordinate di quel punto (es: disegnare qualcosa in quel punto)
- Azioni sulle primitive presenti nello spazio. Esempio:
  Cliccare sopra una primitiva per selezionarla, il programma poi compiera' qualche operazione sulla primitiva selezionata (es: cancellarla)

Azioni sullo spazio

Si catturano associando al pannello gli event listener opportuni.

Nell'esempio associo un MouseListener che nel suo metodo mouseClicked compia l'operazione corrispondente. Le coordinate del punto cliccato si ottengono chiamando getX() e getY() sull'evento MouseEvent.

Azioni sulle primitive

Come sopra, ma in piu' richiedono un modo per conoscere che primitiva e' disegnata nel punto dove e' avvenuto il click.

E' necessario aver disegnato le figure che vogliamo rendere sensibili al click usando oggetti di classe shape ed il metodo fill(Shape s) della classe Graphics2D.

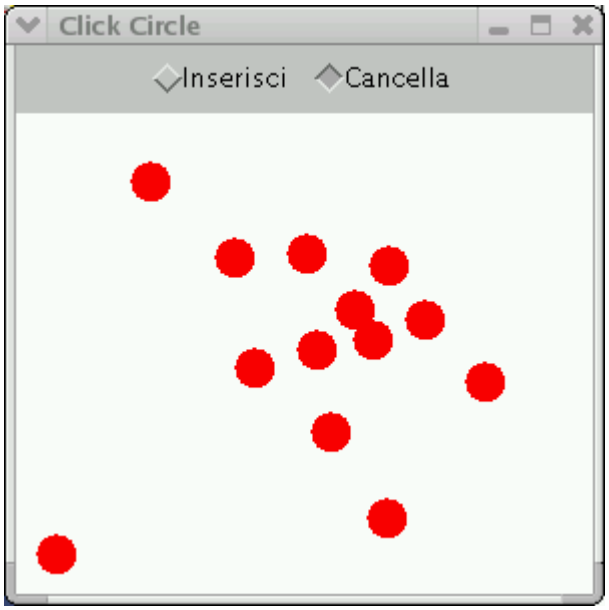
Si scorre la lista delle shape che sono state disegnate (e' necessario averle memorizzate) e si chiede a ognuna se e' stata interessata dal click, usando i metodi della classe Shape:

- `contains(x,y)` oppure `contains(Point2D p)`: controlla se la figura contiene il punto
- `contains(Rectangle2D r)`: controlla se la figura contiene completamente il rettangolo
- `intersects(x,y,w,h)` oppure `intersects(Rectangle2D r)`: controllano se la figura interseca il rettangolo specificato

Esempio

Programma che permette di disegnare cerchi e di cancellarli: ClickCircle.java

- Se l'operazione corrente e' "insert", cliccando su un punto appare un cerchio centrato in quel punto
- Se l'operazione corrente e' "delete", cliccando su un cerchio si cancella il cerchio



Un frame con in alto due bottoni radio "insert" e "delete" per scegliere la modalita' corrente. In basso un'area di disegno.

- se la modalita' e' "insert" cliccando nell'area di disegno appare un cerchio rosso nel punto dove cliccato
- se la modalita' e' "delete" cliccando nell'area di disegno sopra un cerchio, quel cerchio viene cancellato

I cerchi presenti sono tenuti in un array di oggetti di classe Ellipse2D. La reazione al click del mouse e' gestita associando un mouse listener al pannello, che implementa `mouseClicked` cosi':

- quando utente clicca in modalita' "insert" crea un nuovo cerchio e lo aggiunge in fondo all'array (funzione `insertCircle`)
- quando utente clicca in modalita' "delete" scorre l'array fino a trovare (se esiste) un cerchio che contiene il punto cliccato, se trovato lo cancella dall'array spostando indietro gli elementi successivi (funzione `deleteCircle`)
- in ogni caso dopo l'operazione ridisegna il pannello