Cours 2: *Graphics* et création de nouveaux composants

FIP 1 - mise à niveau java

S. Rosmorduc

L'*Event Dispatch Thread* (*EDT*)

- un programme multi-thread («fils d'exécution») exécute plusieurs codes en parallèle
- quand une interface graphique est créée, on lance automatiquement un thread pour la gérer : l'*event dispatch thread*, indépendamment du thread *main*.
- l'EDT gère les *événements* et l'affichage du programme graphique dans une boucle qui:
  - attend un événement
  - dessine ce qui doit être redessiné

La classe JComponent

- classe de base des composants Swing
- méthodes importantes:
  - protected void **paintComponent**(**Graphics** g)
    - dessine le composant quand c'est nécessaire
    - pas appelable directement ;
    - redéfinie dans les composants graphiques
  - public Dimension **getPreferredSize**()
    - permet au composant de dire quelle taille «il a envie» d'avoir.

Exemple de composant

```
public class CarreAvecRond extends JPanel {

  private int centreX = 10, centreY = 10, rayon = 30;
  private Color couleurCercle= Color.BLUE;

  public CarreAvecRond() {
   setBackground(Color.WHITE);
  }

  public Dimension getPreferredSize() {
   return new Dimension(300, 300);
  }

  protected void paintComponent(Graphics g) {
   super.paintComponent(g); // Laisser cette méthode ici.
   g.setColor(couleurCercle);
   g.fillOval(centreX -rayon,
                  centreY - rayon,
                  rayon*2, rayon*2);
  }
}
```

Il faut écrire
au moins ces deux méthodes

On étend
JPanel au lieu de JComponent pour
«bénéficier» d'un remplissage automatique
avec la couleur de fond (essayer en
remplaçant par JComponent)

Ici, la taille préférée est fixe
(elle pourrait dépendre du contenu, par
exemple)

Dessin de l'élément

```
protected void paintComponent(Graphics g) {
  // Effacement de l'ancien dessin:
 super.paintComponent(g);
  // On fixe la couleur pour dessiner:
g.setColor(couleurCercle);
  // On dessine un cercle
  g.fillOval(centreX -rayon,
                    centreY - rayon,
                    rayon*2, rayon*2);
  }
}
```

protected : vous la définissez, mais c'est l'EDT qui l'appelle !

La surface de dessin (de type **Graphics**)

Modifications du composant

```
public class CarreAvecRond extends JPanel {

  private int centreX = 10, centreY = 10, rayon = 30;
  private Color couleurCercle= Color.BLUE;

  public CarreAvecRond() {
   setBackground(Color.WHITE);
  }

  public Dimension getPreferredSize() {
   return new Dimension(300, 300);
  }

  protected void paintComponent(Graphics g) {
   super.paintComponent(g); // Laisser cette méthode ici.
   g.setColor(couleurCercle);
   g.fillOval(centreX -rayon,
                centreY - rayon,
                rayon*2, rayon*2);
  }
}
```

Modifications du composant

- Le composant affiche les données d'un *modèle*, explicite ou implicite
- Ici, le modèle est implicite (pas représenté par un objet)
    - trois int : centreX, centreY, rayon
    - la couleur du cercle
    - la couleur du fond (gérée automatiquement par JPanel)

(en réalité, on aurait pu utiliser foregroundColor à la place de couleurCercle, ça aurait fait un élément de moins à gérer)...

- On ajoute la méthode setCouleurCercle à notre classe et on essaie...

Modification des données...

oh le joli rouge !

repaint()

- Modification des données ➜ il faut redessiner l'objet
- L'appel à la méthode repaint() prévient l'objet graphique qu'il n'est plus à jour
- C'est l'EDT qui decide quand redessiner.
- l'appel à repaint() ne redessine pas immédiatement l'objet et n'est pas coûteux en temps de calcul.

```
public class CarreAvecRond2 extends JPanel {

  private int centreX = 10, centreY = 10, rayon = 30;
  private Color couleurCercle= Color.BLUE;


  ...
  public void setCouleurCercle(Color couleurCercle) {
   this.couleurCercle = couleurCercle;
   repaint();
  }

  public void setCentreX(int centreX) {
   this.centreX = centreX;
   repaint();
  }
  public void setCentreY(int centreY) {
   this.centreY = centreY;
   repaint();
  }
  public void setRayon(int rayon) {
   this.rayon = rayon;
   repaint();
  }
}
```

Intérêt de repaint()

- Le code :

```
carreAvecRond.setCentreX(100);
carreAvecRond.setCentreY(200);
carreAvecRond.setRayon(40);
```

- Appelle trois fois repaint()
- mais le dessin sera fait une seule fois...

revalidate()

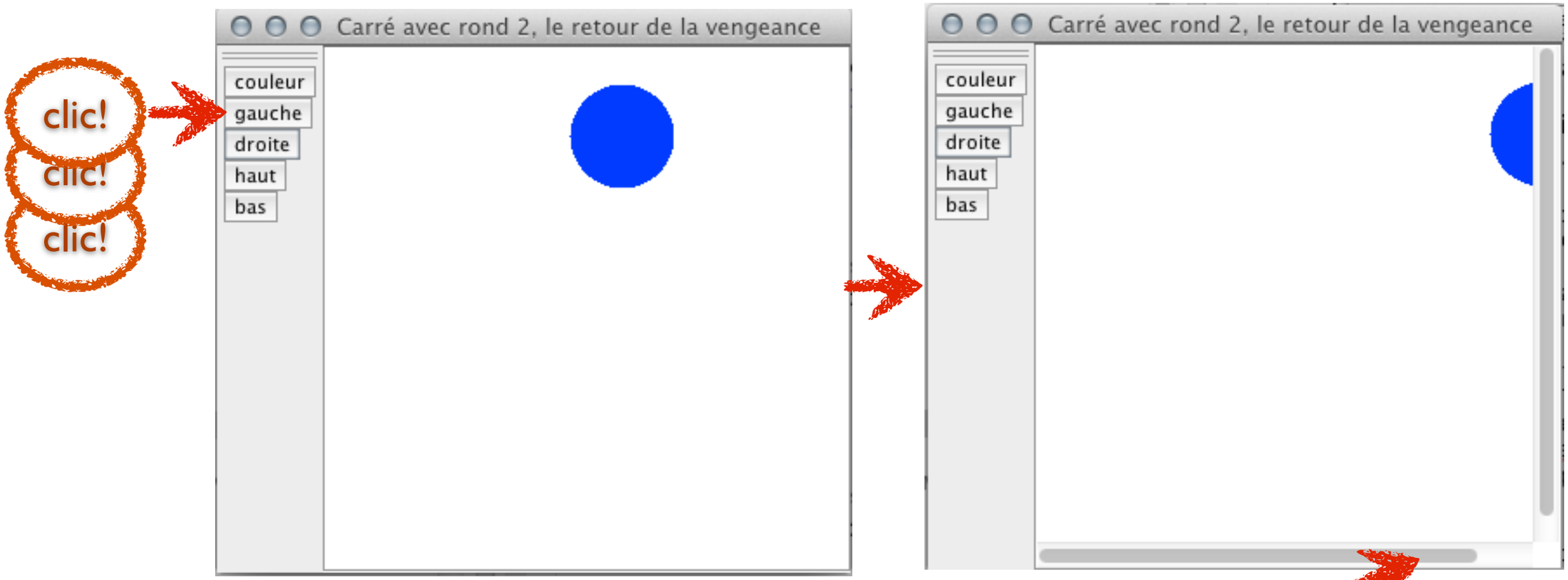
- appelée quand la *taille* de l'objet doit changer
- permet par exemple à un JTextArea de prévenir le JScrollPane qu'il faut modifier la taille des ascenceurs

```
public class CarreAvecRond3 extends JPanel {
  private int centreX = 30, centreY = 30, rayon = 30;
  private Color couleurCercle= Color.BLUE;
...

  @Override
  public Dimension getPreferredSize() {
   int w= Math.max(300, centreX + rayon);
   int h= Math.max(300, centreY+ rayon);
   return new Dimension(w, h);
  }

  public void setCouleurCercle(Color couleurCercle) {
   this.couleurCercle = couleurCercle;
   repaint();
  }

  public void setCentreX(int centreX) {
   if (centreX < rayon) return;
   this.centreX = centreX;
   repaint();
   revalidate();
  }
...
```

Calcule la taille pour contenir le cercle

Changer la couleur ne modifie pas la taille...

Changer la position du cercle modifie la taille...



la taille est modifiée, les ascenseurs apparaissent

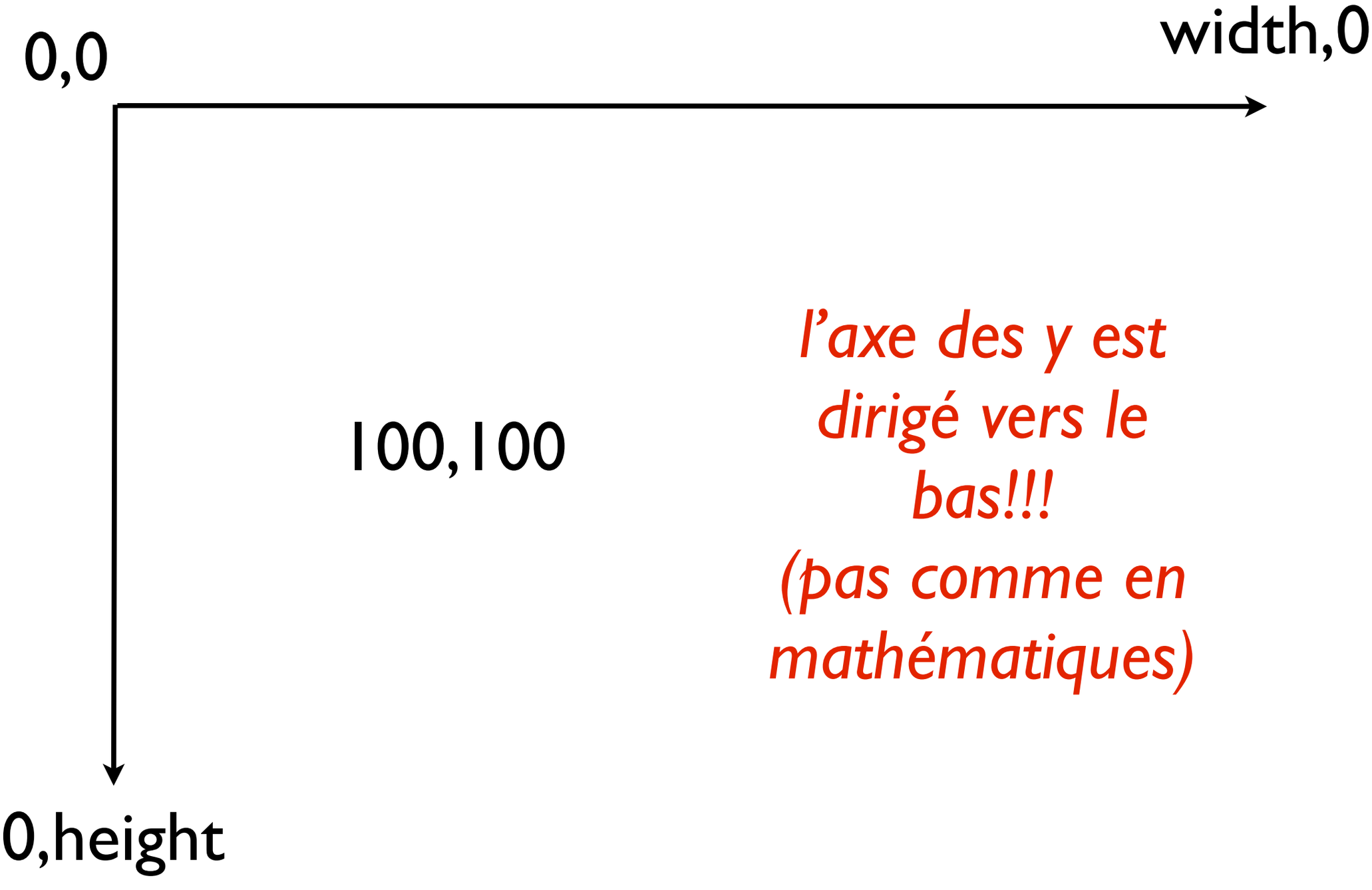
Dimensions

- les méthodes getWidth() et getHeight() renvoient les dimensions actuelles d'un composant graphique.

La classe Graphics

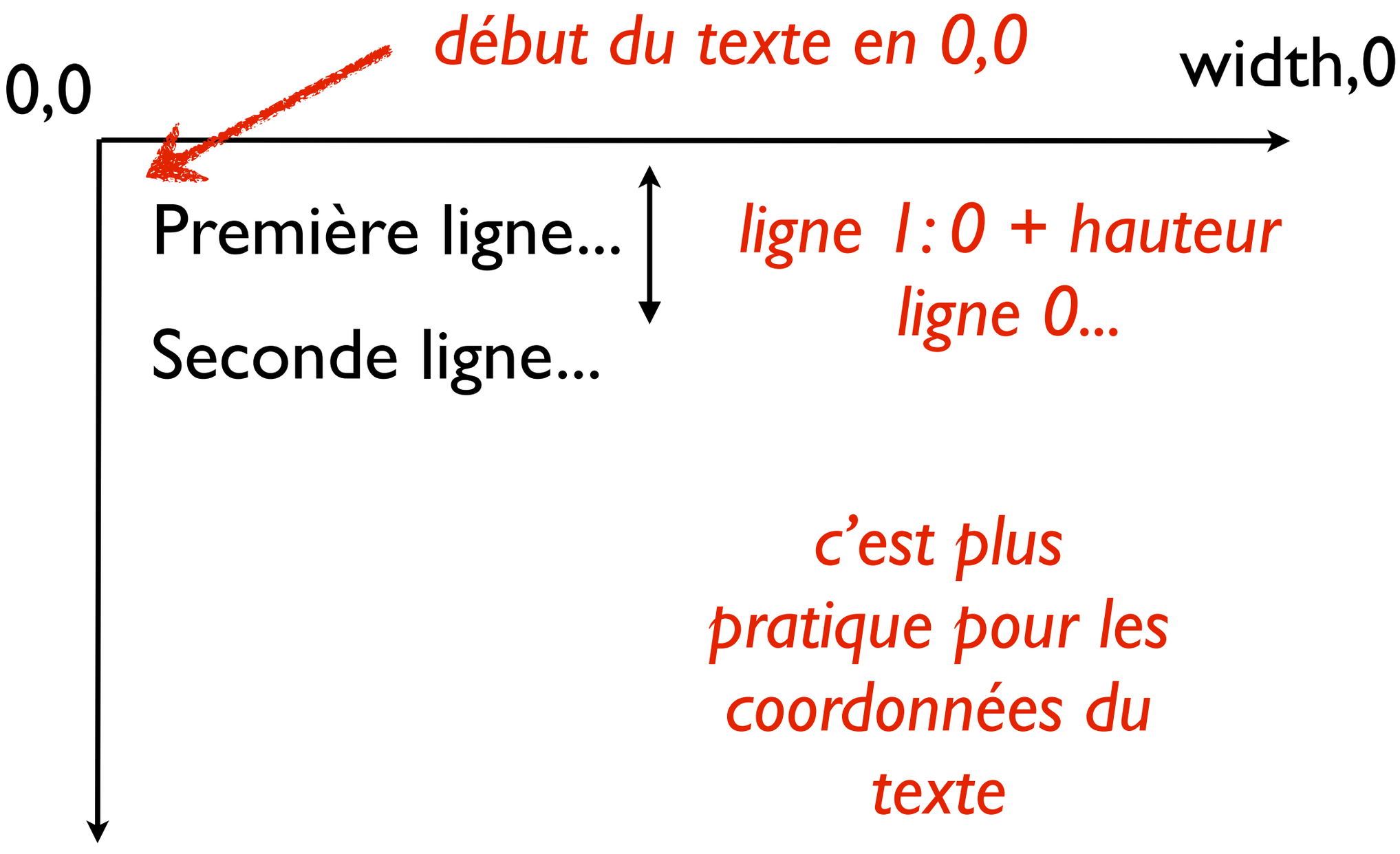
- On n'a pas besoin de créer l'objet Graphics. Il est passé par l'EDT.
- Il représente la surface de dessin, ainsi que les outils à utiliser (couleur de fond, couleur de dessin)
- Sur un objet Graphics, on peut:
  - dessiner: g.drawLine(0,0,100,100);
  - écrire: g.drawString("exemple", 100,100);
  - changer la couleur de dessin : g.setColor(Color.RED);
  - changer la fonte (dans toutes ses caractéristiques) :
  - g.setFont(new Font("Arial", Font.ITALIC, 16)); ...

Repère



l'axe des y est dirigé vers le bas!!!
(pas comme en mathématiques)

Repère

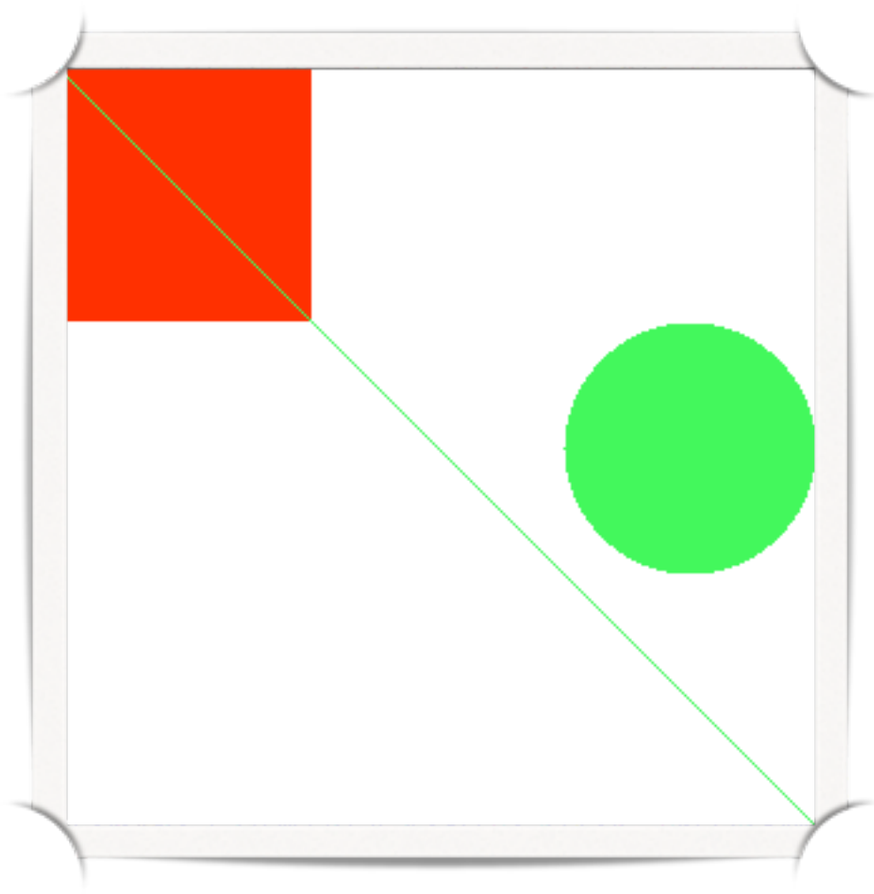


c'est plus pratique pour les coordonnées du texte

Couleurs

- Une couleur de dessin manipulée par
  - setColor(Color c)
  - getColor() : Color
- Quand une couleur est fixée par setColor(), elle est utilisée par les instructions de dessin jusqu'au prochain setColor()
- Il est de bon goût de rétablir la couleur d'origine à la fin du dessin.
- Les couleurs sont souvent données par trois valeurs:
  - rouge, vert, bleu, entre 0 et 255

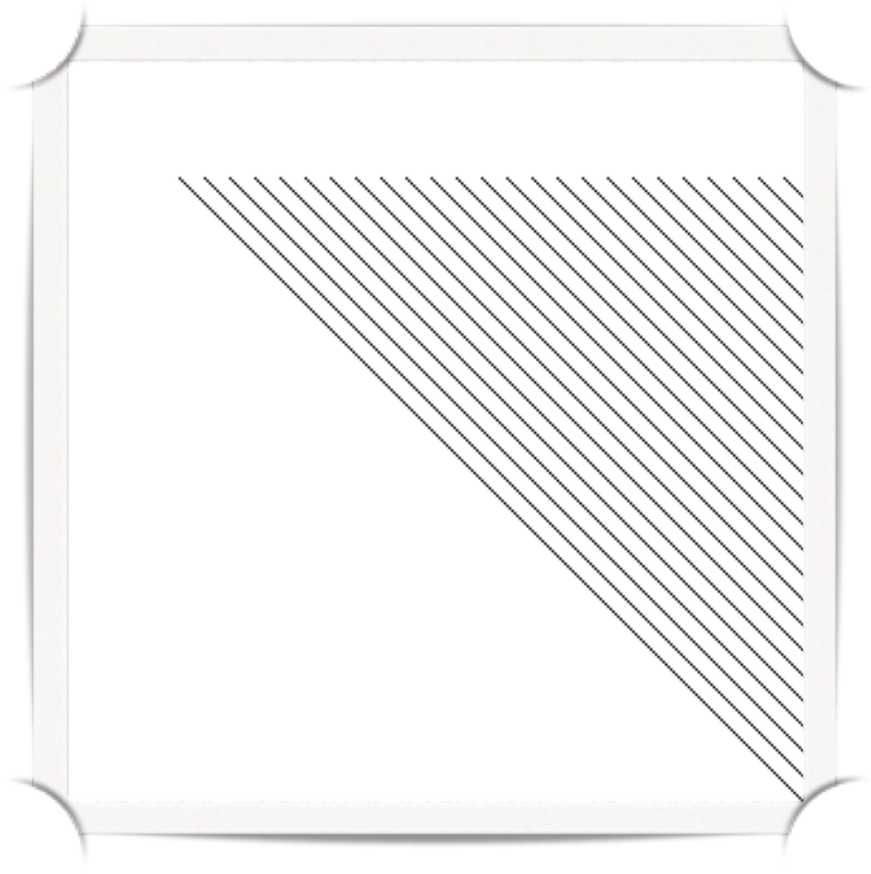
Couleur



```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  Color ancienne= g.getColor();
  g.setColor(Color.RED);
  g.fillRect(0, 0, 100, 100);
  g.setColor(new Color(100,255,100));
  g.drawLine(0, 0, 300, 300);
  g.fillOval(200, 100, 100, 100);
  g.setColor(ancienne);
}
```

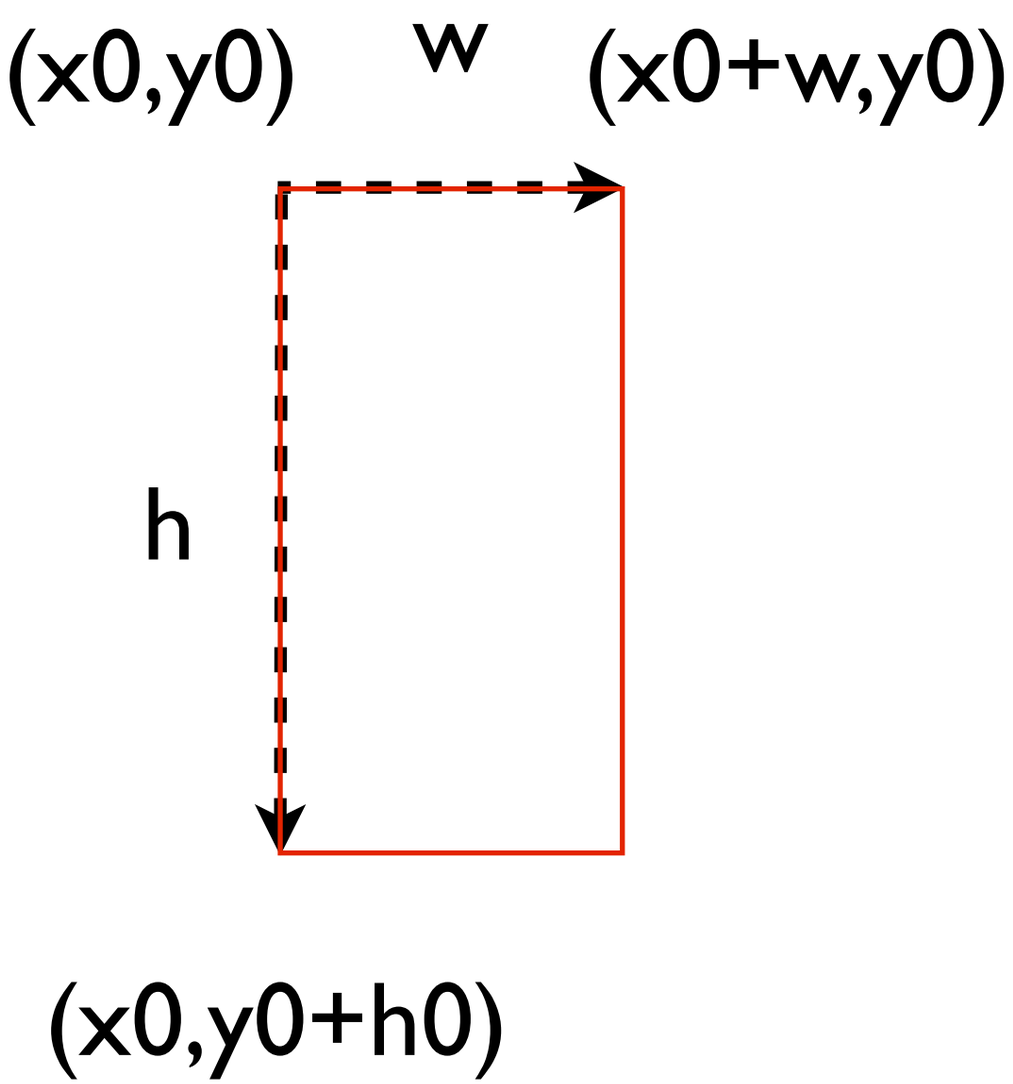
Dessin de lignes

drawLine(x0,y0, x1, y1)
dessine une ligne entre les points (x0,y0) et (x1,y1)



```
for (int i= 0; i < 30; i++) {
    g.drawLine(50 + i* 10, 50,
        1000 + i* 10, 1000);
}
```

Dessin de rectangles)



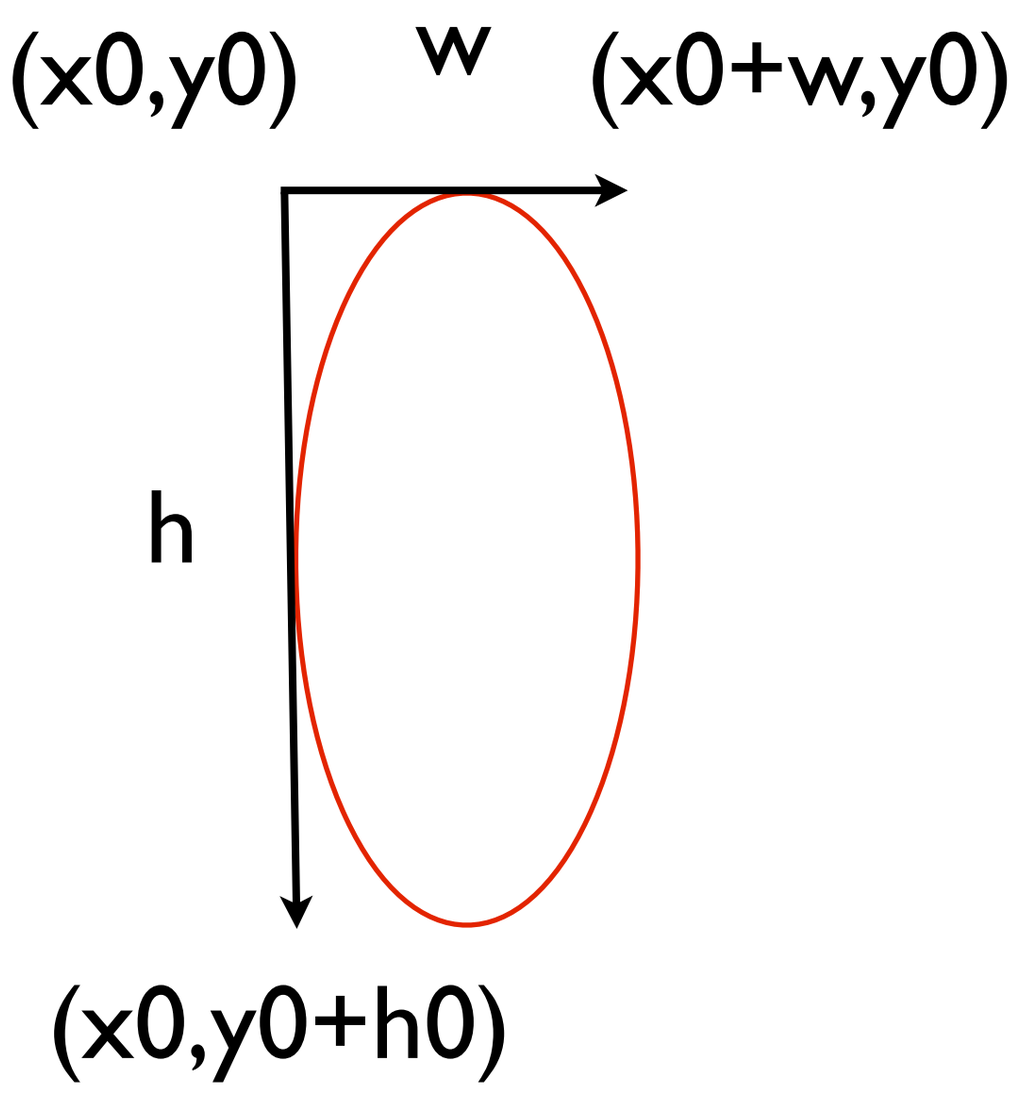
- côtés toujours horizontaux et verticaux
- On donne le point en haut à gauche, la largeur et la hauteur du rectangle

  void **drawRect**(

  int x, int y, int width,
  int height)

- Versions remplies : **fillRect**(...) et **clearRect**(...)

Dessin d'ovales (cercles et ellipses)



- On donne le point en haut à gauche, la largeur et la hauteur du rectangle englobant

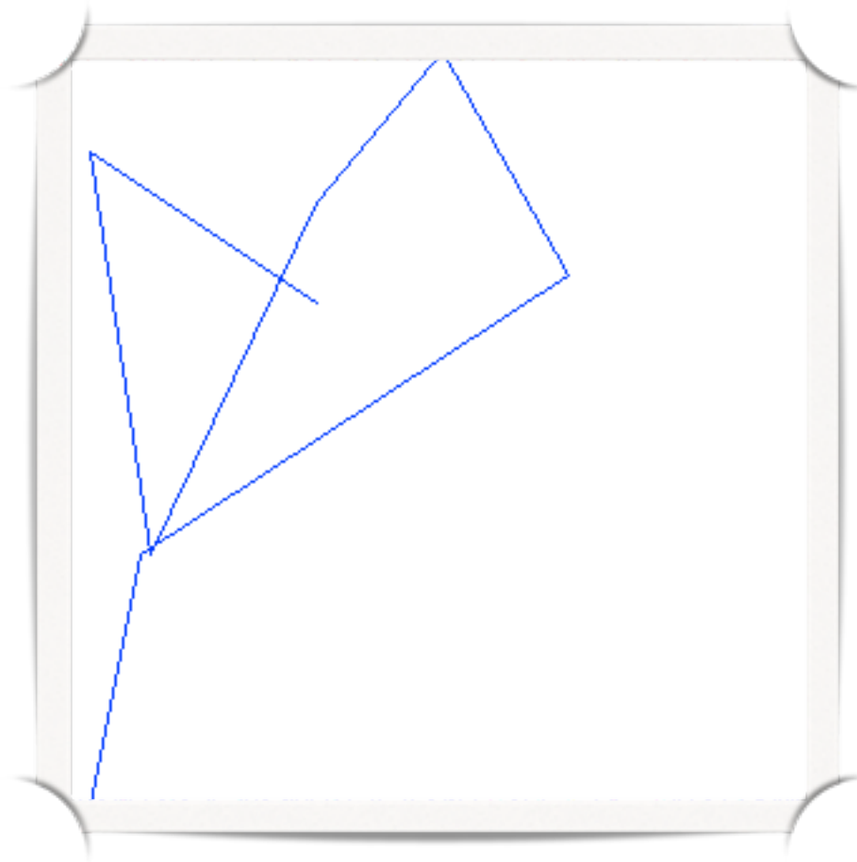
  void **drawOval**(

  int x, int y, int width,
  int height)

- l'ellipse couvre une aire de width+1 de large et height + 1 de haut
- Version remplie : **fillOval**(...)

Poly lignes

- public void drawPolyline(int[] xPoints,
  int[] yPoints, int nPoints)
- xPoint et yPoint sont les tableaux des coordonnées x et y des points
- nPoints est xPoint.length



(Quel artiste !)

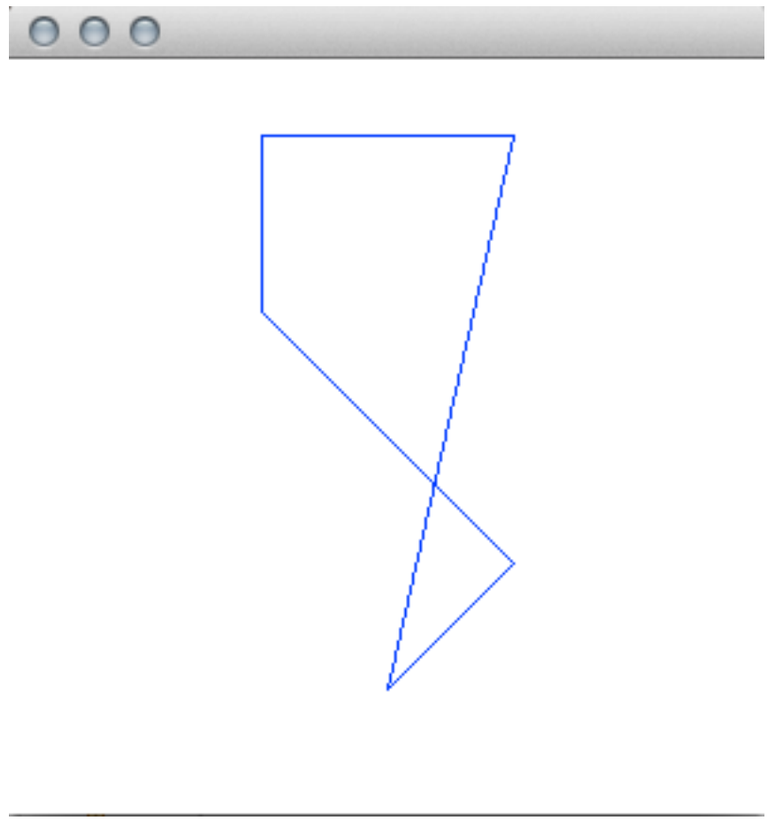
```
g.setColor(Color.BLUE);
int[] xPoints = {100, 10,34,100,150,200, 30,10};
int[] yPoints = {100, 40, 200,60, 0,89,200,300};
g.drawPolyline(xPoints, yPoints, xPoints.length);
```

Polygones

- Classe Polygon
  - ajout de point : addPoint(int x, int y)
- dessin:

  void drawPolygon(Polygon p)

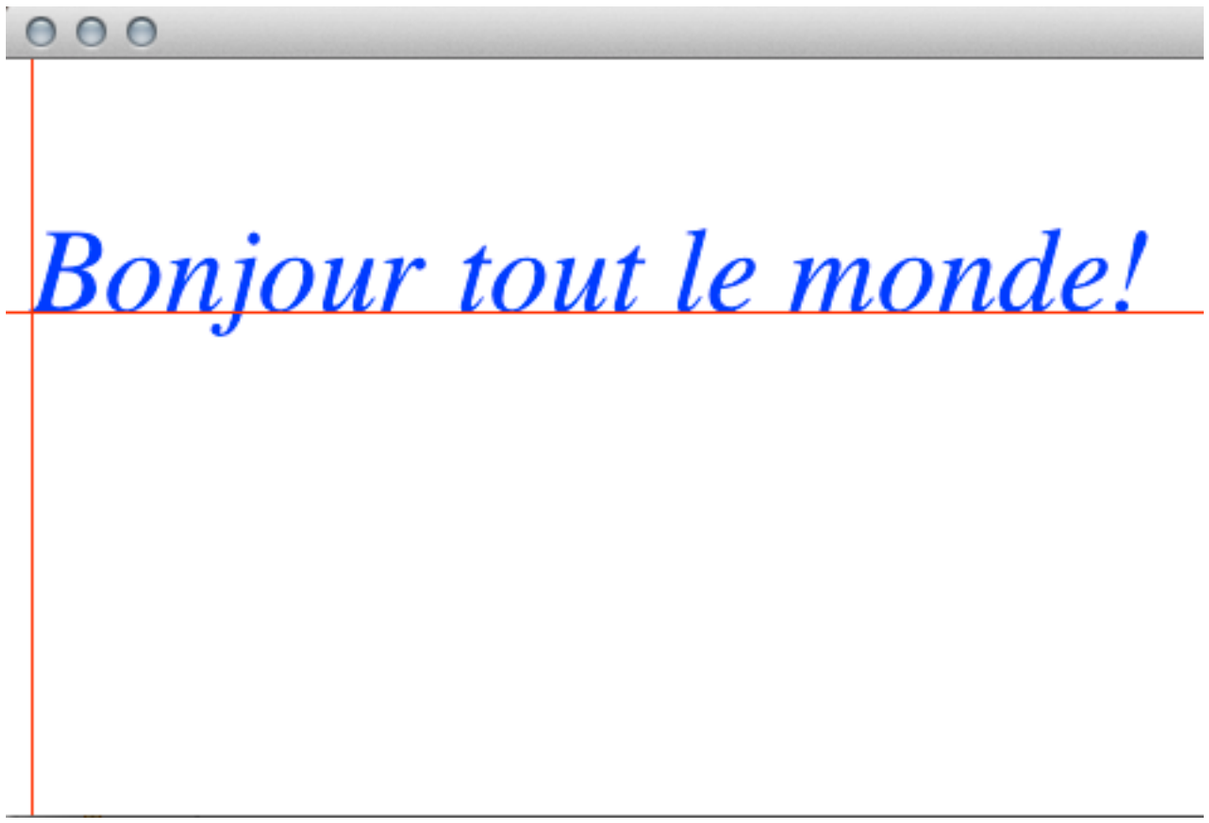
  void fillPolygon(Polygon p)



```
Polygon poly= new Polygon();
poly.addPoint(100, 100);
poly.addPoint(200, 200);
poly.addPoint(150, 250);
poly.addPoint(200, 30);
poly.addPoint(100, 30);
g.drawPolygon(poly);
```

Texte

- Dans la police courante (fixée par setFont)
- affiché par drawString(...)
- coordonnée : ligne de base du texte
- besoins de base. Pour aller plus loin: Graphics2D, TextLayout




```
g.setColor(Color.BLUE);
g.setFont(new Font("Serif", Font.ITALIC, 48));
g.drawString("Bonjour tout le monde!", 10, 100);
g.setColor(Color.RED);
g.drawLine(0, 100, 600, 100); // ligne de base
g.drawLine(10, 0, 10, 300); // ligne verticale x= 10
```

Images bitmap

- Images obtenues à partir de javax.imageio.ImageIO.read(...) (ou autres)
- public abstract boolean
  drawImage(Image img,
  int x, int y, ImageObserver observer)
- l'ImageObserver peut être à null
- Attention : pour simplement afficher une image, on peut directement utiliser un JLabel.

Plus loin : Graphics2D

- L'objet Graphics utilise des coordonnées entières, ne gère pas les transformations (rotation, échelle...), manipule uniquement des ovales et des lignes droites, pas d'épaisseur des lignes...
- Jdk 1.2 : les objets Graphics sont en fait des instances de Graphics2D
- Graphics2D : gère les coordonnées réelles, les courbes, les transformations, les dégradés...
- (à peu près la même expressivité que PDF, SVG...)

L'architecture M/V/C

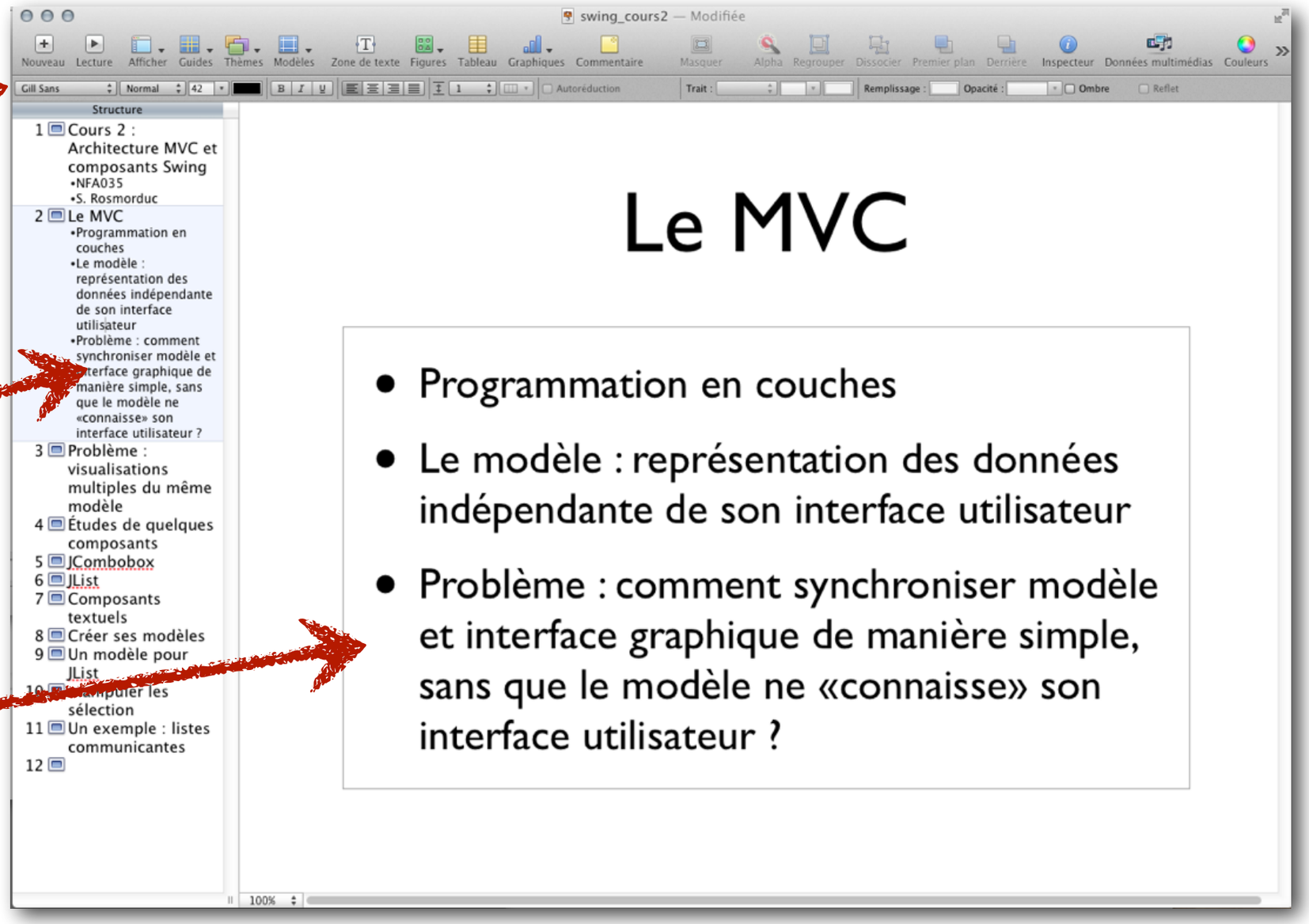
- Programmation en couches
- Le modèle : représentation des données *indépendante* de son interface utilisateur
- Problème : comment synchroniser modèle et interface graphique de manière simple, sans que le modèle ne «connaisse» son interface utilisateur ?

Problème : visualisations multiples du même modèle

Style de l'élément sélectionné

Liste des pages et plan

Page courante



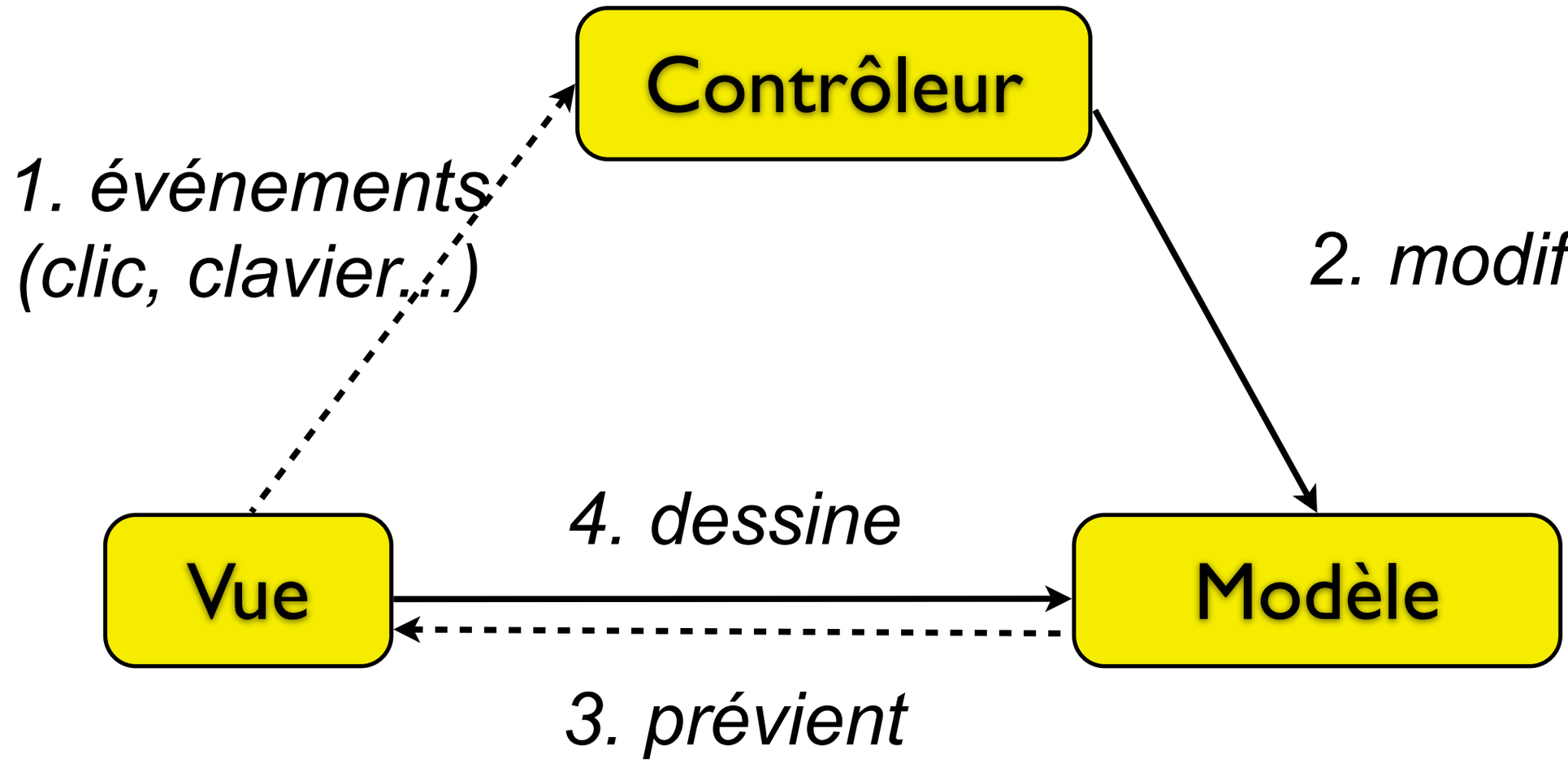
Une solution possible: MVC

- Architecture Modèle-Vue-Contrôleur
- Il existe d'autres solutions (nous verrons Modèle-vue-présentateur pour les applications)
- Trois éléments distincts : Le modèle, La vue, le contrôleur

MVC

- Le modèle : représentation *objet* des données.
  - contient la «logique métier»
- la (ou les) vue(s) : visualisations du modèle. Typiquement, composants swing
- le (ou les) contrôleur(s) : reçoivent les événements provenant des vues, et appellent les méthodes du modèle en conséquence

MVC



⟶ connaît le type

⇢ lien par une interface

(Il y existe des variantes)

Le pattern «observateur/observé»

- pb: le modèle doit prévenir la (ou les) vues qu'il a été modifié
- ... mais on veut que le modèle ne «connaisse» pas les vues
- La classe du modèle ne doit pas dépendre de la classe de la vue (si on supprime la classe de la vue, la classe du modèle doit toujours compiler)
- en pratique: la classe du modèle ne doit jamais contenir de référence à la classe de la vue.

Un exemple simplifié

- élément de saisie pour un entier compris entre 0 et 100
- le modèle : classe ProprietePourcentage, avec une valeur de type int
- la vue : JPourcentage, un JComponent dont le travail sera d'afficher une ProprietePourcentage
- le contrôleur : va recevoir les événements de JPourcentage et permettre de modifier ProprietePourcentage

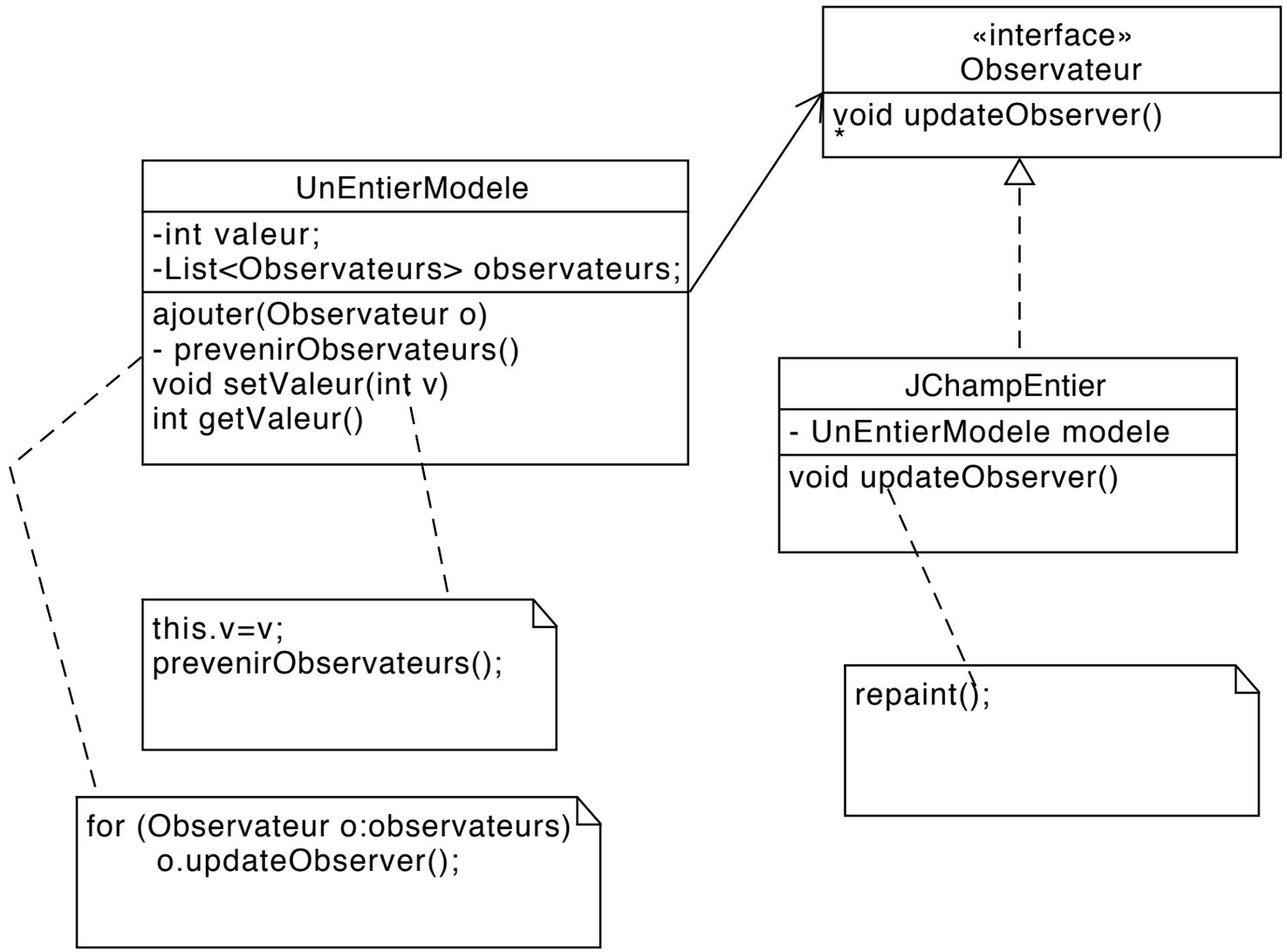
Le problème

```
public class ProprietePourcentage {
   private int valeur;
   private IPourcentage vue;
   ....
   public void setValeur(int v) {valeur=v;}
}
```

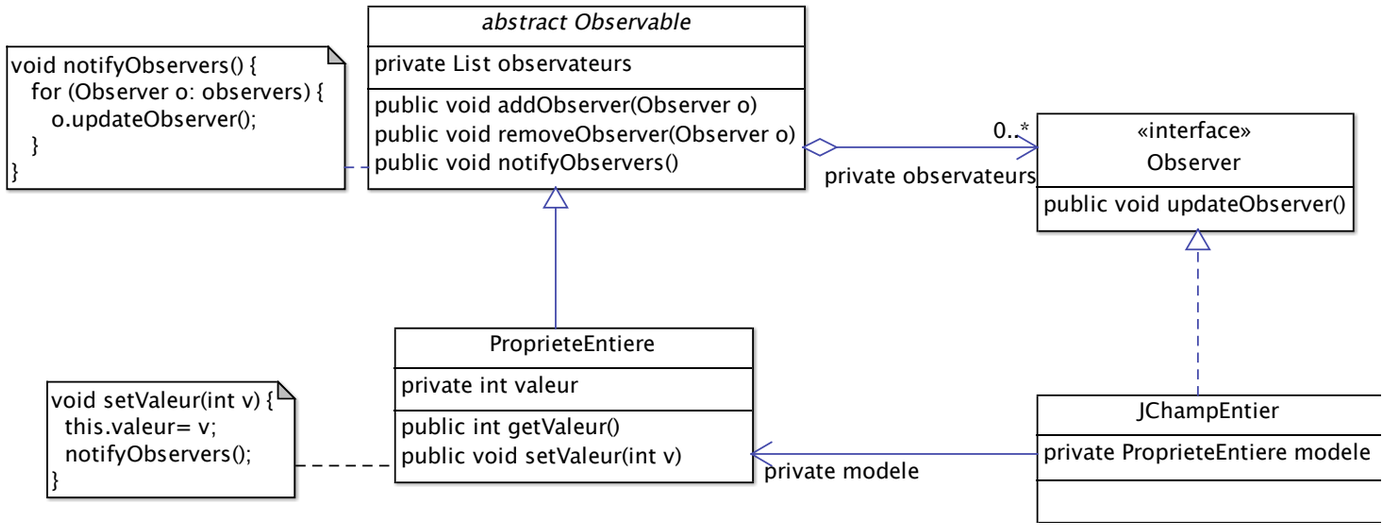
NON !!!

- Le modèle ne doit pas dépendre de la vue
- Plusieurs vues envisageables : champ, curseur graphique, bouton rotatif...
- plusieurs vues simulatanées

La solution: pattern observateur



La solution: pattern observateur

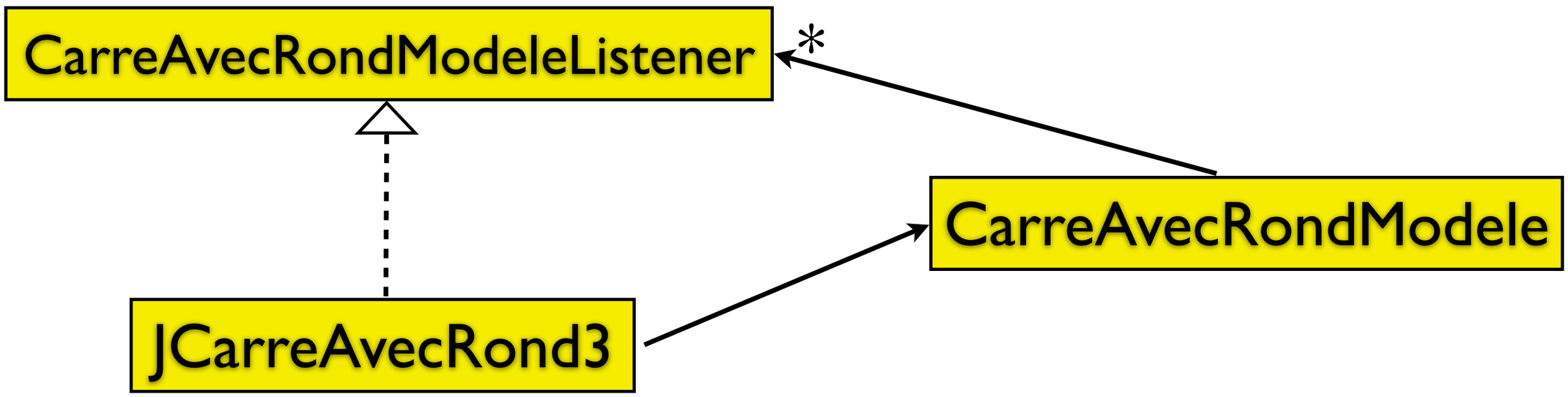


- Quand on modifie le modèle en appelant setValeur...
- le modèle appelle notifyObservers
- *tous les observateurs sont prévenus...*
- et la vue «sait» qu'elle doit se redessiner.

En java...

- Classes Observable, interface Observer existent (mais un peu tordue)
- Surtout: (presque) tous les composants graphiques ont un modèle
- On peut «écouter» les événements qui surviennent sur le modèle
- On peut éventuellement créer son propre modèle
- le plus important: **on sépare le mieux possible le modèle de la vue**

Retour sur le M/V/C



```
public class CarreAvecRondModele {
  private int centreX = 30;
  private int centreY = 30;
  private int rayon = 30;
  private Color couleurCercle = Color.BLUE;
  private List<CarreAvecRondModeleListener> listeners =
          new ArrayList<CarreAvecRondModeleListener>();
    ....
  public void setCentreX(int centreX) {
   if (centreX < rayon)
      return;
   this.centreX = centreX;
   notifyListeners();
}

public void setCouleurCercle(Color couleurCercle) {
 this.couleurCercle = couleurCercle;
 notifyListeners();
}
```

```
public class CarreAvecRondModele {

...

  public void addModeleListener(
   CarreAvecRondModeleListener listener) {
   listeners.add(listener);
  }

  public void removeModeleListener(
   CarreAvecRondModeleListener listener) {
   listeners.remove(listener);
  }

  /**
   * Prévient les listeners d'une modification.
   * @param event
   */
  private void notifyListeners() {
   for (CarreAvecRondModeleListener l: listeners) {
      l.modeleModifie();
   }
  }
}
```

```
public interface CarreAvecRondModeleListener {
  void modeleModifie();
}
```

```
public class JCarreAvecRond3 extends JPanel
                              implements  CarreAvecRondModeleListener{
  private CarreAvecRondModele modele;

   /**
    * Appelée quand le modèle est modifié.
    */
   @Override
   public void modeleModifie() {
         repaint();
         revalidate();
   }

....
```

note: en passant des informations au listener (un événement) on pourrait avoir un comportement plus fin (ne pas appeler revalidate() quand ça n'est pas nécessaire)

```
public class JCarreAvecRond3 extends JPanel {

   ...

   public JCarreAvecRond3() {
    setModele(new CarreAvecRondModele());
    setBackground(Color.WHITE);
   }

   public CarreAvecRondModele getModele() {
    return modele;
   }

   public void setModele(CarreAvecRondModele modele) {
      // Si on observe déjà un ancien modèle, on
      // se désabonne...
      if (this.modele != null) {
         this.modele.removeModeleListener(this);
      }
      // on s'inscrit auprès du nouveau modèle..
      this.modele= modele;
      modele.addModeleListener(this);
   }
...
```

```
public class JCarreAvecRond3 extends JPanel {

  public Dimension getPreferredSize() {
   int w = Math.max(300, modele.getCentreX() + modele.getRayon());
   int h = Math.max(300, modele.getCentreY() + modele.getRayon());
   return new Dimension(w, h);
  }

  protected void paintComponent(Graphics g) {
   super.paintComponent(g); // Laisser cette méthode ici.
   g.setColor(modele.getCouleurCercle());
   g.fillOval(modele.getLeft(), modele.getTop(),
       modele.getDiametre(), modele.getDiametre());
  }

  public void setCouleurCercle(Color couleurCercle) {
   modele.setCouleurCercle(couleurCercle);
  }

  public void setCentreX(int centreX) {
   modele.setCentreX(centreX);
  }
  ....
}
```

Gestion de la souris

- Plusieurs types de listeners :
  - MouseListener : clics, pressions, entrée et sortie de la souris de dessus un composant
  - MouseMotionListener : mouvements de la souris
- et d'autres (molette par exemple)

MouseListener

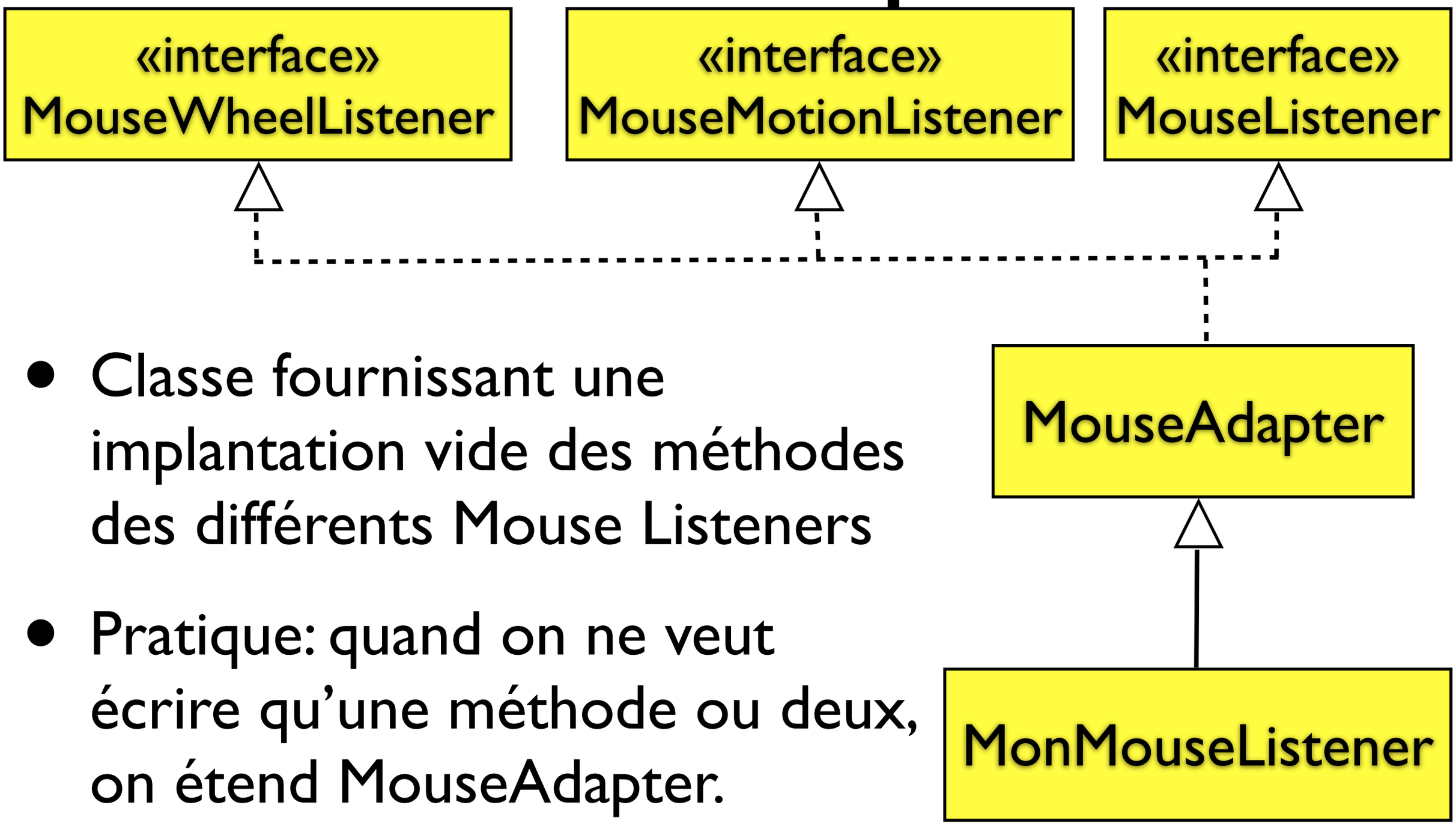
Méthodes à écrire:

- void **mouseClicked**(MouseEvent e) : on a cliqué avec la souris
- void **mouseEntered**(MouseEvent e) : la souris arrive sur le composant
- void **mouseExited**(MouseEvent e) : la souris quitte les limites du composant
- void **mousePressed**(MouseEvent e) : on presse le bouton de la souris au dessus du composant
- void **mouseReleased**(MouseEvent e) : on le relâche

MouseMotionListener

- void mouseDragged(MouseEvent e) : appelée quand la souris est déplacée avec un bouton appuyé «on tire sur la souris»
- void mouseMoved(MouseEvent e) : appelée quand la souris est déplacée sans presser un bouton

MouseAdapter



- Classe fournissant une implantation vide des méthodes des différents Mouse Listeners
- Pratique: quand on ne veut écrire qu'une méthode ou deux, on étend MouseAdapter.

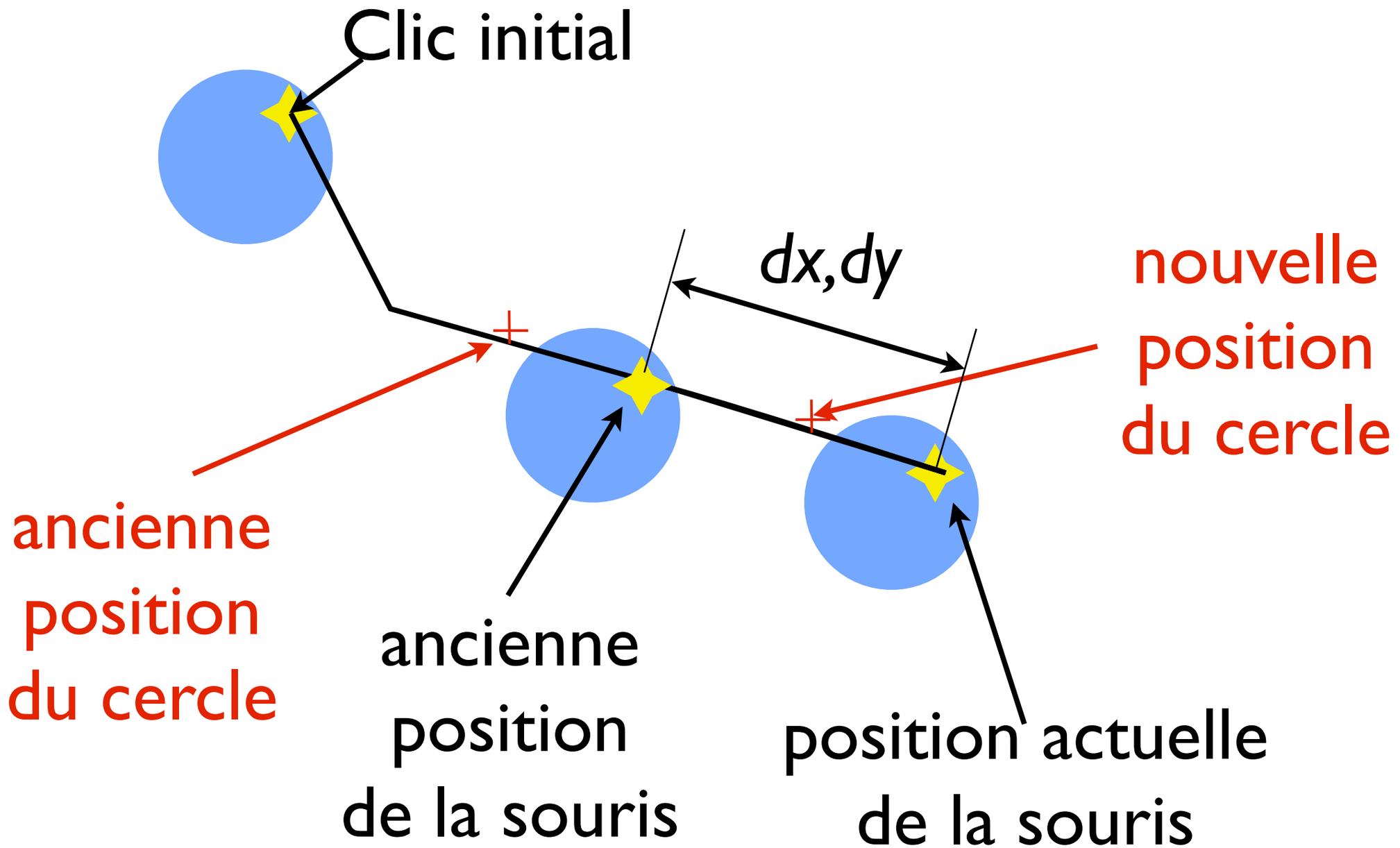
MouseEvent

- Passé à toutes les méthodes. Connaît:
- les coordonnées de la souris / au composant: ev.getPoint()
- le nombre de clics : ev.getClickCount()
- le bouton pressé ev.getButton() (un int)
- sait si des touches sont enfoncées : ev.isAltDown(), ev.isShiftDown(), ev.isControlDown()

Complément (hors programme): déplacer une forme à l'écran...

- Petite variante du programme précédent avec plusieurs cercles
- On veut pouvoir déplacer un cercle...
  - quand on **presse la souris :** on cherche quel cercle est pressé, et on le stocke dans une variable d'instance
  - quand on **tire** (drag) la souris : si un cercle est sélectionné (étape précédente), on le déplace
  - quand on **relâche la souris** : on met le cercle sélectionné à null.
- On crée deux classes:
  - le composant graphique
  - les cercles, dans lesquels on met un peu d'intelligence

Déplacement d'un cercle



```
public class Cercle {
  private CerclesEtSouris proprietaire;
  private int x, y;
  private int rayon;
  private Color color= Color.BLUE;
  ...
  public void setX(int x) {
   this.x = x;
   proprietaire.repaint();
  }
  ...
  public boolean contientPoint(Point p) {
   // un point est dans le disque si sa distance au centre est
   // inférieure au rayon...
   return ((p.x - x)*(p.x -x) + (p.y -y)*(p.y-y)) < rayon* rayon;
  }
  ...
  public void draw(Graphics g) {
   g.setColor(color);
   g.fillOval(x- rayon, y- rayon, 2*rayon, 2*rayon);
  }
}
```

On prévient le composant quand on est modifié...

On sait se dessiner sur le composant

```
public class CerclesEtSouris extends JPanel {
  private ArrayList<Cercle> cercles;

  public CerclesEtSouris() {
   setBackground(Color.WHITE);
   // Initialisation des données
   cercles= new ArrayList<Cercle>();
   cercles.add(new Cercle(this, 100, 100, 40));
   Cercle autre= new Cercle(this, 200, 200, 30);
   autre.setColor(Color.RED);
   cercles.add(autre);
   // On crée le listener :
   DeplacementListener listener= new DeplacementListener();
   // Ne pas oublier de l'ajouter comme listener
   addMouseListener(listener);
   addMouseMotionListener(listener);
  }

  protected void paintComponent(Graphics g) {
   super.paintComponent(g);
   Color old= g.getColor();
   for (Cercle c: cercles) {
      c.draw(g);
   }
   g.setColor(old);
  }
    ...
```

```
public class CerclesEtSouris extends JPanel {
  ...
  private class DeplacementListener extends MouseAdapter {
   // Le cercle actuellement sélectionné.
   private Cercle cercleSelectionne= null;
   // Les derniers x et y enregistrés
   private int lastX, lastY;

   public void mousePressed(MouseEvent e) {
      cercleSelectionne= null;
      // où clique-t-on ?
      Point p= e.getPoint();
      for (Cercle c: cercles) {
         if (c.contientPoint(p)) {
            cercleSelectionne= c;
         }
      }
      lastX= p.x;
      lastY= p.y;
}
```

```
public class CerclesEtSouris extends JPanel {
   ...
   private class DeplacementListener extends MouseAdapter {

   ...
    public void mouseReleased(MouseEvent e) {
       cercleSelectionne= null;
    }

    public void mouseDragged(MouseEvent e) {
       if (cercleSelectionne!= null) {
          Point p= e.getPoint();
          int dx= p.x - lastX;
          int dy= p.y - lastY;
          cercleSelectionne.setX(cercleSelectionne.getX() + dx);
          cercleSelectionne.setY(cercleSelectionne.getY() + dy);
          lastX= p.x;
          lastY= p.y;
       }
    }
}
```