



# Interface Graphique en Java 1.6

## Gestionnaires de géométrie

Sébastien Paumier



# Qu'est-ce que c'est ?

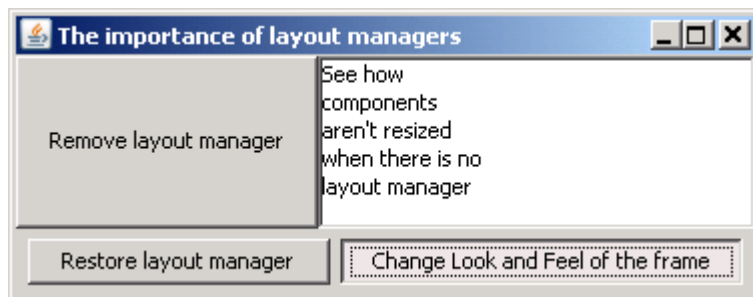
---

- chaque container contenant des composants doit savoir comment les placer
- **LayoutManager**=objet qui calcule les tailles et positions des composants enfants
- tient compte de contraintes:
  - liées aux tailles (préférée, min et max) des composants
  - et/ou de la nature du **LayoutManager**

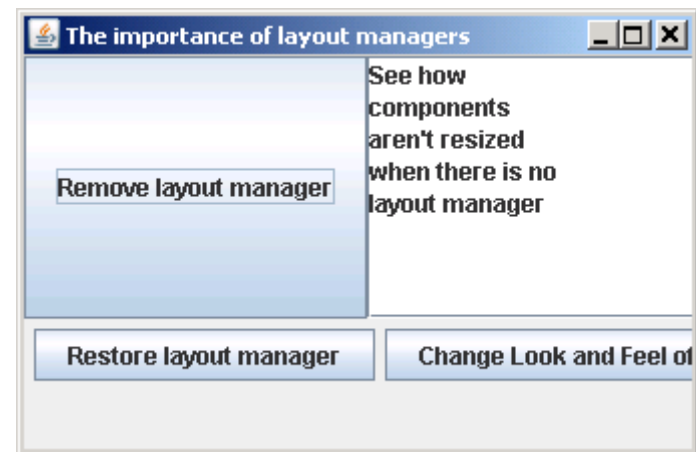
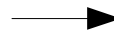


# À quoi ça sert ?

- on n'écrit **JAMAIS** une application où l'on fixe les composants, car:
  - la fenêtre peut être redimensionnée
  - l'espace occupé par un composant peut changer suivant le Look and Feel



bon



plus bon



# Comment ça marche ?

---

- on définit le **LayoutManager** d'un container avec **setLayout/getLayout**
- on peut l'enlever avec **setLayout(null)**
- le container en question est presque toujours un **JPanel**, avec un **FlowLayout** par défaut
- on peut passer un **LayoutManager** au constructeur:

```
JPanel p=new JPanel(new BorderLayout());
```



# Comment ça marche ?

---

- les **LayoutManager** utilisent les différentes tailles des composants
- on peut les fixer:
  - soit avec **set...Size**
  - soit en redéfinissant la méthode **get...Size**

```
private static JComponent create(final Color c,int x,int y) {
    final Dimension d=new Dimension(x,y);
    return new JComponent() {
        @Override public Dimension getPreferredSize() {
            return d;
        }

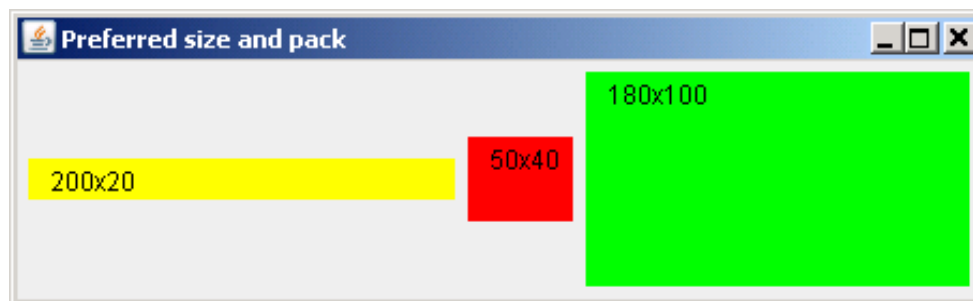
        @Override protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            g.setColor(c);
            g.fillRect(0,0,getWidth()-1,getHeight()-1);
            g.setColor(Color.BLACK);
            g.drawString(d.width+"x"+d.height,10,15);
        }
    };
}
```



# Comment ça marche ?

---

- les **JComponent** ont une taille nulle par défaut, il faut donc leur donner explicitement une taille (préférée ou non)
- sur une **JFrame**, la méthode **pack()** demande à chaque composant sa taille préférée, et s'ajuste en fonction du résultat





# Placement des composants

---

- chaque composant possède:
  - une position par rapport au parent:  
`set/getLocation`, `set/getX`, `set/getY`
  - une taille: `set/getWidth`, `set/getHeight`,  
`set/getSize`
- la zone couverte peut aussi être manipulée directement avec:
  - `set/getBounds`



# Les LayoutManager

---

- **LayoutManager** sans contrainte:
  - **FlowLayout**
  - **GridLayout**
  - **BoxLayout**
- avec contraintes: **LayoutManager2**, qui hérite de **LayoutManager**
  - **BorderLayout**
  - **GridBagLayout**
  - **GroupLayout**



# Le FlowLayout

---

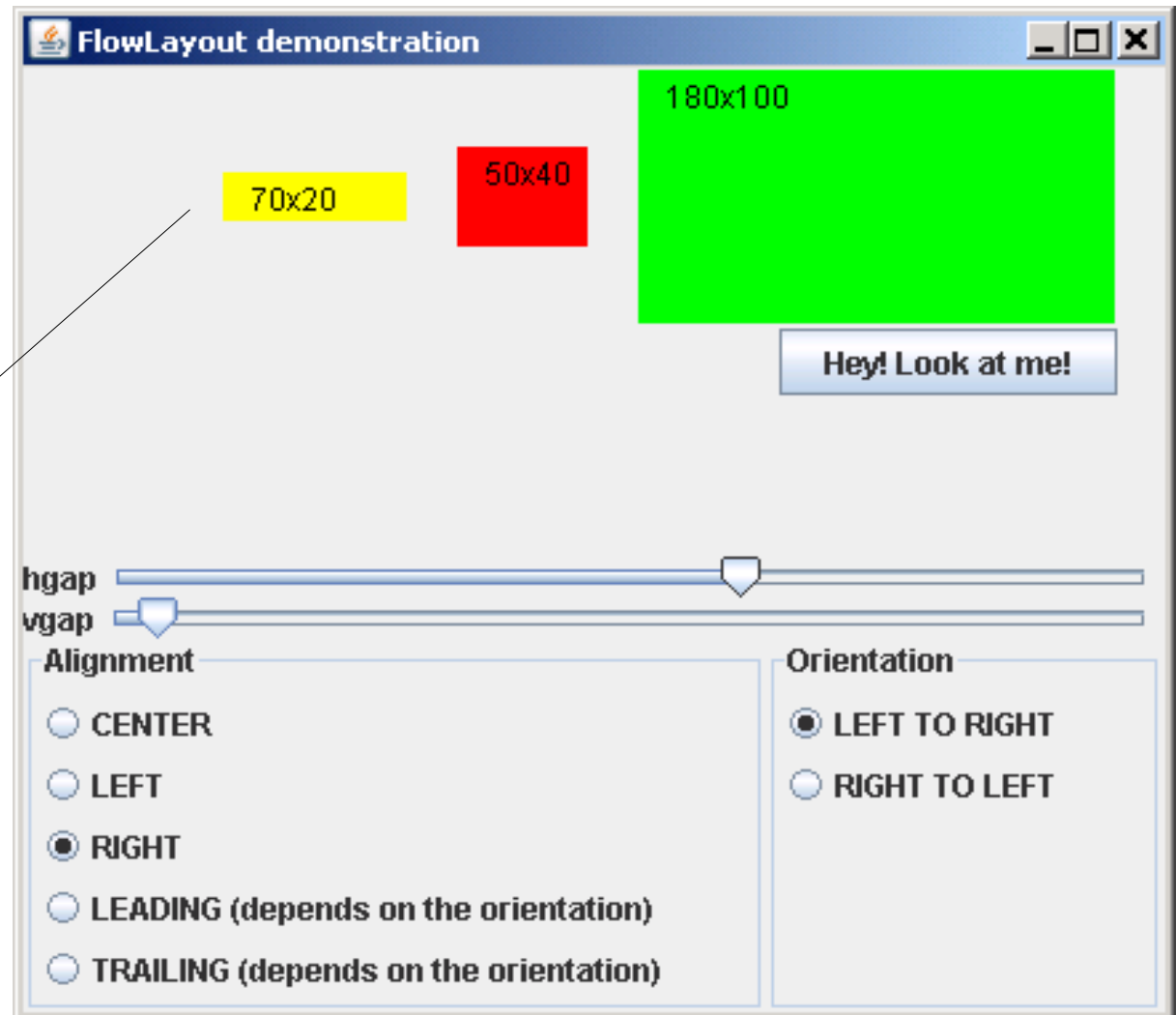
- layout manager par défaut des `JPanel`
- affiche les composants à leur taille préférée, "de la gauche vers la droite", et va à la ligne si nécessaire
- ordre=ordre d'ajout dans le container
- `FlowLayout(int align,int hgap,int vgap)`
  - `align`=alignement sur chaque ligne
  - `hgap/vgap`: espace entre les composants



# Le FlowLayout

- exemple:

les lignes  
partent du  
haut du  
container





# Modification du layout

---

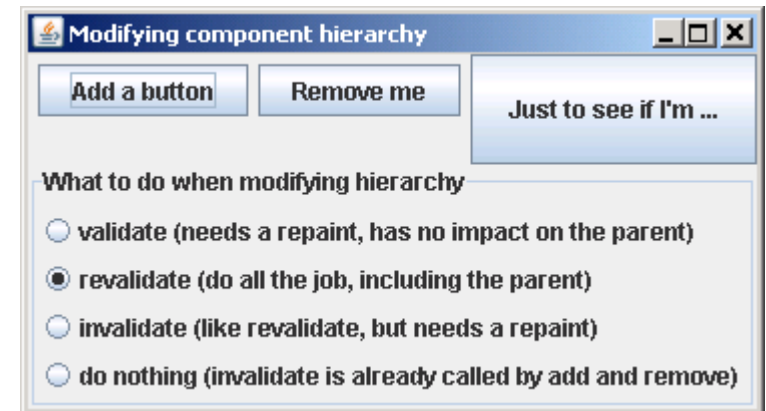
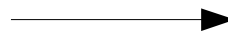
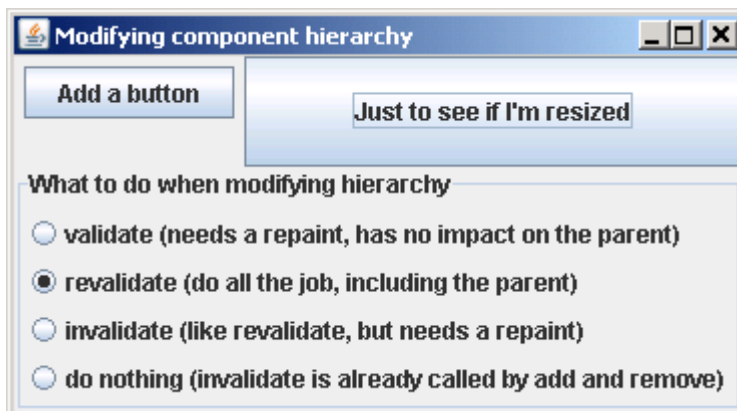
- toute modification du layout doit être prise en compte:
  - `myLayout.layoutContainer`

```
for (int i=0;i<strings.length;i++) {
    final JRadioButton b=new JRadioButton(strings[i],i==0);
    final int val=values[i];
    b.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            if (b.isSelected()) {
                layout.setAlignment(val);
                layout.layoutContainer(p);
                /* or p.revalidate(); */
            }
        }
    });
    alignment.add(b);
    group.add(b);
}
```



# Modification de la hiérarchie

- tout ajout ou suppression de composant doit être pris en compte:
  - **validate+repaint**: si on veut réorganiser l'espace dont on disposait déjà
  - **revalidate**: si on veut changer sa zone et répercuter les changements sur les parents





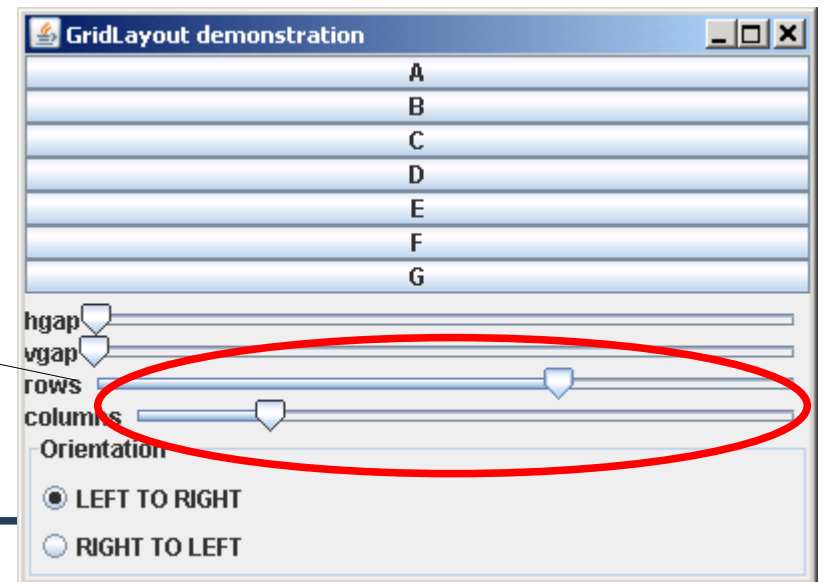
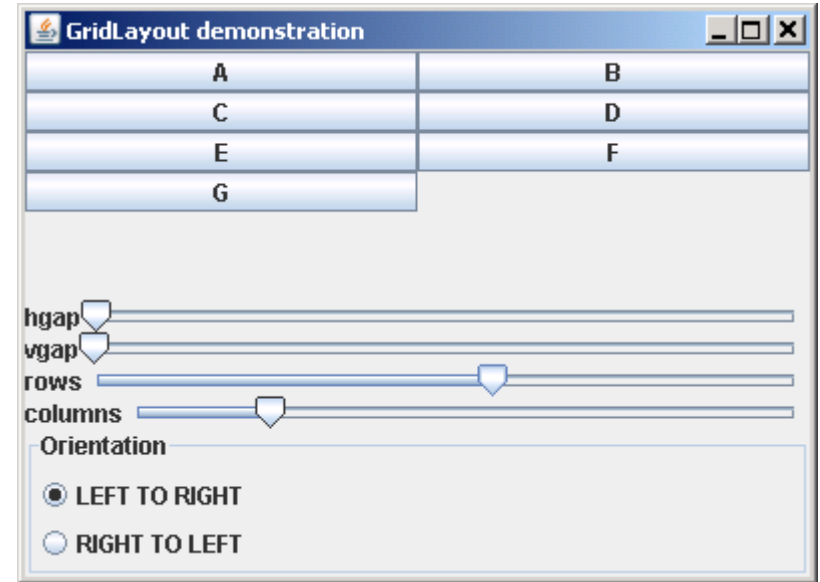
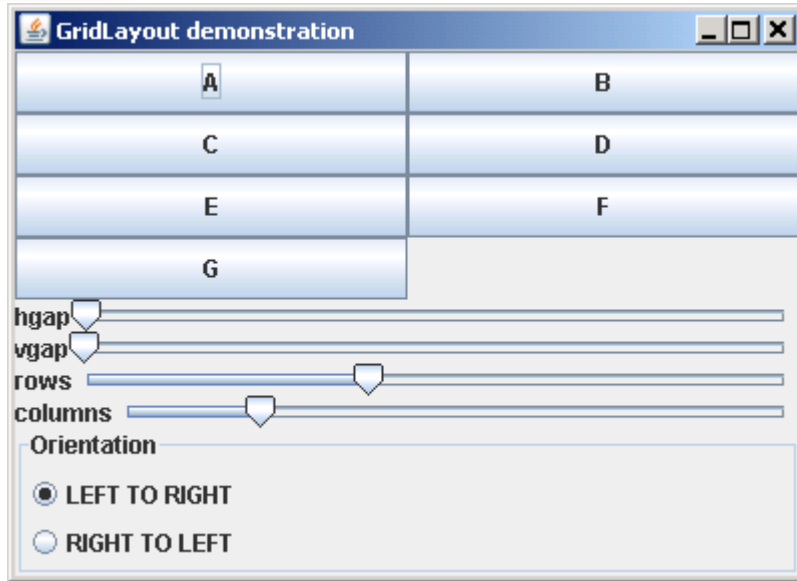
# Le GridLayout

---

- dispose les composants sur une grille, dans leur ordre d'ajout au container
- `GridLayout(int rows, int cols, int hgap, int vgap)`
  - `rows/columns`=lignes/colonnes
  - `hgap/vgap`: espace entre les composants
- tous les composants sont affichés (en recalculant `rows` et `columns` si nécessaire), et ils ont tous la même taille
- s'il y a moins de composants que de cases, le layout s'adapte



# Le GridLayout



comme on a demandé explicitement 7 lignes, le nombre de colonnes est recalculé car le **GridLayout** refuse de laisser plus de la moitié des cases vides



# Le BorderLayout

---

- affiche les composants en ligne ou en colonne, selon leur taille préférée
- ordre=ordre d'ajout dans le container
- on doit créer le container **AVANT** le **BoxLayout**, car celui-ci en a besoin dans son constructeur:

```
/* null avoids the creation of the default FlowLayout */  
final JPanel p=new JPanel(null);  
p.setLayout(new BorderLayout(p,BoxLayout.Y_AXIS));
```

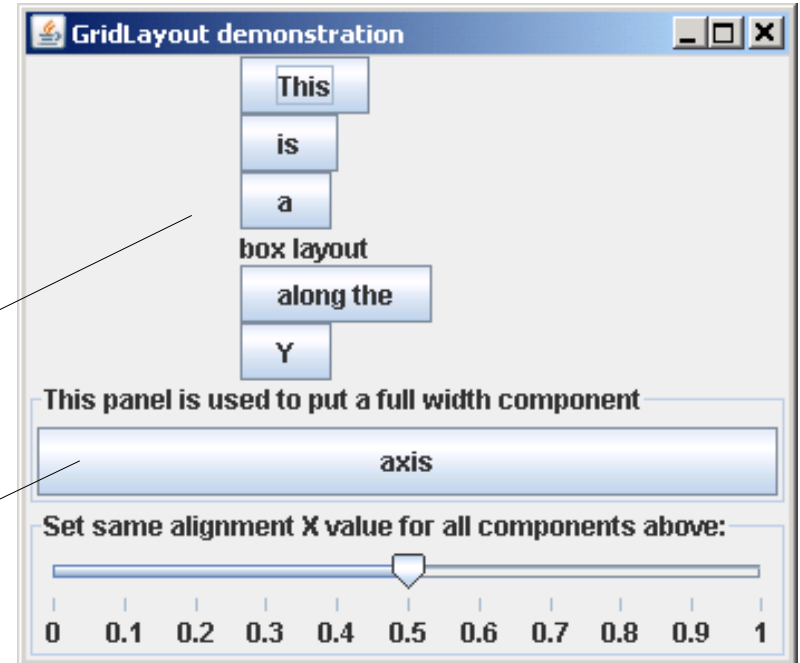


# Le BorderLayout

- par défaut, l'alignement des composants obéit à des règles complexes:

pas étiré

étiré

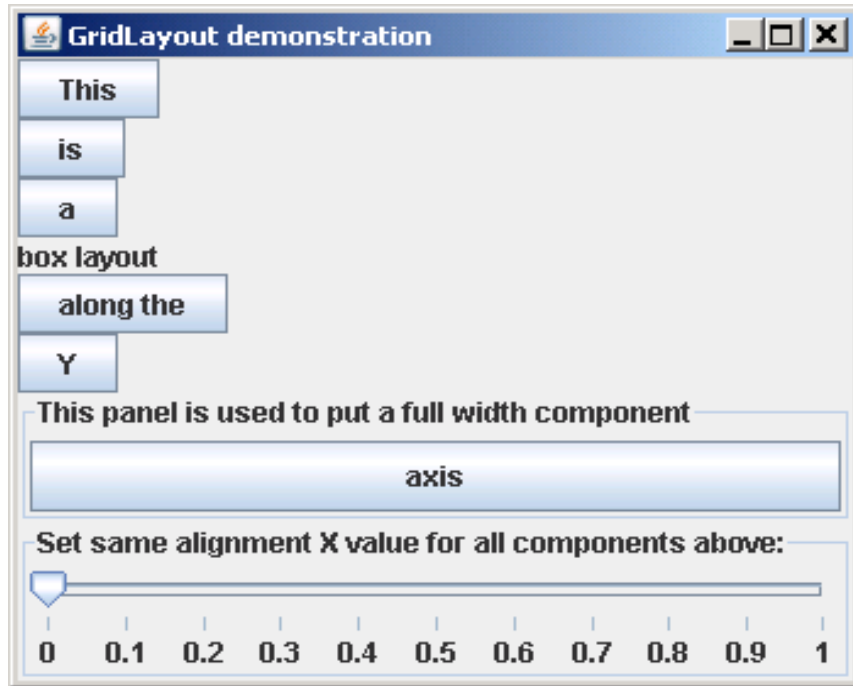


- le plus simple est de donner le même alignement à tous les composants:

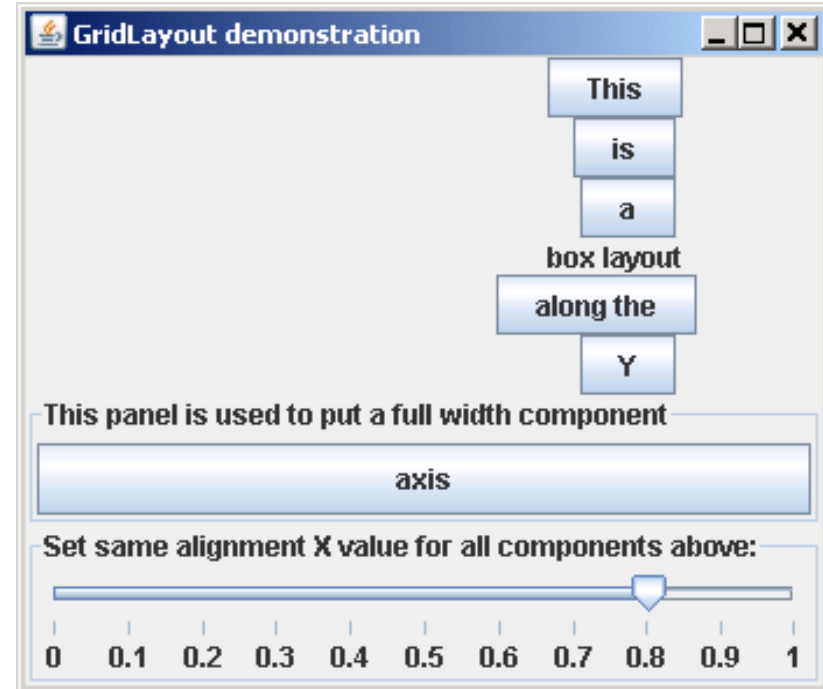
```
for (int i=0;i<p.getComponentCount();i++) {  
    JComponent c=(JComponent) p.getComponent(i);  
    /* 0f means left alignement  
     * 1f means right alignement */  
    c.setAlignmentX(0f);  
}
```



# Le BorderLayout



alignement à gauche de  
tous les composants,  
étirés ou non



alignement à 80% de la  
largeur



# Le BorderLayout

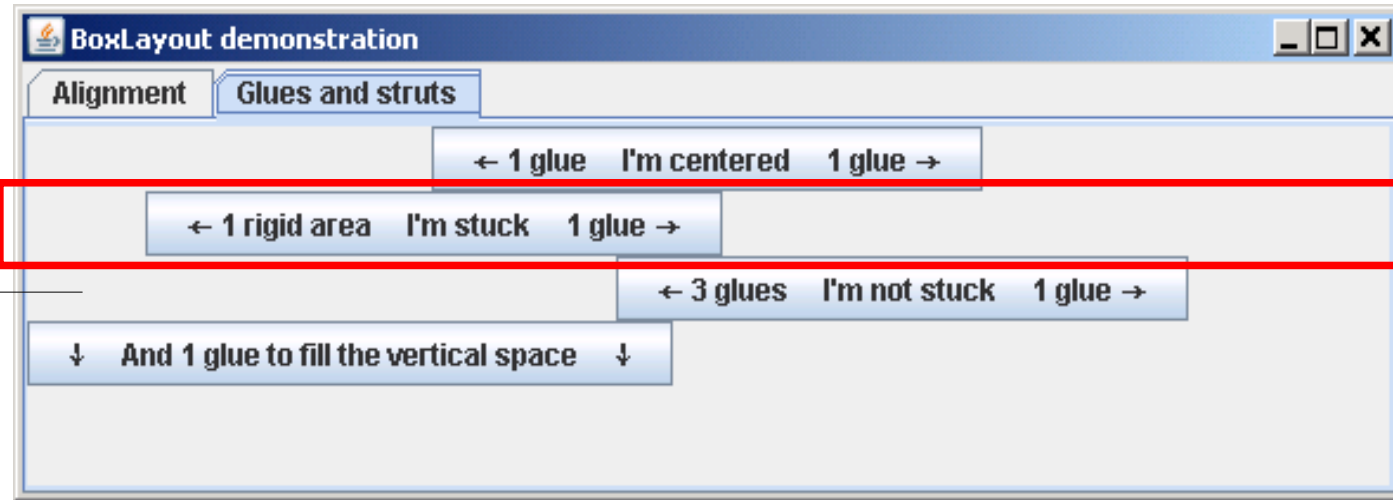
---

- dans un container avec un BorderLayout, on peut ajouter:
  - des *glues*: ressorts invisibles qui se partagent l'espace disponible
  - des *struts*: zones de largeur/hauteur fixe, qui se partagent en hauteur/largeur l'espace disponible
  - des *rigid area*: zones de tailles fixe



# Le BorderLayout

les glues se répartitionnent l'espace disponible



```
JPanel gluePanel2=new JPanel(null);
gluePanel2.setLayout(new BorderLayout(gluePanel2,BoxLayout.X_AXIS));
/* We must use a rigid area and not a strut, because a strut does
 * take vertical space when available */
gluePanel2.add(Box.createRigidArea(new Dimension(50,0)));
gluePanel2.add(
    new JButton("\u2190 1 rigid area      I'm stuck      1 glue \u2192"));
gluePanel2.add(Box.createHorizontalGlue());
gluePanel.add(gluePanel2);
```



# Les LayoutManager2

---

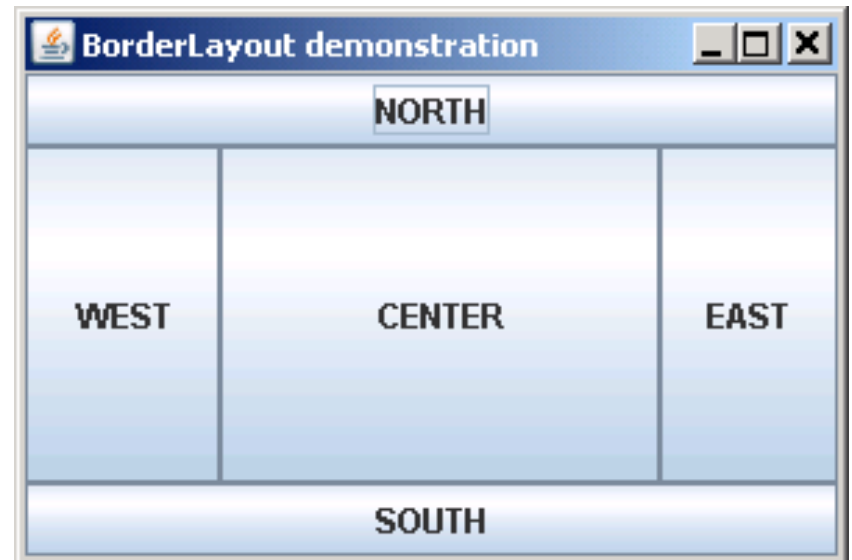
- ajout de composant avec une contrainte de placement:
  - `add(java.awt.Component, Object constraint)`
- la nature de la contrainte dépend du `LayoutManager2` utilisé
- le `add` sans contrainte de `Container` est en fait un `add` avec une contrainte `null`:

```
public Component add(Component comp) {  
    addImpl(comp, null, -1);  
    return comp;  
}
```



# Le BorderLayout

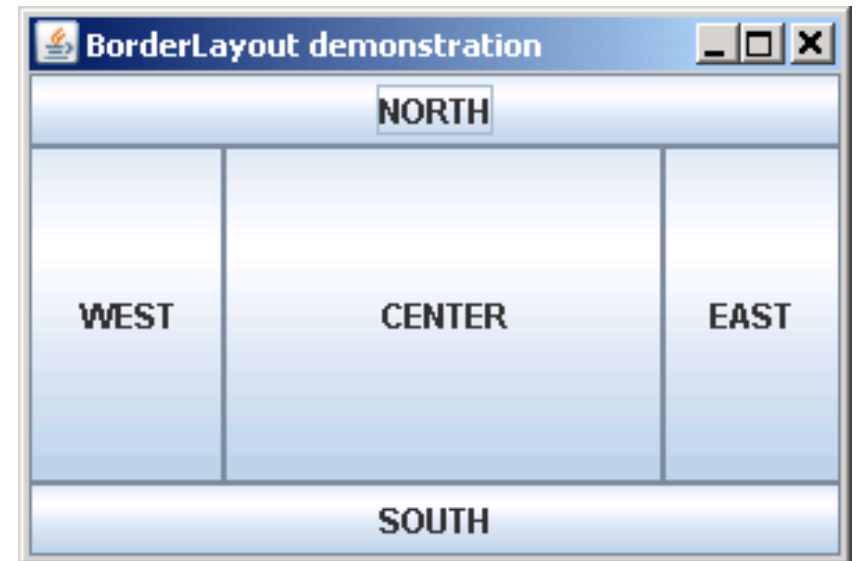
- layout par défaut du `ContentPane` d'une `JFrame`
- 5 positions possibles:
  - `BorderLayout.NORTH`
  - `BorderLayout.SOUTH`
  - `BorderLayout.EAST`
  - `BorderLayout.WEST`
  - `BorderLayout.CENTER` (par défaut)
- 1 seul composant par zone !





# Le BorderLayout

- répartition de l'espace:
  - nord/sud: hauteur préférée, largeur max
  - est/ouest: hauteur max de l'espace restant, largeur préférée
  - centre: tout ce qui reste
- l'ordre d'ajout dans le container n'a pas d'importance





# Le GridBagLayout

---

- très puissant, mais un peu plus complexe
- positionnement des composants selon une grille, dans l'ordre d'ajout, et en tenant compte de contraintes sur:
  - la zone occupée par un composant
  - le placement d'un composant dans sa zone
  - le comportement de la zone en cas de redimensionnement



# Le GridBagLayout

---

- les contraintes sont exprimées par une instance de **GridBagConstraints**
- comme **add** copie la contrainte, on peut réutiliser le même objet pour plusieurs contraintes, en ne modifiant que le nécessaire:

```
GridBagConstraints gbc=new GridBagConstraints();
gbc.gridwidth=3;
gbc.fill=GridBagConstraints.BOTH;
gbc.weightx=1f;
gbc.weighty=1f;
p.add(new JButton("You can't see it but I occupy 3 cells"),gbc);
gbc.gridwidth=GridBagConstraints.REMAINDER;
p.add(new JButton("I'm a button that occupy 1 cell"),gbc);
```

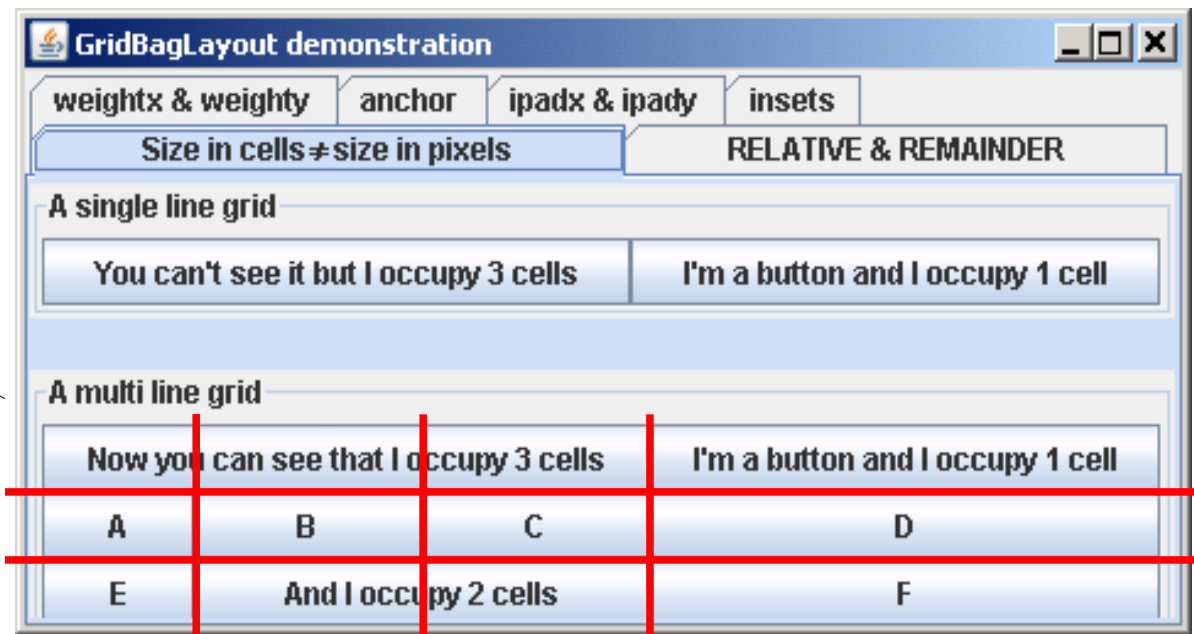


# Le GridBagLayout

- zone occupée par un composant=portion rectangulaire de la grille
- les cases n'ont pas toutes la même taille !
- taille en cellules  $\neq$  taille en pixels

largeur de  
colonne homogène

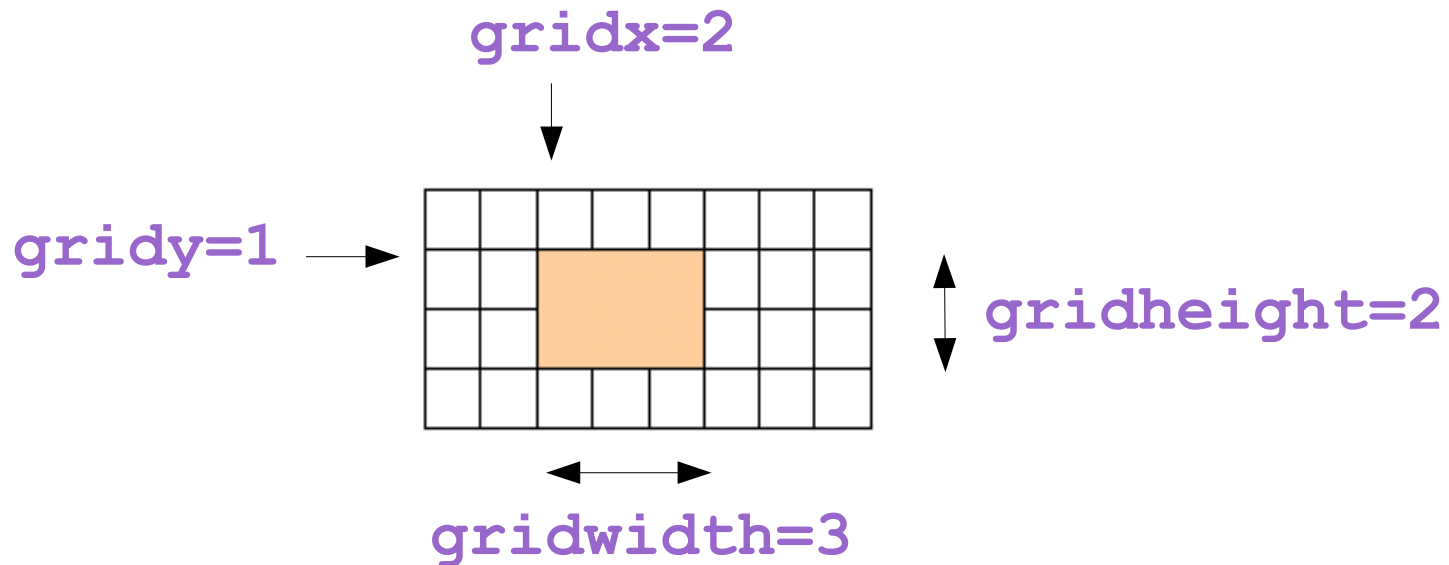
hauteur de ligne  
homogène





# Le GridBagLayout

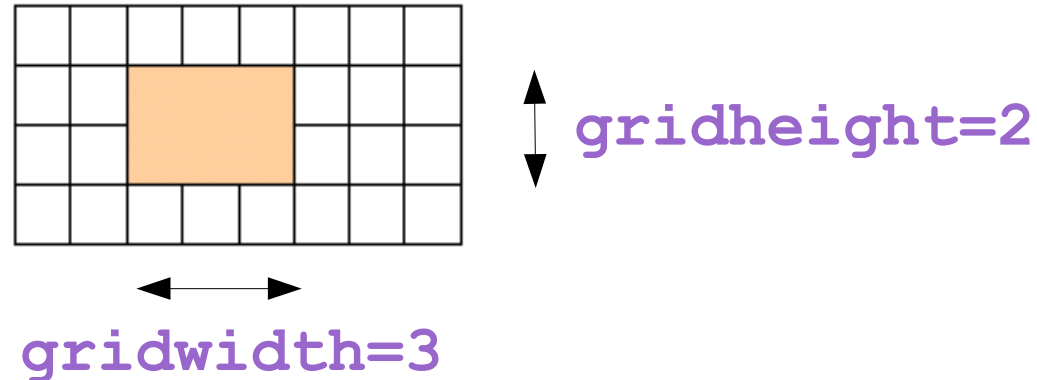
- définition de la zone:



- par défaut, **gridx=gridy=RELATIVE**:
  - la zone est à droite du dernier composant placé, et immédiatement en dessous de la dernière ligne de composants



# Le GridBagLayout



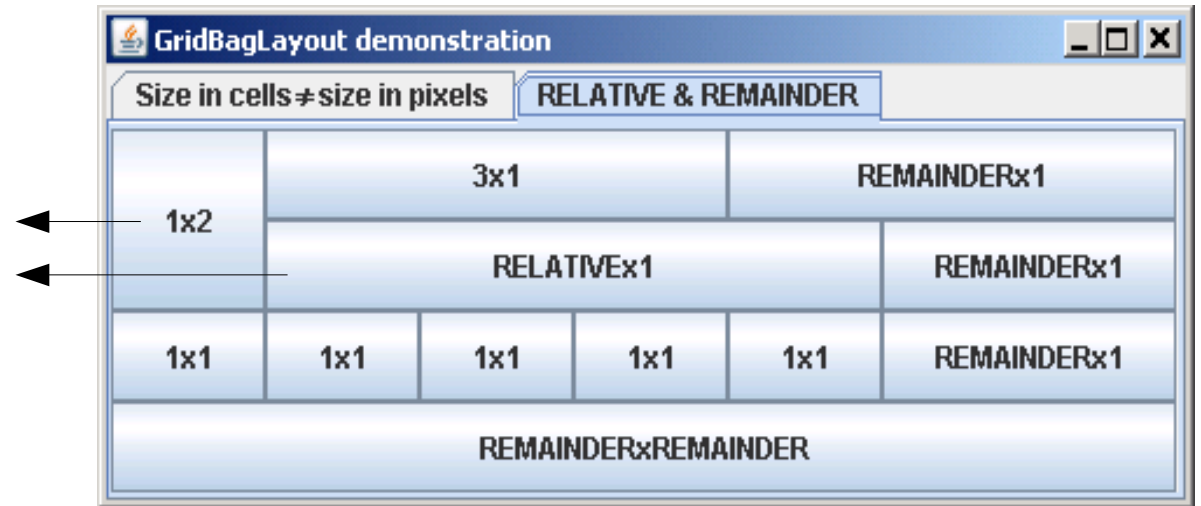
- par défaut, `gridwidth=gridheight=1`:
- si `gridwidth/gridheight=RELATIVE`, on occupe toute la largeur/hauteur jusqu'au prochain composant fixe
- si `gridwidth/gridheight=REMAINDER`, la zone est la dernière de la ligne/colonne



# Le GridBagLayout

- on ajoute les composants ligne par ligne, et le **GridBagLayout** se débrouille pour combiner les contraintes:

la hauteur 2 est utilisée pour savoir que la 2<sup>ème</sup> ligne démarre à X=1 et non pas X=0



NOTE: dans cet exemple, seuls **gridwidth** et **gridheight** ont été modifiés



# Le GridBagLayout

- chaque composant peut demander de l'espace en plus au moyen de poids **double**: **weightx** et **weighty** (0 par défaut)

c'est le max de la ligne/colonne qui domine

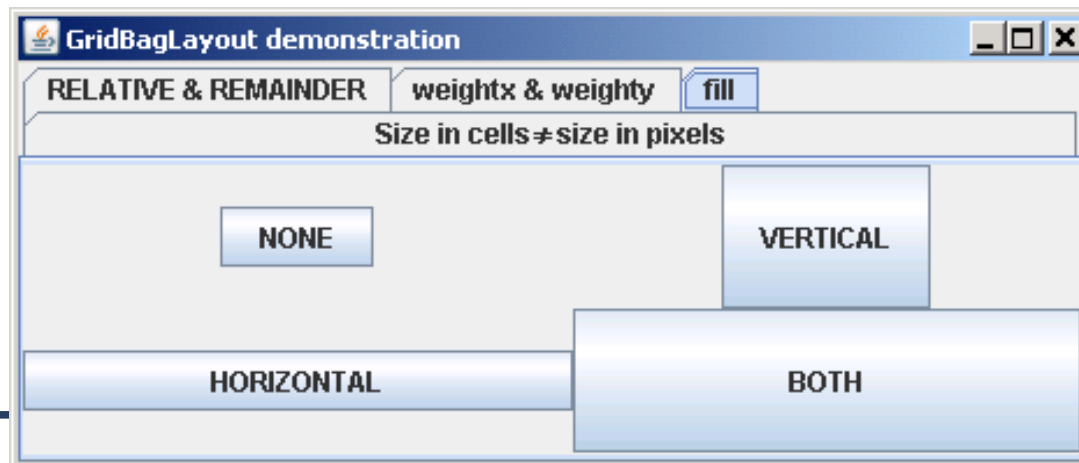
pas tout à fait 1/4 et 3/4,  
car c'est l'espace  
supplémentaire qui est  
réparti: il faut enlever la  
hauteur préférée des 2  
lignes

RELATIVE & REMAINDER		weightx & weighty	
Size in cells ≠ size in pixels			
0x0	0.2x0	0.2x0.3	2x0
0x0.9	0.2x0.9	0.2x0.9	2x0.9



# Le GridBagLayout

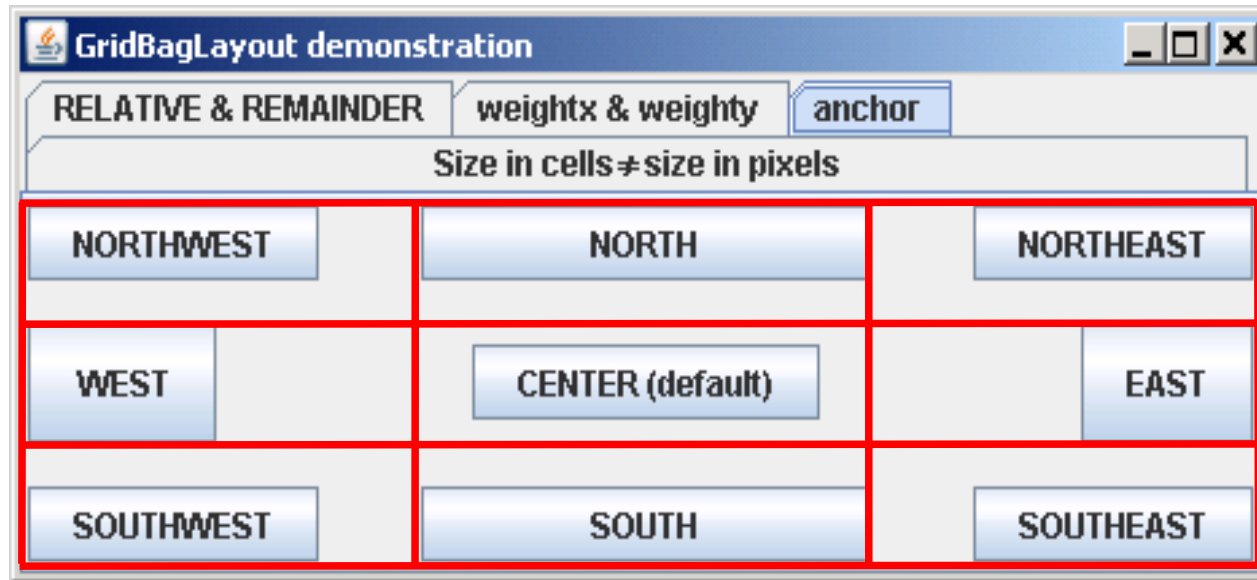
- si taille préférée < taille de la zone, **fill** indique comment gérer cet espace:
  - **NONE** (défaut): ne fait rien
  - **HORIZONTAL**: s'étire en largeur
  - **VERTICAL**: s'étire en hauteur
  - **BOTH**: s'étire en largeur et en hauteur





# Le GridBagLayout

- si le composant n'occupe pas toute sa zone, **anchor** indique où le placer:

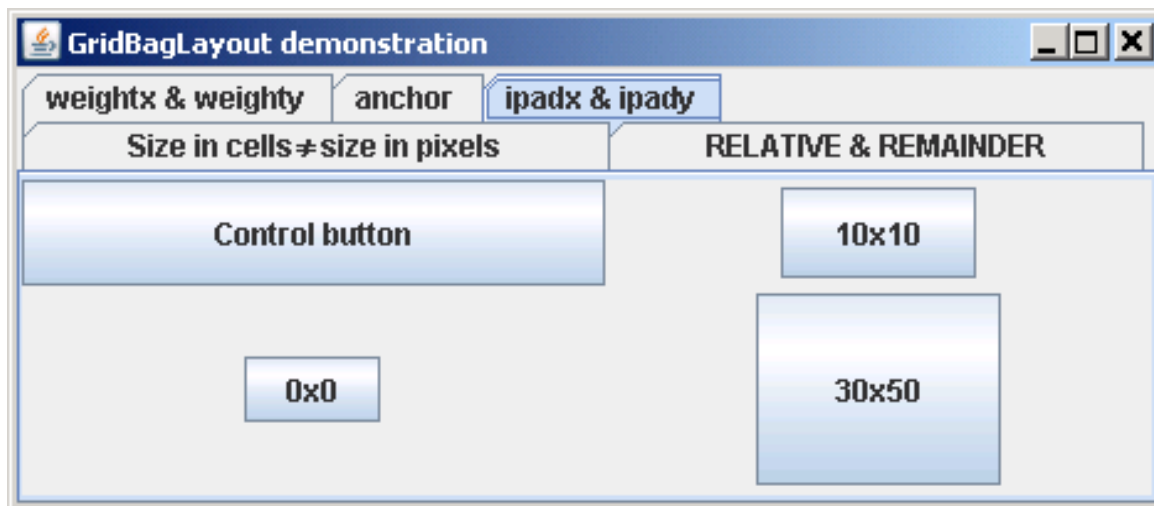


- d'autres valeurs sont possibles: cf Javadoc



# Le GridBagLayout

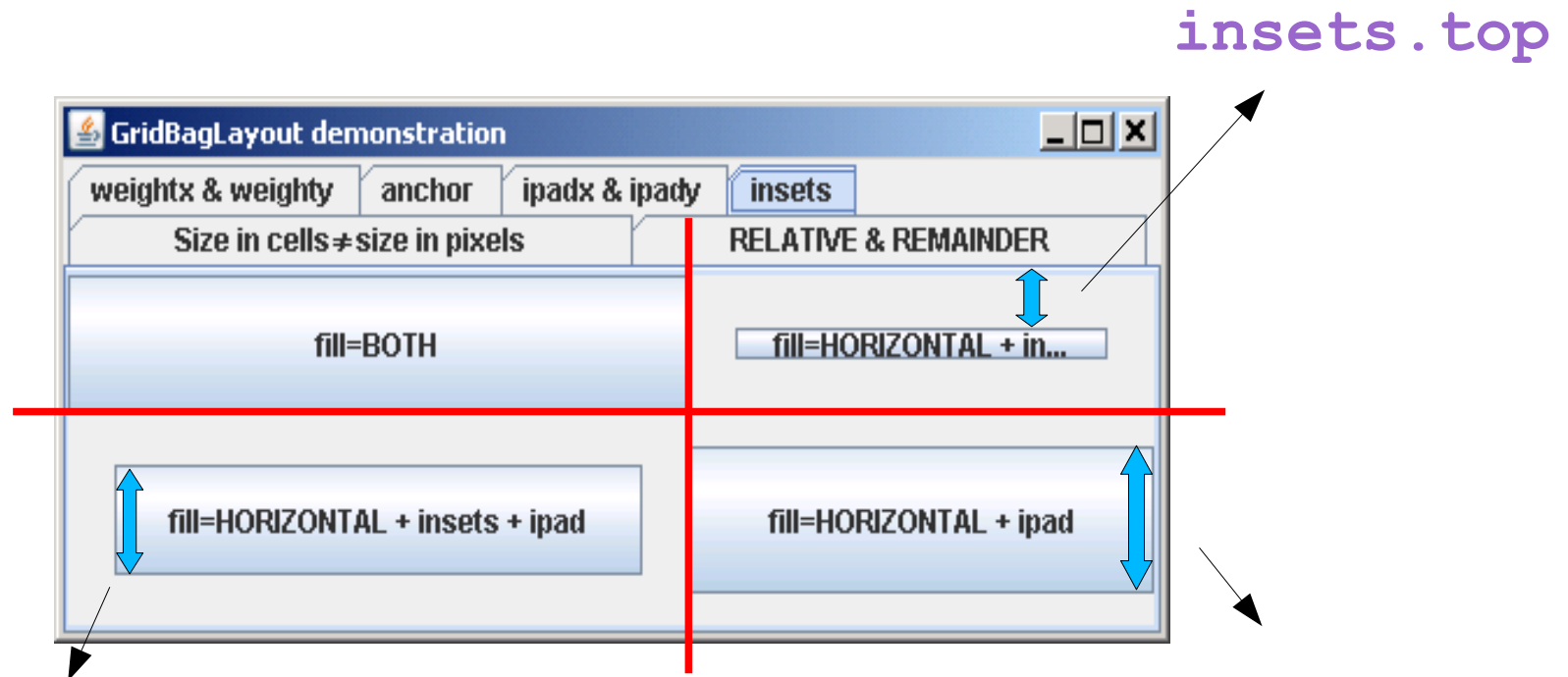
- si le composant n'occupe pas toute sa zone, **ipadx** et **ipady** indiquent les marges internes du composant (marges ajoutées à la taille minimum du composant):





# Le GridBagLayout

- **insets**=marges externes du composant, prioritaires sur tout le reste



moins qu'à côté, car les marges sont prioritaires

**getMinimumSize().height + ipady**



# Le GroupLayout

---

- autre façon (moins pratique) de gérer des grilles de composants, sans qu'ils aient tous la même taille
- chaque composant doit être mis dans 2 groupes:
  - un pour gérer l'alignement horizontal
  - un pour gérer l'alignement vertical
- **on ne met pas les composants directement dans le container !**

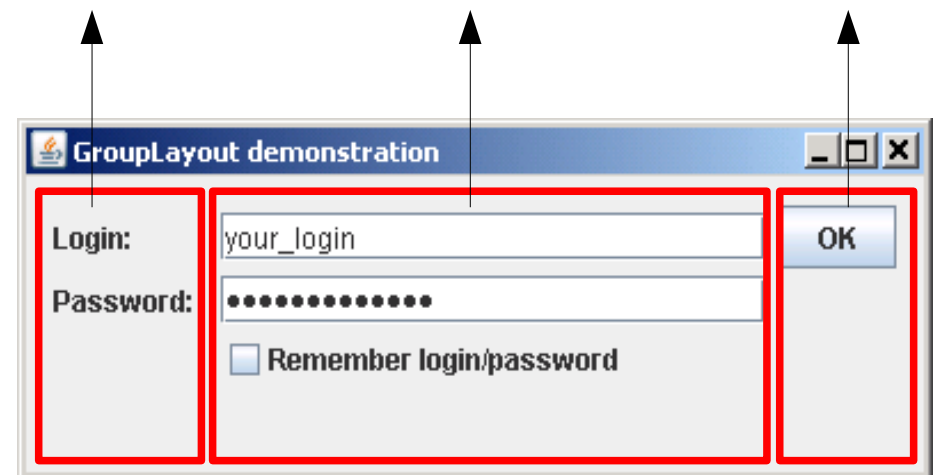


# Le GroupLayout

- on crée d'abord les composants, puis on les ajoute dans le groupe horizontal

```
GroupLayout.SequentialGroup hGroup=  
    layout.createSequentialGroup();  
hGroup.addGroup(layout.createParallelGroup()  
    .addComponent(label1)  
    .addComponent(label2));  
hGroup.addGroup(layout.createParallelGroup()  
    .addComponent(login)  
    .addComponent(passwd)  
    .addComponent(remember));  
hGroup.addGroup(layout.createParallelGroup()  
    .addComponent(OK));  
layout.setHorizontalGroup(hGroup);
```

le groupe horizontal  
contient 3 groupes verticaux

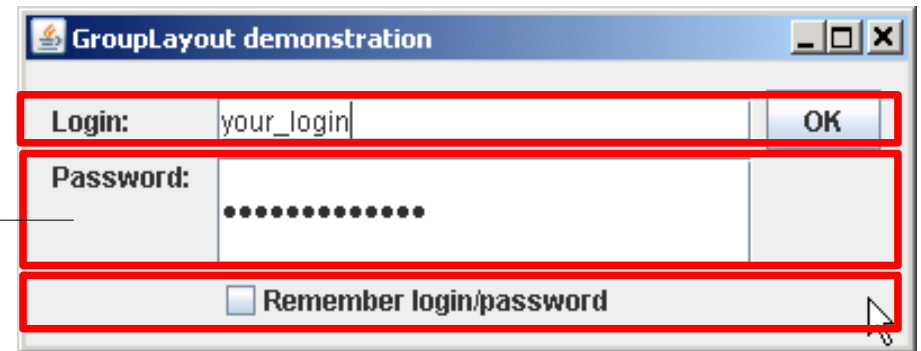




# Le GroupLayout

- puis, on fait la même chose pour le groupe vertical:

composants non alignés  
sur leur ligne de texte  
= pas ergonomique!



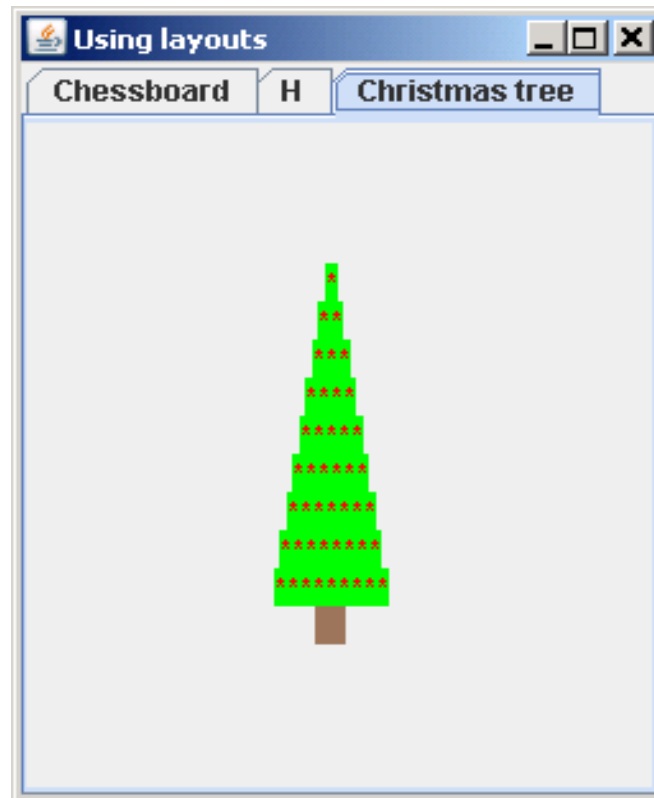
```
GroupLayout.SequentialGroup vGroup=  
    layout.createSequentialGroup();  
vGroup.addGroup(layout.createParallelGroup(Alignment.BASELINE)  
    .addComponent(label1)  
    .addComponent(login)  
    .addComponent(OK));  
vGroup.addGroup(layout.createParallelGroup(/*Alignment.BASELINE*/)  
    .addComponent(label2)  
    .addComponent(passwd));  
vGroup.addGroup(layout.createParallelGroup(Alignment.BASELINE)  
    .addComponent(remember));  
layout.setVerticalGroup(vGroup);
```



# Choisir le bon layout

---

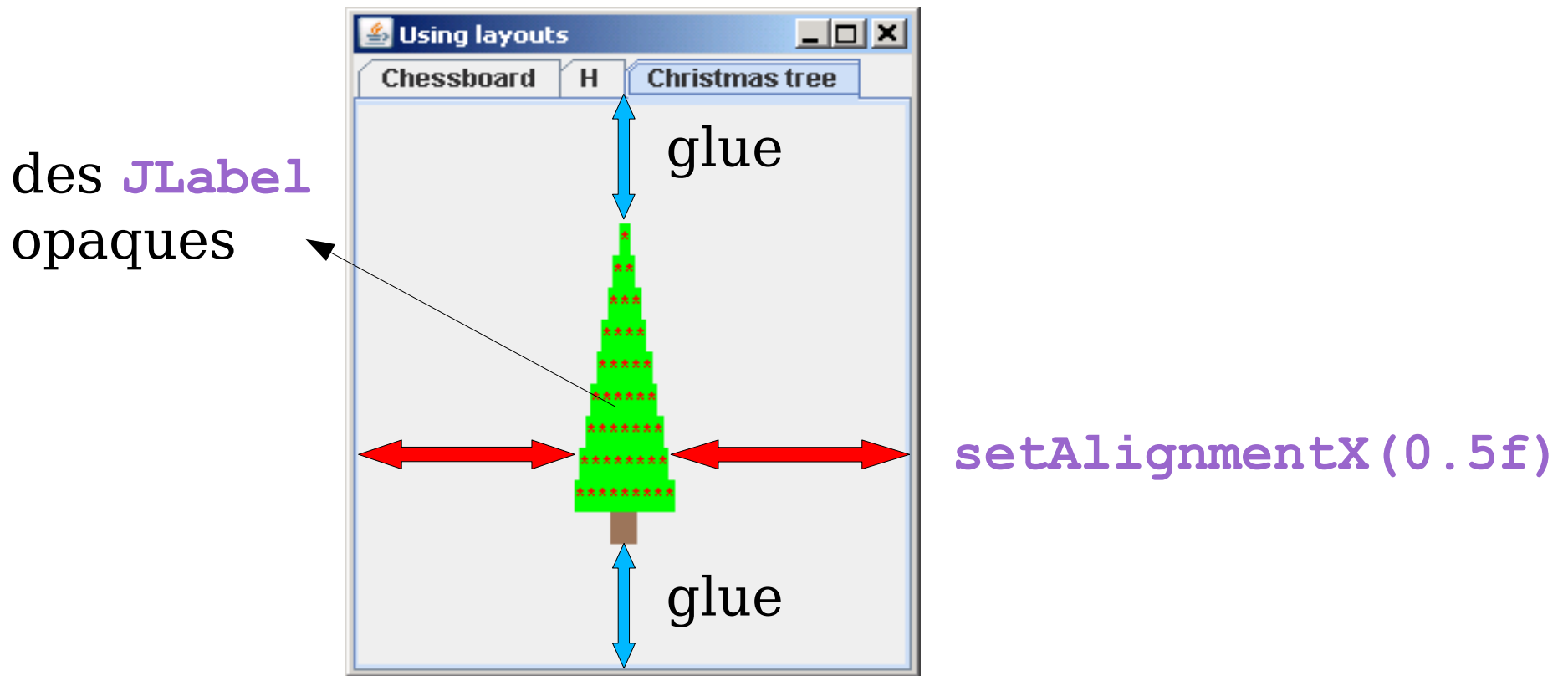
- quel layout choisir pour reproduire ce joli sapin ?





# Choisir le bon layout

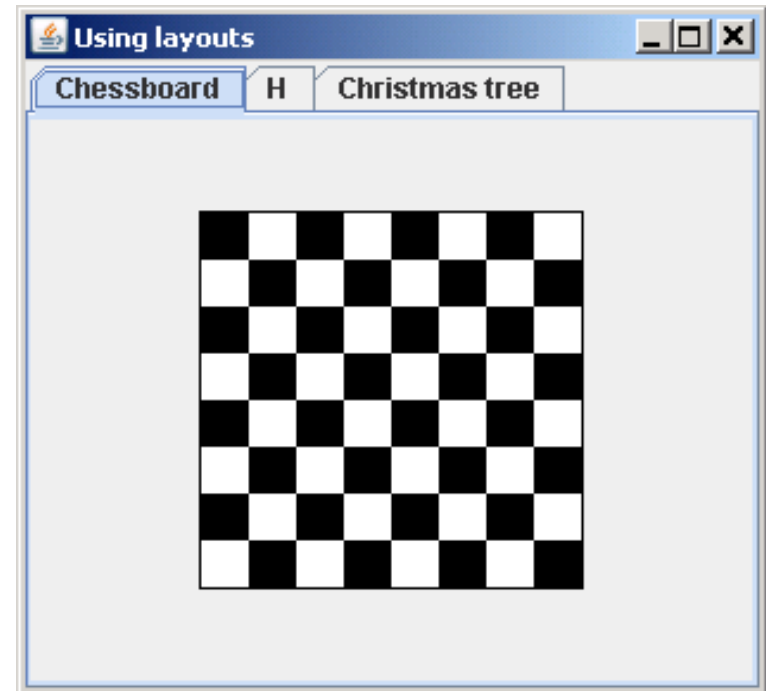
- réponse: le **BoxLayout**





# Combiner les layouts

- en pensant à combiner les layouts, on peut obtenir facilement des choses complexes
- 2<sup>ème</sup> exemple:
  - comment réaliser cet échiquier le plus simplement possible ?



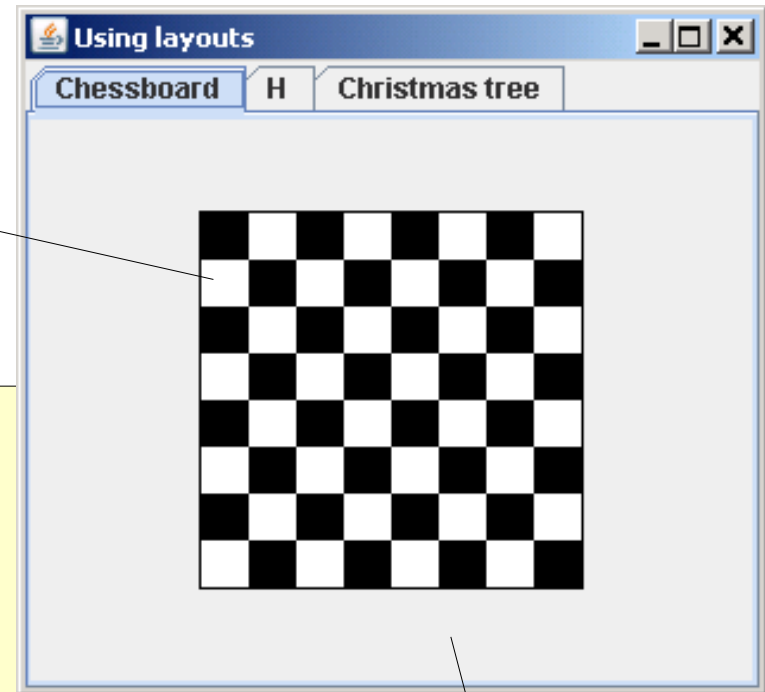


# Combiner les layouts

**GridLayout** + **JLabel** + bordure  
du panel pour la grille

```
private static Component createChessboard() {
    JPanel p = new JPanel(new GridLayout(8,8));
    p.setBorder(
        BorderFactory.createLineBorder(Color.BLACK));
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            JLabel l = new JLabel();
            l.setOpaque(true);
            /* We want square cells */
            l.setPreferredSize(new Dimension(20, 20));
            l.setBackground(((i+j) % 2 == 0) ? Color.BLACK
                : Color.WHITE);

            p.add(l);
        }
    }
    /* We use a GridBagLayout, because we want the chessboard at its
     * preferred size and at the center of the container */
    JPanel p2=new JPanel(new GridBagLayout());
    /* The default constraints are OK for our purpose */
    p2.add(p, null);
    return p2;
}
```

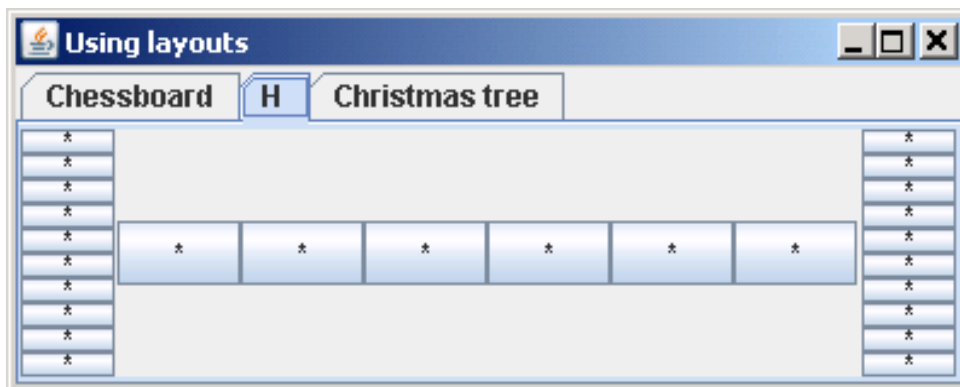
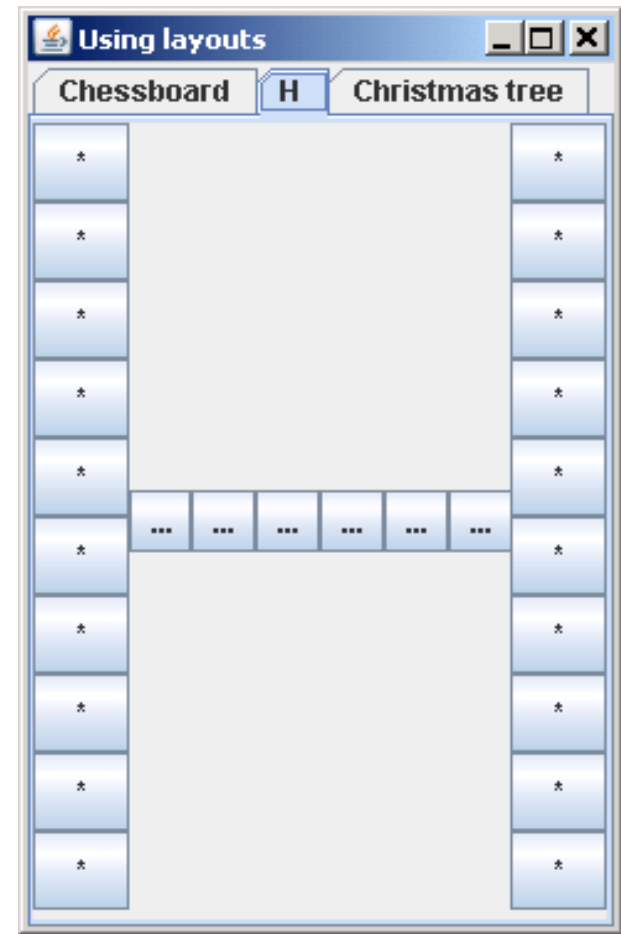
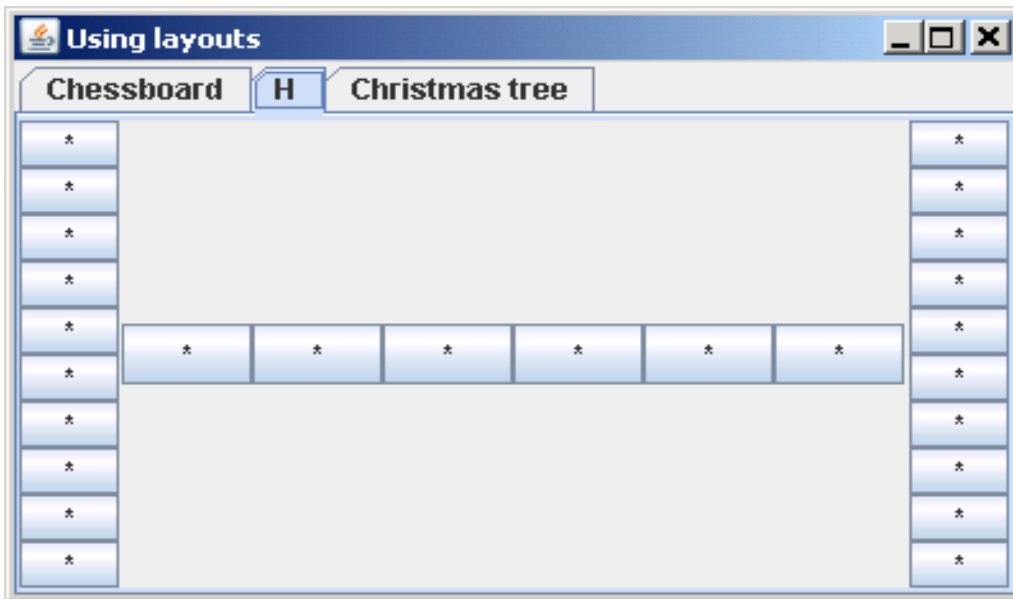


**GridBagLayout**  
pour centrer la  
grille



# Combiner les layouts

- 3<sup>ème</sup> exemple: le H

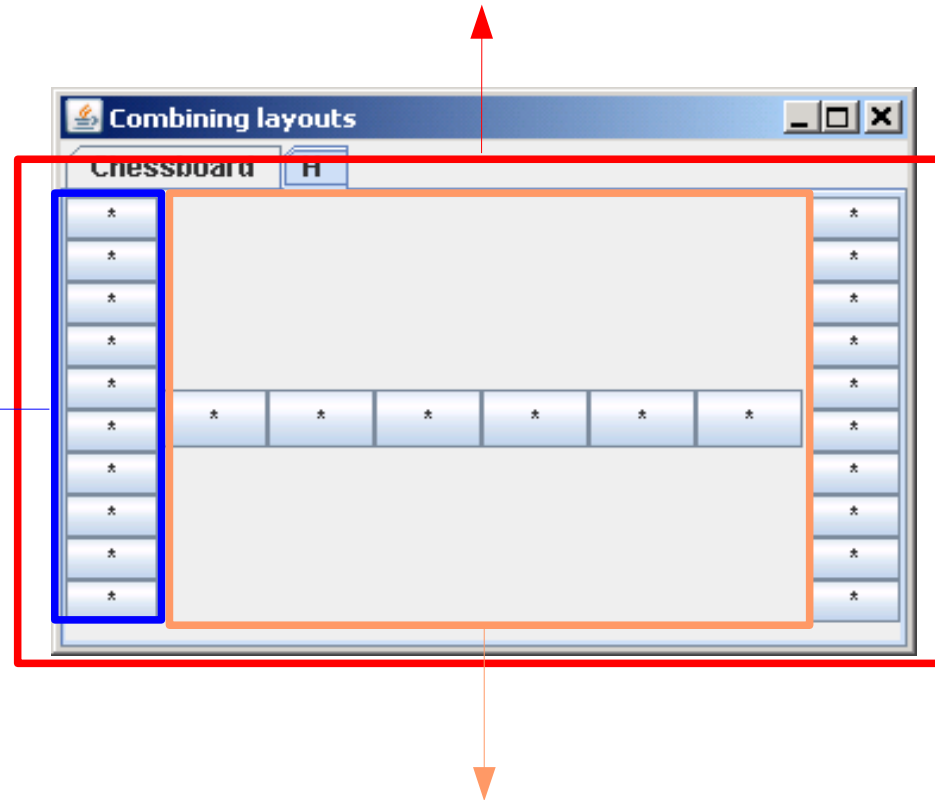




# Combiner les layouts

BorderLayout

GridLayout



GridBagLayout avec  
`fill=HORIZONTAL` et `weightx=1`



# Créer son LayoutManager

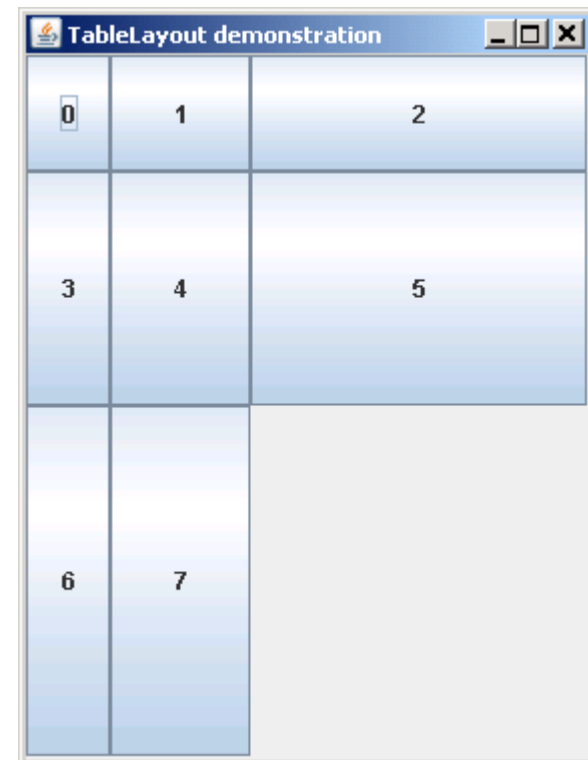
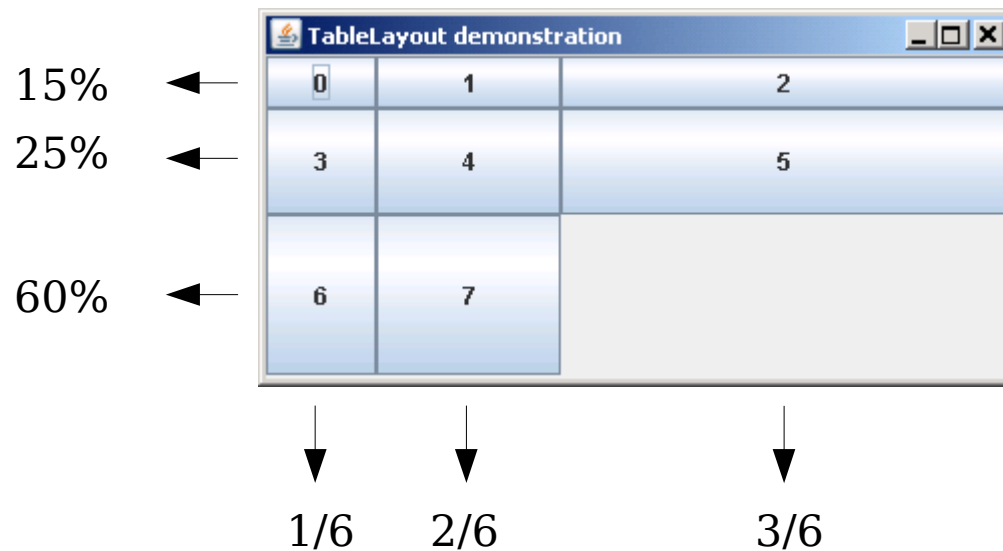
---

- facile, juste 5 méthodes à implémenter:
  - `minimumLayoutSize()`
  - `preferredLayoutSize()`
  - `addLayoutComponent(String name, Component c)`
  - `removeLayoutComponent(Component c)`
  - `layoutContainer(Container parent)`



# Créer son LayoutManager

- le **TableLayout** qui construit une grille en fonction de poids sur les lignes/colonnes, et qui maximise chaque composant:





# Créer son LayoutManager

- le constructeur doit prendre les poids:

```
private double[] weightx,weighty;
private double sumx,sumy;

public TableLayout(double[] weightx, double[] weighty) {
    if (weightx.length==0 || weighty.length==0) {
        throw new IllegalArgumentException(
            "Empty weight arrays are not permitted by TableLayout");
    }
    this.weightx=weightx.clone();
    this.weighty=weighty.clone();
    sumx=sum(weightx); sumy=sum(weighty);
}

private double sum(double[] weights) {
    double sum=0;
    for (int i=0;i<weights.length;i++) {
        sum=sum+weights[i];
    }
    return sum;
}
```

on **DOIT** copier les poids, car les valeurs des cases des tableaux pourraient être modifiées par l'appelant !



# Créer son LayoutManager

---

- la taille minimum est 0,0 (mais on pourrait la calculer plus finement)
- rien à faire quand on ajoute ou retire un composant

```
private final Dimension d=new Dimension(0,0);
@Override public Dimension minimumLayoutSize(Container parent) {
    return d;
}

@Override public void addLayoutComponent(String name, Component comp) {
    /* nothing to do */
}

@Override public void removeLayoutComponent(Component comp) {
    /* nothing to do */
}
```



# Créer son LayoutManager

---

- on calcule la taille préférée de façon à ce que chaque composant soit au moins à sa propre taille préférée:

```
@Override
public Dimension preferredLayoutSize(Container parent) {
    if (parent.getComponentCount() > weightx.length * weighty.length) {
        throw new IllegalStateException("Too much components in TableLayout!");
    }
    int i=0, n=parent.getComponentCount();
    int preferredX=0, preferredY=0;
    for (int y=0; i < n && y < weighty.length; y++) {
        for (int x=0; i < n && x < weightx.length; x++, i++) {
            Component c=parent.getComponent(i);
            preferredX=max(preferredX, (int) Math.ceil(c.getPreferredSize().width * sumx / weightx[x]));
            preferredY=max(preferredY, (int) Math.ceil(c.getPreferredSize().height * sumy / weighty[y]));
        }
    }
    return new Dimension(preferredX, preferredY);
}
```



# Créer son LayoutManager

- on calcule la hauteur/largeur de chaque ligne/colonne et on place les composants:

```
@Override
public void layoutContainer(Container parent) {
    if (parent.getComponentCount() > weightx.length * weighty.length) {
        throw new IllegalStateException(
            "Too much components in TableLayout!");
    }
    int i=0, n=parent.getComponentCount();
    int width=parent.getWidth(), height=parent.getHeight(), lastY=0;
    for (int y=0; i<n && y<weighty.length; y++) {
        int rowHeight=(int) (height*weighty[y]/sumy);
        int lastX=0;
        for (int x=0; i<n && x<weightx.length; x++, i++) {
            int columnWidth=(int) (width*weightx[x]/sumx);
            Component c=parent.getComponent(i);
            c.setBounds(lastX, lastY, columnWidth, rowHeight);
            lastX=lastX+columnWidth;
        }
        lastY=lastY+rowHeight;
    }
}
```