Interface-utilisateur graphique d'une application Java

DUT Informatique 1ère année Henri Garreta

TP 3 - Gérer la disposition. Réagir aux évenements

Objectif. Dans ce TP nous examinons deux notions fondamentales qui interviennent dans toutes les interfaces graphiques :

- un *gestionnaire de disposition* (ou *layout manager*) est un objet invisible associé à un conteneur ; il se charge du placement et du dimensionnement de chaque composant inclus dans le conteneur, aussi bien lors de l'affichage initial du conteneur que lors de chaque modification ultérieure de la taille et la forme de ce dernier.
- le *mécanisme des événements*, par lequel les actions que l'utilisateur fait sur l'interface, principalement à l'aide de la souris et du clavier, sont détectées et transmises à un objet compétent, dont elles provoquent l'activation d'une méthode, définie comme étant la réponse requise par l'action de l'utilisateur.

3.1 - Gestionnaires de disposition

Certains composants contiennent d'autres composants, on les appelle alors des *conteneurs*. Les *panneaux* (`JPanel`) en sont l'exemple le plus souvent utilisé, car ils ne servent qu'à cela : supporter un ensemble de composants qui leur ont été placés *dedans* (ou plutôt *dessus*). Cela impose un certain nombre de responsabilités, dont une des principales est la gestion de la taille et la disposition des composants en fonction de la taille et la forme du conteneur.

A chaque conteneur est attaché un *gestionnaire de disposition* (`LayoutManager`) qui se charge de *dimensionner* et *positionner* les composants placés dans le conteneur, aussi bien lors de l'affichage initial de ces objets que, ultérieurement, chaque fois que la taille ou la forme du conteneur change.

Dans cette section nous allons examiner les principaux gestionnaires de disposition prédéfinis :

`BorderLayout`
: Un tel gestionnaire place jusqu'à cinq composants dans cinq zones : le centre, le nord, le sud, l'est et l'ouest. Au besoin, ces composants sont étirés pour atteindre une largeur et/ou une hauteur déterminée par le gestionnaire.

`FlowLayout`
: Place les composants sans les étirer et « comme on écrit » (de la gauche vers la droite, puis du haut vers le bas)

`GridLayout`
: Disposition des composants, étirés si nécessaire, pour remplir un « quadrillage » rectangulaire dont toutes les cases sont identiques, ayant un nombre de lignes et de colonnes défini à l'avance.

Chaque conteneur est associé à un gestionnaire de disposition par défaut. La méthode `setLayout` permet de créer et associer un gestionnaire de disposition différent.

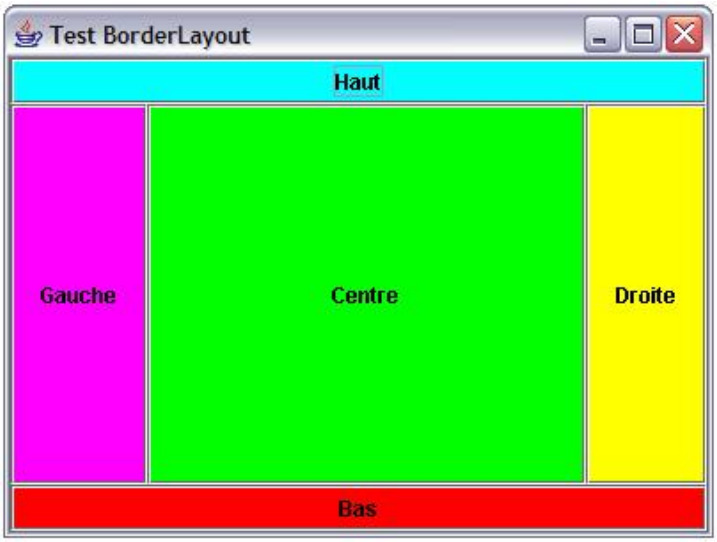
Le gestionnaire par défaut [du panneau de contenu] d'un cadre est `BorderLayout`. Le gestionnaire par défaut des autres types de panneaux est `FlowLayout`.

Exo 3.1.1 - BorderLayout

Correction

A. Réalisez une application réduite à un cadre de taille 400 x 300 portant cinq boutons (objets `JButton`) étiquetés « Haut », « Droite », « Bas », « Gauche », « Centre », placés dans les cinq zones reconnues par le gestionnaire `BorderLayout` `NORTH`, `EAST`, `SOUTH`, `WEST` et `CENTER` (ces identificateurs sont les noms des constantes de la classe

`BorderLayout`). Faites en sorte que ces boutons soient de couleurs différentes.

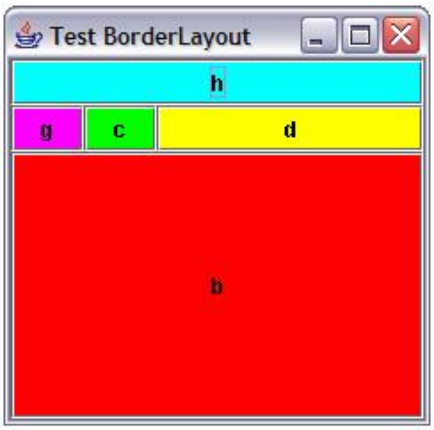


BorderLayout

Qu'arrive-t-il aux boutons lorsque vous modifiez la taille ou la forme du cadre ? Examinez comment sont placés les boutons lorsqu'il y en a moins de cinq. En « jouant » avec la longueur des textes inscrits dans les boutons et avec la taille du cadre, découvrez comment le gestionnaire `BorderLayout` détermine la forme et la taille de chaque composant.

Remplacez l'appel de `setSize(400,400)` par un appel de `pack()` (attention, un tel appel doit se faire *après* avoir ajouté les composants au conteneur). Que deviennent les tailles des composants?

B. Faites en sorte d'obtenir le cadre de la figure ci-dessous, où les textes sont réduits à une lettre, les boutons sont placés dans les cinq zones mentionnées et ils sont aussi réduits que possible, sachant que celui de droite *doit* avoir une largeur de 150 points et celui du bas *doit* avoir une hauteur de 150 points. Le cadre sera aussi réduit que possible, compte tenu de ce qui précède.



BorderLayout, encore

Indication. Pour définir la taille des bouton en présence d'un `BorderLayout`, les méthodes `setSize`, `setMinimumSize` et `setMaximumSize` n'ont pas l'effet qu'on pourrait espérer. Ici vous devrez vous intéresser plutôt à la méthode `setPreferredSize`.

Exo 3.1.2 - FlowLayout

Correction

A. Réalisez une application constituée d'un cadre contenant douze boutons (classe `JButton`) étiquetés « Janvier », « Février », etc. Les étiquettes des boutons seront définies par un tableau de chaînes de caractères constant. Le gestionnaire de disposition associé au panneau de contenu sera `FlowLayout` :

FlowLayout

B. Observez les changements de disposition des boutons qui se produisent lorsqu'on change la forme et la taille du cadre. Comme vous le voyez, la taille de chaque bouton est déterminée par le texte qui s'y affiche ; faites le nécessaire pour que tous les boutons aient la même taille :

Avec des boutons de même taille

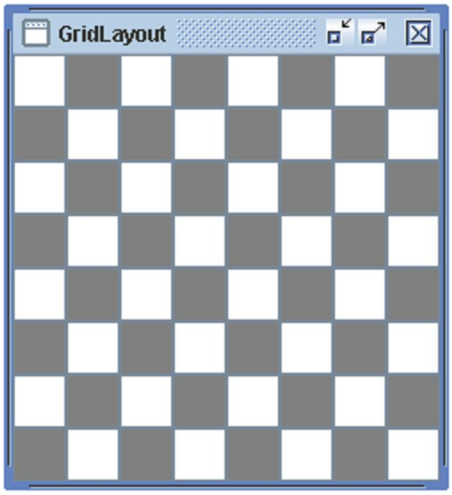
Indications.

- la taille des boutons se règle principalement par la méthode **`setPreferredSize`** (comme pour les autres composants),
- par défaut les boutons ne seront pas aussi resserrés autour du texte qu'ils contiennent ; pour réduire la marge autour du texte, intéressez-vous à leur méthode **`setMargin`** (il vous faudra construire un objet **`Insets`**, voyez la documentation),
- l'espacement entre les composants gérés par un *layout manager* est défini par un des paramètres du constructeur.

Exo 3.1.3 - GridLayout

Correction

A. Le gestionnaire de disposition **`GridLayout`** ne respecte pas la taille préférée des composants qu'il administre, mais leur impose une taille qui est la même pour tous ; de plus il est très contraignant. Réalisez une application qui affiche un échiquier (8 x 8 cases, alternativement blanches et grises). Les cases seront réalisées par des boutons (classe **`JButton`**) :



Il n'y a plus qu'à dessiner les pièces.

B. Le gestionnaire **`GridLayout`** commande aux composants qu'il gère de remplir tout l'espace alloué. Si ces composants sont très « malléables » (c'est le cas des boutons) cela peut ne pas être l'effet souhaité. Comment obtenir que lors d'un agrandissement du cadre, chaque composant garde sa taille préférée, comme sur la figure ci-dessous à droite (nous avons pris un échiquier 4 x 4 pour économiser l'espace) ?

Agrandissement du conteneur : résister à l'effet montré à gauche et obtenir l'effet montré à droite

Exo 3.1.4 - Une interface graphique « utile »

Correction

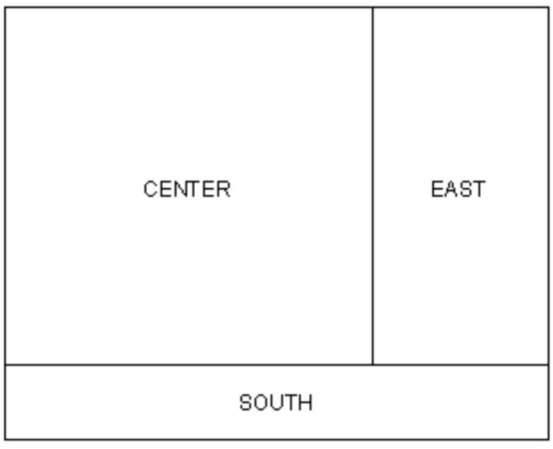
Les trois exercices précédents montrent comment faire fonctionner les gestionnaires de disposition basiques, mais ne donnent pas vraiment l'impression qu'avec ces gestionnaires on peut réaliser des interfaces graphiques comme on a l'habitude d'en voir dans les applications courantes. Pour vous convaincre que la chose est possible, vous allez réaliser l'interface graphique suivante, correspondant à un masque de saisie pour l'acquisition des membres d'un club où on pratique divers sports (On ne s'occupe ici que d'apparence graphique, on verra plus tard comment valider et acquérir les informations saisies à travers ce composant) :

Définissez ici une classe **`SaisieMembre`**, sous-classe de **`JFrame`**. Nous la reprendrons plus tard, sous forme de **`JDialog`**, dans un autre exercice (cf. Exo 3.2.6) où on verra comment récupérer dans un programme les informations saisies à l'aide d'une telle interface.

Il s'agit donc d'organiser la taille et l'emplacement des composants. Les choix suivants paraissent raisonnables :

- les deux boutons *OK* et *Annuler* occupent une région qui n'a pas besoin d'être beaucoup plus haute que les boutons eux-mêmes et qui, surtout, n'a pas besoin de grandir verticalement lorsque la hauteur du cadre augmente,
- ces deux boutons sont centrés horizontalement, c'est-à-dire à égale distance des bords gauche et droit du panneau,
- les cases à cocher concernant les sports occupent une région rectangulaire qui n'a pas besoin d'être plus large que le sport dont le nom est le plus long et qui, surtout, n'a pas besoin de grandir horizontalement lorsque la largeur du cadre augmente,
- à l'intérieur de cette zone, verticalement, les cases à cocher concernant les sports sont disposées régulièrement.

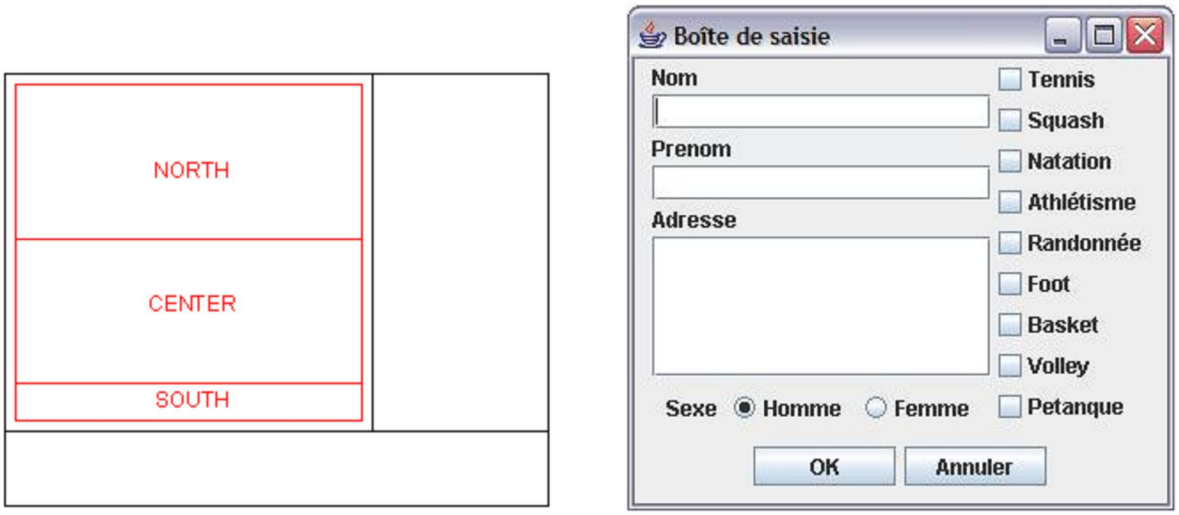
Tout cela amène à une première organisation du panneau de contenu de **`SaisieMembre`** : trois panneaux gérés par un **`BorderLayout`** comme sur la figure suivante :



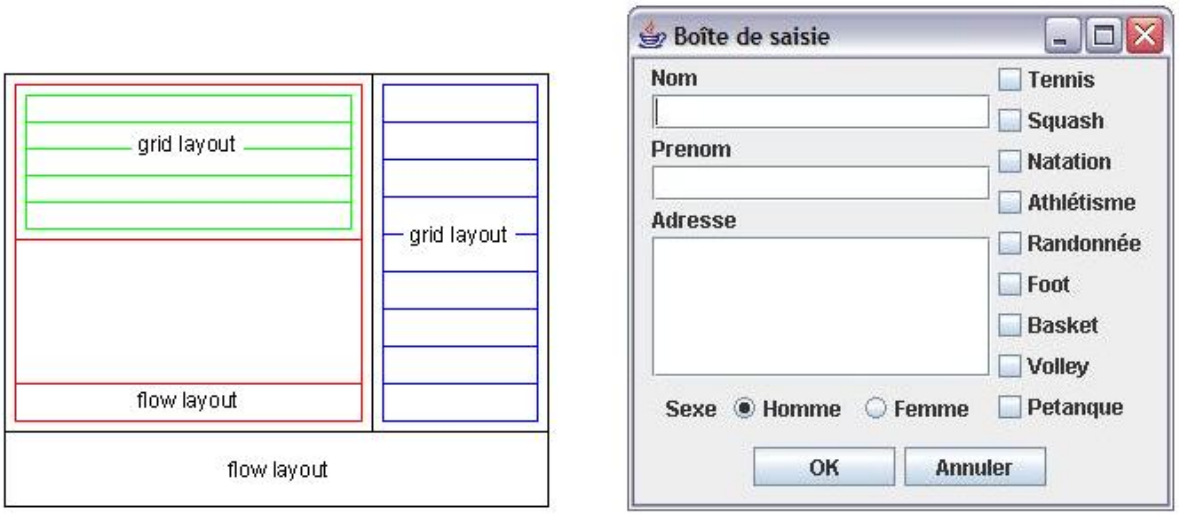
Le panneau central porte neuf composants. Les cinq premiers (l'étiquette *Nom*, un champ de texte, l'étiquette *Prenom*, un autre champ de texte et l'étiquette *Adresse*) se partagent verticalement et régulièrement un panneau qui occupe toute la largeur et qui n'a pas besoin de grandir lorsque la hauteur augmente.

Les trois derniers composants (l'étiquette *Sexe* et les deux boutons-radio *Homme*, *Femme*) occupent un panneau qui prend toute la largeur mais n'a pas besoin de grandir lorsque la hauteur augmente. Finalement, le composant restant est une zone de texte qui doit grandir lorsque la hauteur augmente (sinon, pour quelle raison la hauteur augmenterait-elle ?).

Ce qui nous amène à organiser le panneau central comme trois composants (deux panneaux et une zone de texte) gérés par un second **`BorderLayout`** comme sur la figure suivante :



Enfin, deux **`GridLayout`** et deux **`FlowLayout`** sont nécessaires pour gérer les composants élémentaires (cases à cocher, boutons, etc.) :



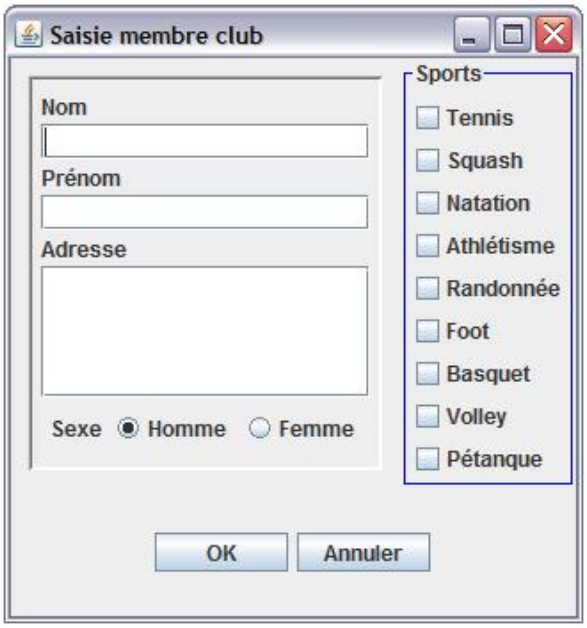
Exo 3.1.5 - A propos des marges et bordures

Correction

Il est souvent nécessaire de montrer graphiquement qu'un ensemble de composants sont regroupés ensemble et, éventuellement, chapeautés par un titre commun. On obtient cela en plaçant ces composants sur un panneau introduit à cet effet, auquel on donne une bordure *ad hoc*.

La manière la plus simple de construire des bordures consiste à appeler des méthodes statiques de la classe « fabrique » **`BorderFactory`**. Les bordures les plus fréquemment utilisées sont biseautées (*bevel*ed), gravées (*etched*) ou vides (*empty*, cela met de la marge autour de quelque chose) ; de plus, on peut titrer (*titled*) n'importe quel autre type de bordure. Enfin, la création d'une bordure composée (*compound*) permet d'associer deux bordures pour qu'elle se présentent comme une seule.

Modifiez la classe réalisée dans l'exercice précédent afin que l'interface obtenue ressemble à ceci :



3.2 - Réagir aux événements

Il va être question ici de la détection des actions de l'homme sur la machine. En Java, une action d'un utilisateur sur un composant produit, de la part de ce dernier, la création et l'envoi d'un *événement*, qu'il faut voir comme une sorte de « compte rendu » décrivant l'action en question et ses circonstances. Trois sortes d'objets sont mis en œuvre :

- le *type* de l'événement, représenté par une classe dont le nom est `java.awt.event.XxxEvent`, où ***`Xxx`*** dépend de l'événement : `ActionEvent`, `MouseEvent`, etc.
  Ces objets comportent de nombreuses propriétés destinées à représenter les circonstances dans lesquelles l'événement est né : `x`, `y` (position de la souris), `source` (cf. ci dessous), `actionCommand` (texte éventuellement associé à la source), etc.

- la *source* de l'événement, c'est-à-dire l'objet qui, ayant subi une action de l'utilisateur, a créé un événement afin de notifier cette action. En Java, *notifier* veut dire appeler une méthode (spécifique de l'événement) sur l'objet destinataire de la notification.
  La plupart des composants peuvent être sources d'événements dont les types dépendent de celui du composant. Un objet est source d'un événement ***`Xxx`*** si et seulement si dans sa classe il y a une méthode de signature `void addXxxListener(XxxListener unAuditeur)`

- un ou plusieurs *auditeurs* de l'événement. Ce sont les objets qui reçoivent les notifications en question, ce qui requiert deux choses :
  - il faut que les auditeurs *puissent* recevoir ces notifications, c'est-à-dire qu'ils possèdent les méthodes spécifiques en question. On garantit cela en imposant que les auditeurs soient des implémentations d'interfaces définies à cet effet : `ActionListener`, `MouseListener`, etc.
  - il faut que les auditeurs *souhaitent* recevoir ces notifications, c'est-à-dire qu'ils aient été « inscrits » comme tels auprès de la source correspondante. Cette inscription se fait à l'aide de la méthode `addXxxListener(XxxListener unAuditeur)` mentionnée.

Exo 3.2.1 - Le plus simple : « écouter » des boutons

Correction

Il s'agit ici de réaliser une application qui affiche un cadre contenant un panneau (coloré) avec trois boutons dont l'unique effet est de changer la couleur du panneau :

Trois boutons pour définir la couleur d'un panneau

Vous allez écrire plusieurs versions de ce programme, suivant les manières les plus fréquentes d'écouter les événements dont les composants sont sources.

A. Dans une première version il y aura deux classes : une sous-classe de `javax.swing.JFrame` qui représente le cadre et une implémentation de l'interface `java.awt.event.ActionListener` dont l'unique instance est l'auditeur des événements issus des trois boutons. Cette classe auditeur mémorise le composant (donné comme argument du constructeur) dont elle commande la couleur.

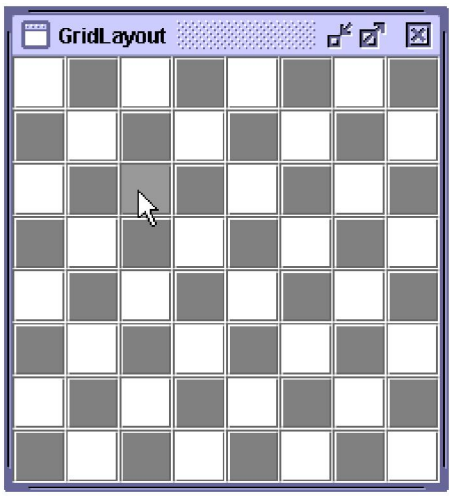
B. Deuxième version : comme la précédente, mais avec une seule classe qui est en même temps le cadre et l'auditeur des événements sur les boutons.

C. Comme dans la version A, mais la classe auditeur est destinée à avoir trois instances, une pour chaque bouton. Chacune mémorise donc le composant à colorier et aussi la couleur à appliquer, indiqués l'un et l'autre dans le constructeur de la classe.

D. On reprend l'idée de la version C (un auditeur distinct pour chaque bouton) mais en remarquant qu'il n'y a pas besoin de définir une classe spéciale pour cela. Il suffit d'utiliser une classe anonyme, crée par spécialisation et instanciation de `ActionListener` juste au moment de l'appel de `addActionListener`.

Exo 3.2.2 - Distinguer les sources

Correction



Un casier fait de 64 boutons

Reprenez l'échiquier de l'exercice 3.1.3 (A) et faites en sorte que chaque clic produise l'affichage des coordonnées de la case sur laquelle on a cliqué (convention usuelle : lignes 1, 2, 3... du bas vers le haut, colonnes a, b, c... de la gauche vers la droite). Par exemple, le clic montré ci-dessus produit l'affichage de la chaîne `c6`.

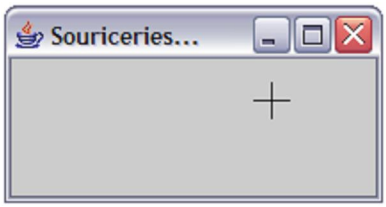
A. Il faut donc que chacun des 64 boutons mémorise les coordonnées de la case du damier qu'il représente. Dans une première version, utilisez pour cela la propriété `ActionCommand`, une chaîne de caractères associée à chaque bouton qu'on retrouve dans l'objet `ActionEvent`.

B. Imaginons qu'on ne souhaite pas mémoriser les coordonnées de chaque bouton dans la chaîne `ActionCommand`. Ecrivez une deuxième version dans laquelle on crée les boutons comme instances d'une sous-classe de `JButton` définie expressement, capable de mémoriser ces coordonnées.

Exo 3.2.3 - Guetter la souris

Correction

Écrivez une classe de panneau (sous-classe de `JPanel`) ayant la propriété que le curseur prend la forme de deux cheveux croisés (*cross hair*) lorsqu'il se trouve au-dessus de ce panneau.



Le curseur change de forme au-dessus du cadre

Définissez une fonction `main` qui crée un cadre (classe `JFrame`) et lui ajoute un tel panneau.

A. Dans une première version, faites en sorte que le panneau soit aussi l'auditeur de ses propres événements souris, et constatez que cela fonctionne mais n'est pas très commode.

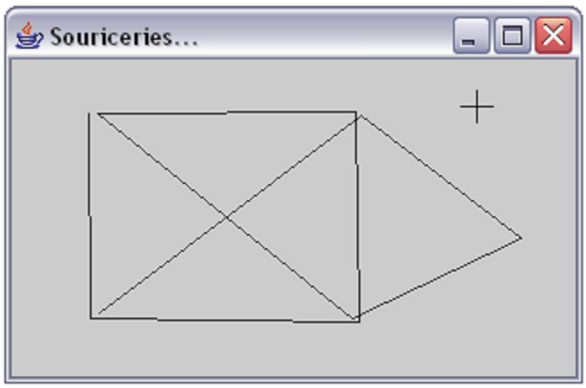
B. Deuxième version : écrivez un programme plus simple dans lequel l'auditeur d'événements souris est une instance d'une classe anonyme, sous-classe de `MouseAdapter`, créée à l'occasion de l'appel de `addMouseListener`.

Exo 3.2.4 - Tracer une ligne polygonale, I

Correction

Complétez le programme précédent (l'une ou l'autre version) de sorte que chaque clic de la souris produise la création, à l'endroit où le clic s'est produit, d'un nouveau sommet d'une ligne polygonale qu'il faut afficher.

Les points seront représentés par les instances d'une classe `Point` (à définir), la ligne polygonale par un `ArrayList` de points. L'affichage sera réalisé par une des méthodes `paint(Graphics g)` ou `paintComponent(Graphics g)` de la classe du panneau, méthode qu'il faudra donc redéfinir.



Tracé d'une ligne

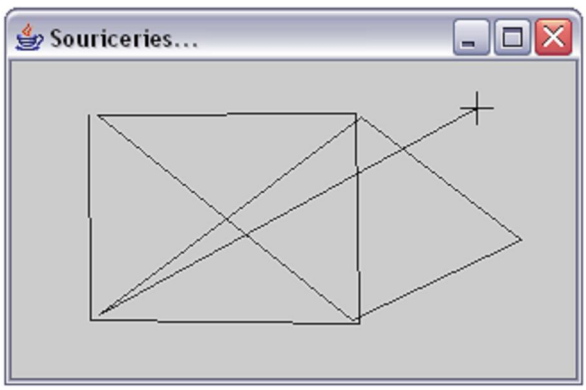
Exo 3.2.5 - Tracer une ligne polygonale, II

Correction

Complétez le programme précédent de sorte que le dernier point cliqué et la position courante de la souris soient *constamment* reliés par un trait, dont l'extrémité bouge avec la souris.

Indications. 1° En plus des événements « *Mouse* » il faudra s'intéresser aussi aux événements « *MouseMotion* ».

2° Une manière simple de gérer le dernier segment (mobile) de la ligne polygonale est de considérer que celle-ci possède un sommet de plus, le dernier, dont les coordonnées sont celles de la souris, mis à jour chaque fois que cette dernière bouge.



Le dernier segment suit le curseur

Exo 3.2.6 - Récupérer les données d'une boîte de dialogue

Correction

Il s'agit ici de reprendre l'interface graphique réalisée à l'exercice 2.1.4 et de montrer qu'on est capable de récupérer des données saisies grâce à elle. On écrira une méthode **`main`** extrêmement simple qui fait apparaître la boîte de saisie puis, lors de sa disparition, affiche sur la console les valeurs saisies.

Voici les principales modifications qu'il faut faire sur la classe **`SaisieMembre`** :

- en faire une sous-classe de **`JDialog`** (une boîte de dialogue) au lieu de **`JFrame`** (un cadre),
- par conséquent, l'appel du constructeur de la super-classe est plus complexe : il faut indiquer un cadre ou un dialogue propriétaire de la nouvelle boîte de dialogue (ici il n'y en a pas) et, surtout, préciser que la nouvelle boîte de dialogue est *modale* (c'est-à-dire bloquante, c'est-à-dire synchrone). En première ligne du constructeur, mettez : « **`super((JFrame)null, true);`** »,
- s'assurer que l'appel de `setVisible(true)` *n'est pas* fait dans le constructeur de **`SaisieMembre`**, mais dans la méthode qui appelle ce constructeur,
- définir dans la classe **`SaisieMembre`** un certain nombre de variables d'instance publiques (**`nom`**, **`prenom`**, **`adresse`**, etc.) destinées à contenir, à la fin du dialogue, les informations saisies,
- définir dans **`SaisieMembre`** une variable d'instance publique additionnelle, nommée par exemple **`sortieParOK`**, destinée à indiquer si le dialogue s'est « bien » (bouton *OK*) ou « mal » (bouton *Annuler*) terminé,
- définir les gestionnaires des actions (**`ActionListener`**) sur les bouton *OK* et *Annuler*. Tous les deux se terminent par un appel de la méthode `dispose()` qui fait disparaître la boîte de dialogue et *débloque la méthode* qui avait appelé **`setVisibled(true)`** et, donc, était bloquée à cet endroit. Juste avant, ils donnent la valeur *true* ou *false* à la variable **`sortieParOK`**.
- faire en sorte que le gestionnaire d'action sur le bouton *OK* copie dans les variables d'instance **`nom`**, **`prenom`**, **`adresse`**, etc., les contenus des champs et cases à cocher correspondants.
  N.B. C'est ici que, dans une application sérieuse, il faudrait écrire un code plus ou moins important pour valider les données acquises.

Bien entendu, on modifiera ensuite la méthode main pour afficher les valeurs acquises, par exemple sous la forme suivante

```
Nouveau membre:
        Garreta Henri (homme)
        11, rue Va-a-la-calanque - 13008 Marseille
    Tennis
    Randonnée
    Petanque
```

La suite des TP...