Programmation Orientée Objet
IG2I, LE2 – 2023-2024

TP9 – Paint

Diego Cattaruzza, Maxime Ogier, Pablo Torrealba

Contexte et objectif du TP

Dans ce TP nous allons terminer l'application très innovante et futuriste que nous avons commencé lors du dernière TP : le paint !!

Ce TP permet d'illustrer les notions suivantes :

- les objets graphiques **JButon** et **JMenu** ;
- les classes *internes*, *internes locales* et *anonymes*.

Bonnes pratiques à adopter

- Testez votre code au fur et à mesure de votre avancement.
- Respectez les conventions de nommage Java.
- Faites des indentations de manière correcte.
- Donnez des noms compréhensibles et pertinents à vos variables, attributs, classes, méthodes.
- Respectez autant que possible le principe d'encapsulation.
- Utilisez des structures de données adaptées.
- Faites des méthodes courtes avec peu de niveaux d'indentation.
- Ne faites pas de duplication de code, mais privilégiez le code réutilisable.
- Utilisez les outils de votre IDE pour faire du *refactoring* (clic-droit puis '*Refactor*' sur IntelliJ IDEA).
- Utilisez la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez.
- Quand c'est nécessaire, commentez votre propre code de façon pertinente au format Javadoc (commentaire commençant par "`/**`" au-dessus des définitions des attributs et méthodes).

1 Ajout de formes graphiques

Dans le TP précédent, vous avez créé une application vous permettant de dessiner des droites. L'application est très belle et en parfait état de marche, mais quand même un peu limitée en terme de fonctionnalités. Dans ce TP, nous allons inclure la possibilité de dessiner d'autres formes, de les effacer, d'effacer une partie de nos œuvres d'art.

Commençons par dessiner des cercles. Pour ce faire, vous pouvez utiliser la méthode **drawOval** de la classe **Graphics**. Cette méthode possède quatre paramètres : les coordonnées du point en haut à gauche de l'ovale, la largeur et la hauteur de l'ovale. Un cercle est en fait un ovale avec la même largeur et hauteur. De la même façon que pour une droite, un cercle est dessiné en cliquant sur la souris en un point de la zone graphique. La souris est ensuite déplacée jusqu'à un autre point. Le cercle doit apparaître à l'intérieur du carré défini par les deux points où l'utilisateur a appuyé et relâché la souris.

Question 1. Après avoir ouvert votre projet **Paint**, ajoutez dans le paquetage **modele** une classe **Cercle**. Cette classe hérite de la classe **Forme**. La classe a deux attributs : le point en haut à gauche et la largeur du cercle. Allez voir la Javadoc de la classe **Graphics** pour bien choisir le type de l'attribut largeur. Ajoutez un constructeur par données du point initial (où l'utilisateur clique sur la souris), du point final (où l'utilisateur relâche la souris), et de la couleur de votre dessin. Les attributs devront être initialisés correctement par rapport aux valeurs des paramètres du constructeur. Redéfinissez la méthode **seDessiner**. N'hésitez pas à faire quelques dessins sur une feuille pour vous aider, et testez en particulier les différents cas : si le point initial est à gauche ou à droite du point final, ou si le point initial est au-dessus ou en dessous du point final.

Question 2. Dans la classe **ZoneGraphique**, modifiez la méthode **dessin** pour pouvoir dessiner des cercles. Testez le bon fonctionnement de votre code.

Très bien, maintenant notre application **Paint** nous permet de dessiner des cercles. Ça serait dommage de ne pas pouvoir dessiner aussi des ovales (en plus des cercles) !

Question 3. Modifiez votre code pour pouvoir dessiner des ovales.

Question 4. La liste déroulante des formes vous donne la possibilité de sélectionner des formes que vous ne pouvez pas encore dessiner. Ajoutez dans votre application la possibilité de dessiner des rectangles (et si vous le souhaitez, des carrés aussi). Consultez la Javadoc pour trouver la méthode pour les dessiner.

Il reste une seule forme dans la liste déroulante que nous ne n'avons pas encore dessinée : le triangle. Malheureusement, la classe **Graphics** ne possède pas de méthode "*drawTriangle*" qui permettrait de faire cela directement. Cepen-

dant, cette classe possède une méthode **drawPolygon** qui permet de dessiner des polygones. Cette méthode prend comme paramètres deux tableaux avec respectivement les abscisses et les ordonnées des points du polygone. Le troisième paramètre est le nombre de sommets du polygone (ici 3 pour un triangle). Vous pouvez utiliser cette méthode pour dessiner un triangle isocèle : il sera inscrit dans la boîte englobante définie par les points lors du clic et du relâchement de la souris.

Question 5. Ajoutez dans votre application une classe **Triangle**. Cette classe hérite de la classe **Forme**. Ajoutez les constructeurs et les méthodes nécessaires pour dessiner des triangles. N'oubliez pas de modifier également la méthode **dessin** de la classe **ZoneGraphique**. Testez que vous dessinez bien des triangles isocèles.

2 Les boutons : JButton

Jusqu'ici, nous n'avons fait que dessiner dans notre fenêtre. Pour effacer le dessin, nous devons fermer notre application et la redémarrer. Pas terrible, n'est-ce pas ? Il serait utile d'avoir :

- un bouton "*effacer*" pour pouvoir ramener la zone graphique à son état initial ;
- un bouton "*defaire*" pour effacer la dernière forme dessinée.

L'API Swing possède une classe **JButton** permettant de placer des boutons sur des composants graphiques (des panneaux de type **JPanel** par exemple). Le code suivant permet de créer un bouton sur lequel sera écrit "*Appuyez ici*", de donner une taille de 150 pixels de long par 10 pixels de hauteur à ce bouton, et de l'ajouter sur un panneau.

```
1  JButton monBouton = new JButton("Appuyez ici");
2  monBouton.setPreferredSize(new Dimension(150, 10));
3  JPanel monPanneau = new JPanel();
4  monPanneau.add(monBouton);
```

Notez que, contrairement à une fenêtre, un bouton est visible par défaut ; il n'est donc pas nécessaire de lui appliquer la méthode **setVisible(true)** (mais cela reste possible).

Question 6. Ajoutez dans la classe **BarreHaute** deux attributs de type **JButton** et nommez-les *effacer* et *defaire*. Initialisez-les dans le constructeur de la classe. Vérifiez que les boutons apparaissent bien dans la barre.

2.1 Gestion du bouton avec un écouteur

Un bouton ne peut déclencher qu'un seul événement correspondant à l'action de l'utilisateur sur ce bouton. Généralement, cette action est déclenchée par un clic sur le bouton, mais elle peut aussi être déclenchée à partir du clavier (sélection du bouton et appui sur la barre d'espace).

L'événement qui nous intéresse est l'unique événement d'une catégorie d'événements nommée *Action*. Il faudra donc :

- créer un écouteur qui sera un objet d'une classe qui implémente l'interface **ActionListener** ; cette dernière ne comporte qu'une seule méthode nommée **actionPerformed** qu'il faudra implémenter ;
- associer cet écouteur au bouton par la méthode **addActionListener** (présente dans tous les composants qui en ont besoin, donc en particulier dans la classe **JButton**).

Vous pouvez remarquer ici que la classe **BarreHaute** implémente déjà l'interface **ActionListener**. Les objets qui représentent les listes déroulantes pour les couleurs et les formes (objets de type **JComboBox**) utilisent déjà comme écouteur celui défini dans la classe **BarreHaute**. On pourrait imaginer associer aux deux boutons que nous venons d'ajouter ce même écouteur. Cela impliquerait alors de modifier la méthode **actionPerformed** de la classe **BarreHaute**. C'est techniquement possible, et en utilisant l'objet de type **ActionEvent** passé en paramètre de la méthode **actionPerformed**, on pourrait quelle est la source qui a déclenchée l'appel de la méthode. On pourrait ainsi faire un traitement différent selon que la source soit une liste déroulante, le bouton "*effacer*" ou le bouton "*défaire*".

Cependant, il convient de remarquer que ce n'est pas une bonne pratique de gérer plusieurs sources d'événements(par exemple plusieurs boutons) avec le même écouteur. En effet, cela oblige à utiliser une structure conditionnelle (avec des **if** ou un **switch**) afin de déterminer la source de l'événement, puis à écrire le traitement correspondant. Cela fonctionne, mais n'aide pas à la lisibilité ni la maintenabilité du code.

Ainsi, une autre possibilité serait de créer un écouteur (une classe qui implémente l'interface **ActionListener**) pour chaque bouton et dans chaque classe de définir la méthode **actionPerformed** propre à chaque bouton. Cette possibilité n'est pas très pertinente. En effet, on évite certes d'avoir des conditions **if**, mais on ajoute potentiellement de nombreuses classes (une par bouton). Est-ce vraiment utile d'ajouter une nouvelle classe dans une application pour chaque bouton alors que chaque classe ne contient qu'une seule méthode (**actionPerformed**) et n'est utile que pour un seul bouton ?

Pour régler ce problème, nous pouvons utiliser des classes dites *internes*, *internes locales* et *anonymes*. Dans la section qui suit vous trouvez une explication de ces trois concepts. Par la suite, nous utiliserons une classe interne pour gérer les événement associés au bouton **effacer**, une classe interne locale pour gérer

les événement associés au bouton **defaire**, et enfin une classe anonyme pour gérer les événement associés au bouton **gommer** qui sera introduit plus tard.

2.2 Classe interne

Une classe est dite interne lorsque sa définition est située à l'intérieur de la définition d'une autre classe, comma dans l'exemple suivant :

```
1  public class ClasseNormale {
2     // Attributs, constructeurs et méthodes de la classe
        ClasseNormale
3
4     class ClasseInterne {
5        // Attributs, constructeurs et méthodes de la classe
        ClasseInterne
6     }
7
8     // Autres méthodes de la classe ClasseNormale
9  }
```

Il faut noter que cette notion de classe interne est différente de la notion d'attribut. La définition de la classe **ClasseInterne** n'introduit pas d'office d'attributs de type **ClasseInterne** dans la classe **ClasseNormale**. La définition de **ClasseInterne** est utilisable au sein de la définition de **ClasseNormale** pour instancier, quand on le souhaite, un ou plusieurs objets de ce type. Par exemple :

```
1  public class ClasseNormale {
2     private ClasseInterne ci1, ci2;
3     // Les attributs ci1 et ci2 peuvent être utilisés dans la
        classe ClasseNormale
4
5     public void maMethode () {
6        ClasseInterne i = new ClasseInterne();
7        // La variable i peut être utilisée dans la méthode maMethode
8     }
9
10    private class ClasseInterne {
11       // Attributs, constructeurs et méthodes de la classe
        ClasseInterne
12    }
13 }
```

Un objet d'une classe interne a toujours accès aux attributs et méthodes (même privés) de l'objet externe lui ayant donné naissance (il s'agit d'un accès restreint à l'objet et non à tous les objets de la classe). Un objet de la classe externe a toujours accès aux attributs et méthodes (même privés) d'un objet d'une classe interne auquel il a donné naissance. Le code suivant est donc valide :

```
1  public class ClasseNormale {
2     private int val;
3
4     public void maMethode () {
5        ClasseInterne i = new ClasseInterne();
6        i.val = 2*i.val; // on change la valeur de l'attribut de
        l'objet i
7     }
8
9     private class ClasseInterne {
10       private int val;
11
12       public updateValues() {
13          ClasseNormale.this.val = 5; // on change la valeur de
        l'attribut de la classe externe
14          this.val = 10; // on change la valeur de l'attribut de la
        classe interne
15       }
16    }
17 }
```

2.3 Classe interne locale

Il est possible de définir une classe interne directement dans le corps d'une méthode d'une classe. Dans ce cas l'instanciation d'objets de type de la classe interne ne peut se faire que dans la méthode dans laquelle la classe interne est définie. Par exemple :

```
1  public class ClasseNormale {
2     // Attributs, constructeurs et méthodes de la classe
        ClasseNormale
3
4     public void maMethode () {
5        class ClasseInterne {
6           // Attributs, constructeurs et méthodes de la classe
        ClasseInterne
7        }
8
```

```
        ClasseInterne i = new ClasseInterne();
        // La variable i peut être utilisée dans la méthode maMethode
    }

    // Ici on ne peut pas instancier d'objets de type ClasseInterne
}
```

L'intérêt des classes internes locales par rapport aux classes internes est de restreindre l'utilisation des objets de la classe interne. En effet, on ne peut instancier ces objets qu'à l'intérieur de la méthode. Si les objets du type de la classe interne ne sont nécessaires que dans une seule méthode, on évite ainsi de potentiels mauvais usages.

2.4 Classe anonyme

Avec le langage Java, il est possible de définir ponctuellement une classe sans lui donner de nom. Par exemple, si on dispose d'une classe **MaClasse**, il est possible de créer un objet d'une classe fille de **MaClasse** en utilisant une syntaxe de la forme :

```
MaClasse monObjet;
monObjet = new MaClasse() {
    // Attributs et méthodes qu'on introduit dans la classe fille
       de la classe MaClasse
};
```

Tout se passe comme ci on avait procédé ainsi :

```
MaClasse monObjet;
class MaClasseFille extends MaClasse {
    // Attributs et méthodes qu'on introduit dans la classe fille
       de la classe MaClasse
}
monObjet = new MaClasseFille();
```

Cependant, dans ce dernier cas, il serait possible de définir plusieurs références de type **MaClasseFille**, ce qui n'est pas possible dans le premier cas.

L'intérêt des classes anonymes par rapport aux classes internes locales est de restreindre encore plus l'utilisation des objets d'une classe fille d'une autre classe. En effet, avec une classe anonyme on ne peut générer qu'une seule instance d'un objet d'une classe fille d'une autre classe. La classe est dite anonyme car on ne lui donne même pas de nom (voir le premier exemple).

Notez que les classes anonymes sont également souvent utilisées pour définir une classe qui implémente une interface. On donne alors seulement les implémentations des méthodes définies dans l'interface.

Supposons que l'on dispose de l'interface suivante :

```
public interface MonInterface {
    void methodeA();
    int methodeB(double val);
}
```

Alors, si dans un code on souhaite utiliser une seule fois une classe qui implémente cette interface, on pourra alors utiliser une classe anonyme de la manière suivante :

```
MonInterface monObjet = new MonInterface() {
    @Override
    public void methodeA() {
        // implémentation de la méthode
    }

    @Override
    public int methodeB(double val) {
        // implémentation de la méthode
    }
}
```

2.5 Retour sur le TP

Question 7. Dans la classe **BarreHaute**, ajoutez une classe interne et nommez-la **EcouteurEffacer**. Nous souhaitons que cette classe implémente l'interface **ActionListener**. Pour le moment, dans la méthode **actionPerformed**, faites seulement en sorte qu'on affiche sur la console le message "*bouton appuyé : effacer*". Ajoutez ensuite un écouteur de type **EcouteurEffacer** au bouton **effacer**. Testez le bon fonctionnement de votre écouteur.

La classe interne **EcouteurEffacer** est visible dans toute la classe **BarreHaute**. Par contre vous voyez bien que nous utilisons cette classe interne

EcouteurEffacer seulement une fois dans le constructeur de la classe **BarreHaute**. On aurait donc pu définir la classe interne **EcouteurEffacer** directement dans le constructeur de la classe **BarreHaute**. On va faire cela pour le bouton *defaire*. Remarquez que cette fois ci, nous allons utiliser une classe interne *locale*, et pas seulement une classe interne comme nous l'avons fait auparavant. La classe sera donc *interne* à la classe **BarreHaute**, mais visible seulement *localement* dans le constructeur de la classe **BarreHaute**.

Question 8. Dans le constructeur de la classe **BarreHaute**, ajoutez une classe interne locale et nommez-la **EcouteurDefaire**. Cette classe doit implémenter l'interface **ActionListener**. Pour le moment, dans la méthode **actionPerformed**, faites seulement en sorte qu'on affiche sur la console le message "*bouton appuyé : defaire*". Ajoutez donc un écouteur de type **EcouteurDefaire** au bouton **defaire**. Testez le bon fonctionnement de votre écouteur.

Pour que l'appui sur le bouton **effacer** efface bien tous les dessins, et que l'appui sur le bouton **défaire** supprime le dernier dessin, vous pouvez procéder de la manière suivante :

- dans la classe **ZoneGraphique**, ajoutez deux méthodes **defaire** et **effacer** : la première enlève depuis la liste des formes la dernière dessinée ; et la seconde vide complètement la liste des formes ;
- dans la classe **Fenetre**, ajoutez deux méthodes **defaire** et **effacer** : elles font appel aux méthodes **defaire** et **effacer** de la classe **ZoneGraphique** ;
- ajoutez dans la classe **BarreHaute** un attribut de type **Fenetre** ;
- modifiez le constructeur par défaut de la classe **BarreHaute** : il prendra maintenant en paramètre un objet de type **Fenetre** ;
- faites appel à ce dernier constructeur lors de l'instanciation de l'objet **BarreHaute** dans le constructeur de votre fenêtre ;
- faites appel aux méthodes **effacer** et **defaire** dans les écouteurs sur les événements **actionPerformed** des boutons *effacer* et *defaire*.

Toutes ces modifications visent en fait à faire communiquer entre eux les objets **BarreHaute**, **Fenetre** et **ZoneGraphique** de telle sorte que l'appui sur un bouton de la barre haute modifie le dessin de la zone graphique.

Question 9. Suivez les étapes décrites ci-dessus afin que l'appui sur les boutons *effacer* et *defaire* provoque le comportement attendu. Vérifiez que tout fonctionne correctement.

2.6 Utiliser une gomme dans vos dessins

Votre application **Paint** peut effacer chaque forme que vous avez dessinée, mais l'application ne peut pas effacer une partie d'une forme (comme avec une gomme sur un dessin papier). On introduit à présent cette possibilité. Tout d'abord on ajoute dans la barre haute un bouton permettant d'activer cette fonctionnalité.

Question 10. Ajoutez dans la classe **BarreHaute** un attribut de type **JButton** et nommez-le *gomme*. Initialisez-le dans le constructeur de la classe **BarreHaute**. Affichez sur le bouton le texte "*Activer gomme*".

Pour ajouter un écouteur d'événements sur ce bouton, vous utiliserez une classe anonyme. Nous avons vu comment utiliser une classe interne et une classe interne locale pour faire cette opération. Nous avons aussi remarqué que nous utilisons les classes écouteurs une seule fois avec la création d'un seul objet. Pour ce faire, nous pouvons en fait utiliser une classe dite *anonyme*.

Question 11. Dans la classe **BarreHaute** ajoutez un écouteur d'événements au bouton *gomme*. Faites ceci à l'aide d'une classe anonyme. L'appui sur ce bouton remplace le texte "*Activer Gomme*" par le texte "*Désactiver Gomme*" et vice-versa.

Pour réaliser la gomme, il suffit en fait de dessiner des cercles remplis en blanc à chaque point où se déplace la souris. Ces cercles sont centrés autour de l'endroit où se situe la souris, avec un rayon qu'il vous revient de définir.

Pour activer le fonctionnement de la gomme vous pouvez suivre les étapes définies ci-dessous.

1. Modifiez la méthode **actionPerformed** de la classe anonyme de telle sorte que :
   - lors d'un clic sur le bouton **gomme**, vous faites appel à la méthode **activerGomme** de la classe **Fenetre** (cette dernière méthode n'est pas encore codée), et le texte affiché sur le bouton devient "*Désactiver Gomme*" ;
   - lors du clic suivant sur le bouton **gomme** (sur lequel est alors affiché "*Désactiver gomme*"), vous faites appel à la méthode **desactiverGomme** de la classe **Fenetre** (elle aussi n'est pas encore codée).
2. Ajoutez les méthodes **activerGomme** et **desactiverGomme** dans la classe **Fenetre**. Ces méthodes font appel aux méthodes **activerGomme** et **desactiverGomme** de la classe **ZoneGraphique** ;
3. Ajoutez dans la classe **ZoneGraphique** un attribut de type **boolean** et appelez-le **gommeEnCours** (cet attribut est initialisé à **false** dans le

constructeur). Ajoutez aussi les méthodes **activerGomme** et **desactiverGomme** dans la classe **ZoneGraphique**. Ces méthodes modifient la valeur de l'attribut **gommeEnCours**.

4. Modifiez la méthode **dessin** de la classe **ZoneGraphique** de telle sorte que si la gomme est active, au lieu de faire un dessin normal elle appelle la méthode **gommer** (pas encore codée).

5. Ajoutez dans le paquetage **modele** une classe **Gomme** qui hérite de **Forme**. Cette classe a un attribut de type **Point** qui représente le centre de la gomme et un attribut de type **int** qui représente le rayon de la gomme (c'est en fait un cercle centré autour du point). Nommez-les respectivement **centre** et **rayon**. Redéfinissez la méthode **seDessiner** afin de dessiner un cercle rempli de centre **centre** et de rayon **rayon** de la même couleur que celle du fond de la zone graphique (blanc en principe). Utilisez pour cela la méthode **fillOval** de la classe **Graphics**.

6. Ajoutez une méthode **gommer** dans la classe **ZoneGraphique**. Cette méthode crée un objet de type **Gomme**, le dessine dans la zone graphique, l'ajoute dans la liste des formes.

Question 12. Suivez les étapes décrites ci-dessus afin de mettre en place un fonctionnement correct du bouton gomme. Vérifiez que tout fonctionne correctement.

3 Les menus

Il est possible avec l'API Swing de doter une fenêtre de menus déroulants comme dans la plupart des applications client lourd. Vous disposez de la possibilité de créer une barre de menus qui s'affichera en haut de la fenêtre, et dans laquelle chaque menu pourra faire apparaître une liste d'options.

Nous allons voir les principes des menus déroulants en considérant le cas le plus usuel, c'est-à-dire celui où ils sont rattachés à une barre de menus.

Ces menus déroulants usuels font intervenir trois sortes d'objets :

- un objet barre de menus (objet **JMenuBar**) ;
- différents objets menu déroulant (objet **JMenu**) qui seront visibles dans la barre de menus ;
- pour chaque menu déroulant, les différents éléments, (objets **JMenuItem**), qui le constituent.

Le code suivant permet d'ajouter une barre de menus sur une fenêtre graphique, ainsi que d'y ajouter un menu déroulant avec un élément.

```
1  JFrame maFenetre = new JFrame();
2  JMenuBar barreMenus = new JMenuBar();
3  maFenetre.setJMenuBar(barreMenus);
4  JMenu couleur = new JMenu ("Couleur");
5  barreMenus.add(couleur);
6  JMenuItem rouge = new JMenuItem ("Rouge");
7  couleur.add(rouge);
```

Dans le code ci-dessus, on crée une barre de menus, puis on attache cette barre à la fenêtre graphique avec la méthode **setJMenuBar**. On crée ensuite un menu déroulant qui est ajouté sur la barre de menus avec la méthode **add**. Enfin, on crée un élément de menu qui est ajouté au menu déroulant avec la méthode **add**.

Notez que lors de la création des menus déroulants et des éléments de menus, on indique en paramètre du constructeur la chaîne de caractères qui sera affichée. Par ailleurs, les menus déroulants apparaissent sur la barre de menus dans l'ordre dans lequel ils ont été ajoutés. De même, les éléments de menus apparaissent dans le menu déroulant dans l'ordre dans lequel ils ont été ajoutés.

On peut indiquer les actions à réaliser lorsque qu'un élément de menus est sélectionné par l'utilisateur (par exemple lors du clic avec la souris), en ajoutant un écouteur sur l'objet de type **JMenuItem**. Pour cela, il faut ajouter un écouteur de type **ActionListener** dans lequel on doit implémenter la méthode **actionPerformed**.

Question 13. Ajoutez dans votre application une barre de menu avec un menu nommé "*Fichier*". Ajoutez dans le menu "*Fichier*" un élément "*Fermer*". Faites en sorte que l'application s'arrête lorsque cet élément est sélectionné.

4 Formes complexes

Nous voulons maintenant dessiner des formes composées, c'est-à-dire des formes créées à partir des formes que nous avons déjà utilisées. En particulier nous voulons dessiner des camions. Le camion doit rentrer dans la boîte englobante délimitée par les deux points lors du clic et du relâchement de la souris. Chaque partie du camion doit être proportionnelle à la dimension du cadre. Quelques exemples sont donnés dans la Figure 1.

Question 14. Ajoutez dans le type énuméré **EnumForme** l'identificateur **CAMION**. Ajoutez dans le paquetage **modele** une classe **Camion**. Utilisez les classes déjà codées pour créer les camions.

Parmi les formes que vous avez déjà définies il y a le triangle. Mais pour le moment, vous ne dessinez que des triangles isocèles. Nous voulons maintenant créer des triangles équilatéraux. Le premier point que vous sélectionnez lors du

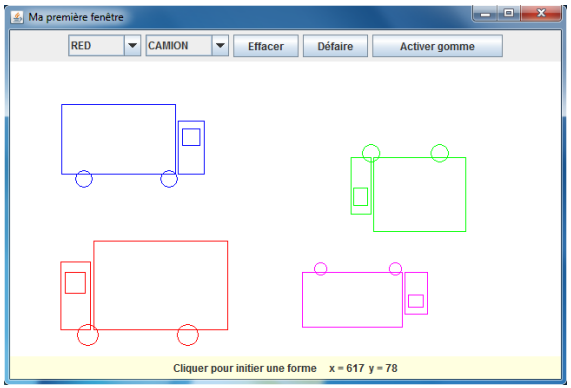

Figure 1: Exemple de dessin de camions.

clic sur le bouton de la souris est le centre du triangle. L'autre point (lors du relâchement du bouton de la souris) est une des extrémités du triangle.

Question 15. Ajoutez dans votre application **Paint** la possibilité de dessiner des triangles équilatéraux. Utilisez les coordonnées polaires pour déterminer les coordonnées des deux autres points du triangle.

References

[1] Delannoy, C., *Programmer en Java, Eyrolles, 2014*