Programmation Orientée Objet
IG2I, LE2 – 2018-2019

TP9 – Paint

Diego Cattaruzza, Philippe Kubiak, Maxime Ogier

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* ;
- la communication entre différents projets.

On s'attachera en particulier dans ce TP à commenter proprement le code et à garantir que le code fonctionne et respecte les spécifications données.

Bonnes pratiques à adopter

Pour réaliser ce TP, il vous est très fortement recommandé d'utiliser la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez. Par ailleurs, il est aussi important que votre propre code soit commenté au format Javadoc (commentaire commençant par "`/**`" au-dessus des définitions des attributs et méthodes). Ceci vous permet (1) de spécifier rigoureusement le comportement d'une méthode au moment où vous l'implémentez, (2) de savoir ce que fait une méthode sans avoir besoin de regarder son code (qui d'ailleurs peut ne pas être accessible), et donc (3) de faciliter le partage et la réutilisabilité du code.

Créez vos classes en mettant toujours les attributs en premier, suivis par les constructeurs (avec le constructeur par défaut en premier). Après les constructeurs, insérez les accesseurs et les mutateurs de la classe. Faites suivre les accesseurs et les mutateurs par toutes vos 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 a 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** dans NetBeans, ajoutez dans le paquetage **modele** une classe **Cercle**. La 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. Dans le constructeur vous initialisez les attributs. Dans la classe **Cercle**, redéfinissez la méthode **seDessiner** de la classe **Forme**. 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. Après huit séances, vous devez tous être capables d'écrire une partie de vos applications sans être guidés pas à pas.

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 dessiné : le triangle. Malheureusement, la classe **Graphics** ne possède pas une méthode "*drawTriangle*". Nous devons la créer nous même. Nous pouvons dessiner des polygones à l'aide de la méthode **drawPolygon**. 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. Modifiez les autres classes de votre code si nécessaire. 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 dans les objets graphiques.

Un bouton est déclaré de la manière suivante :

```
1  JButton monBouton;
```

Il est initialisé de la manière suivante :

```
1  monBouton = new JButton("nom bouton");
```

Un bouton avec le texte "*nom bouton*" apparaîtra alors dans l'objet graphique auquel il est associé. Pour cela, n'oubliez pas d'ajouter le bouton à un objet graphique à l'aide de la méthode **add** (exactement comme pour les **JComboBox**). De plus, vous pouvez modifier la taille des boutons de la manière suivante :

```
1  monBouton.setPreferredSize(new Dimension(largeur, hauteur));
```

Notez que, contrairement à une fenêtre, un bouton est visible par défaut ; il est donc inutile 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 noter que la méthode **actionPerformed** possède un paramètre de type **ActionEvent**, et que cette dernière classe définit une méthode **getSource**. Cette méthode fournit une référence (de type **Object**) sur l'objet ayant déclenché l'événement concerné. Ainsi, si plusieurs objets graphiques sont associés au même écouteur, il est possible dans cet écouteur de savoir quel est l'objet qui a déclenché l'appel de la méthode.

Considérons, par exemple, le code suivant :

```
1  @Override
2  public void actionPerformed (ActionEvent ae) {
3     if(ae.getSource().equals(bouton1)) {
4        System.out.println("bouton appuyé : bouton1");
5     }
6     if(ae.getSource().equals(bouton2)) {
7        System.out.println("bouton appuyé : bouton2");
8     }
9  }
```

Nous pourrions alors afficher sur la console le message "*bouton appuyé : bouton1*" si l'utilisateur a appuyé sur le bouton **bouton1**, et le message "*bouton appuyé : bouton2*" si l'utilisateur a appuyé sur le bouton **bouton2**.

Question 7. La classe **BarreHaute** implémente déjà l'interface **ActionListener**. Dans le constructeur de la classe, ajoutez à votre bouton un écouteur avec la méthode **addActionListener** en passant en paramètre **this** (qui est une instance de **BarreHaute** et donc aussi de **ActionListener**). La méthode **actionPerformed** est appelée lors de l'appui sur n'importe quel bouton (ainsi que lors de la sélection d'une valeur dans les listes déroulantes). Modifiez la méthode **actionPerformed** pour afficher un message dans la console chaque fois vous cliquez sur un bouton ou une des listes déroulantes.

Vous pouvez remarquer que ce n'est pas terrible de gérer tous les boutons avec le même écouteur (et donc dans la même méthode **actionPerformed**) : nous aurions plein de conditions **if** ou nous devrions utiliser un **switch** pour rechercher à chaque fois le bouton qui a déclenché la méthode **actionPerformed**.

Une autre possibilité est de créer une classe qui implémente l'interface **ActionListener** pour chaque bouton et donc de définir la méthode **actionPerformed** propre à chaque bouton. Cette possibilité n'est pas très pertinente : on évite certes d'avoir des conditions **if**, mais on ajoute plein de classes. Est-ce vraiment utile d'ajouter une nouvelle classe dans l'application pour chaque bouton alors que chaque classe n'est utile que pour un seul bouton ?

Pour régler ce problème nous pouvons utiliser les classes *internes*, *internes locales* et *anonymes*. Dans la section qui suit vous trouvez une explication de ces trois concepts. Lisez attentivement ces parties, avant de reprendre le code de votre application. On utilisera 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 par la suite.

2.2 Classe interne

Une classe est dite interne lorsque sa définition est située à l'intérieur de la définition d'une autre classe. Par exemple :

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

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

```
1  public class E {
2     private I i1, i2;
3     // Les attributs i1 et i2 peuvent être utilisés dans la classe E
4
5     public void f () {
6        I i = new I();
7        // La variable i peut être utilisée dans la méthode f
8     }
9
10    class I {
11       // Attributs, constructeurs et méthodes de la classe I
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.

2.3 Classe interne locale

Il est possible de définir une classe interne **I** dans une méthode **f** d'une classe **E**. Dans ce cas l'instanciation d'objets de type **I** ne peut se faire que dans la méthode **f**. Un objet de type **I** a accès aux variables locales finales de **f**. Par exemple :

```
1  public class E {
2     // Attributs, constructeurs et méthodes de la classe E
3
4     public void f () {
5        class I {
6           // Attributs, constructeurs et méthodes de la classe I
7        }
8
9        I i = new I();
```

```
10        // La variable i peut être utilisée dans la méthode f
11    }
12
13    // Ici on ne peut pas instancier d'objets de type I
14 }
```

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

2.4 Classe anonyme

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

```
1  A a;
2  a = new A() {
3     // Attributs et méthodes qu'on introduit dans la classe dérivée
4  de la classe A
5  };
```

Tout se passe comme ci on avait procédé ainsi :

```
1  A a;
2  class A1 extends A {
3     // Attributs et méthodes qu'on introduit dans la classe dérivée
4  de la classe A
5  }
6  a = new A1();
```

Cependant, dans ce dernier cas, il serait possible de définir plusieurs références de type **A1**, ce qui n'est pas possible dans le premier cas. De la même manière, les classes anonymes sont souvent utilisées pour implémenter une interface (en implémentant les méthodes définies dans l'interface).

L'intérêt des classes anonymes par rapport aux classes internes locales est de restreindre encore plus l'utilisation des objets de la classe dérivée de **A** (la classe anonyme). En effet, avec une classe anonyme on ne peut générer qu'une seule instance d'un objet de type dérivé de **A**.

2.5 Retour sur le TP

Question 8. Dans la classe **BarreHaute** ajoutez une classe interne et nommez-la **EcouteurEffacer**. Cette classe implémente l'interface **ActionListener**. Pour le moment la méthode **actionPerformed** 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 seulement une fois dans le constructeur de la classe. On aurait donc pu la définir directement dans le constructeur de la classe. On va faire cela pour le bouton *defaire*. Remarquez que nous allons alors utiliser une classe interne *locale* et pas seulement interne (elle est *interne* à la classe **BarreHaute**, mais visible seulement *localement* dans le constructeur de la classe).

Question 9. Dans le constructeur de la classe **BarreHaute** ajoutez une classe interne locale et nommez-la **EcouteurDefaire**. Cette classe implémente l'interface **ActionListener**. Pour le moment la méthode **actionPerformed** 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 10. 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 11. 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 12. 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 13. Suivez les étapes décrites ci-dessus afin de mettre en place un fonctionnement correct du bouton gomme. Vérifiez que tout fonctionne correctement.

3 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 (Monsieur Houblon sera content de le savoir). 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.

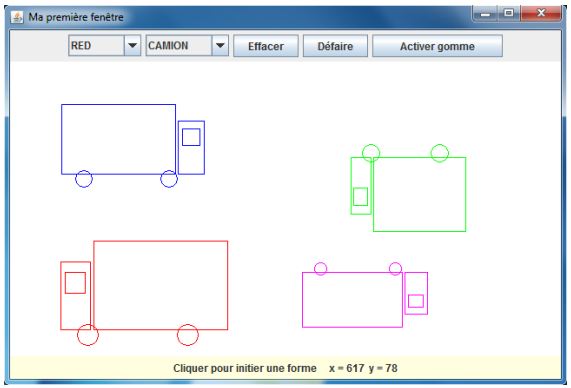

Figure 1: Exemple de dessin de camions.

4 Les menus

Il est possible avec l'API Swing de doter une fenêtre de menus déroulants comme dans la plupart des applications du commerce. 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 (objet **JMenu**) qui seront visibles dans la barre de menus ;
- pour chaque menu, les différentes options, (objets **JMenuItem**), qui le constituent.

La création d'un objet barre de menus se fait ainsi :

```
1  JMenuBar barreMenus = new JMenuBar();
```

Cette barre sera rattachée à une fenêtre par :

```
1  maFenetre.setJMenuBar(barreMenus);
```

Les différents objets menus sont créés par l'appel d'un constructeur de **JMenu**, auquel on fournit le nom du menu, tel qu'il figurera dans la barre. Chaque objet menu est ajouté à la barre à l'aide de la méthode **add** (comme pour tous les autres objets) et il apparaît dans l'ordre où il a été ajouté ; par exemple :

```
1  JMenu couleur = new JMenu ("Couleur");
2  barreMenus.add(couleur);
```

Enfin, les différentes options d'un menu sont créées par l'appel d'un constructeur de **JMenuItem**, auquel on fournit le nom de l'option telle qu'elle apparaîtra lorsque l'utilisateur fera s'afficher le contenu du menu. Chaque option est ajoutée à un menu en faisant appel à la méthode **add**. Par exemple :

```
1  JMenuItem rouge = new JMenuItem ("Rouge");
2  couleur.add(rouge);
```

Question 15. Ajoutez dans votre application une barre de menu avec un menu nommé "*Fichier*". Ajoutez dans le menu "*Fichier*" une option "*Fermer*". Sur cette option (de type **JMenuItem**), ajoutez, en utilisant une classe anonyme, un écouteur de telle sorte que l'application s'arrête lorsque l'utilisateur clique sur cette option de menu. Il faudra utiliser un écouteur de type **ActionListener** et implémenter la méthode **actionPerformed**.

5 Communication entre différents projets

Vous vous rappelez de Monsieur Houblon lors des TPs précédents ? Il vient de vous proposer un nouveau projet. Il voudrait bien pouvoir afficher ses plannings de livraison. L'application **Paint** que vous venez de coder tombe vraiment bien. Netbeans nous permet de mettre en relation différents projet et donc de pouvoir utiliser les classes de l'un dans l'autre. Par exemple, pour pouvoir faire appel aux classes du projet **Paint** dans le projet **Livraison**, vous devez ajouter dans les librairies du projet **Livraison**, la référence au projet **Paint** (ouvrez le projet **Livraison**, clic-droit sur le fichier *Libraries*), puis *Add Project*. Sélectionnez alors votre projet **Paint** et cliquez sur *Add Project JAR Files*.

Question 16. Ouvrez le projet **Livraison** et ajoutez la référence à votre projet **Paint**. Ajoutez dans la classe **Routage** (du projet **Livraison**) une méthode **visualiserPlanning**. La méthode crée un nouvel objet de la classe **Fenetre** (du projet **Paint**). Dans la méthode principale de la classe **Routage**, faites

appel à cette méthode pour pouvoir utiliser votre application **Paint** depuis le projet **Livraison**.

Question 17. Ajoutez dans votre projet **Paint**, la référence à votre projet **Livraison**. Dans la classe **Fenetre**, ajoutez un constructeur par donnée d'un objet de type **Routage**. Ajoutez dans les projets **Livraison** et **Paint** toutes les classes, méthodes, modifications nécessaires pour pouvoir donner à Monsieur Houblon un bel affichage de son planning : mettez le dépôt au centre de votre zone graphique, utilisez une forme pour le dépôt et une pour les clients, utilisez une couleur différente pour chaque tournée, etc.

6 Pour aller plus loin

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 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 18. 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*