Programmation Orientée Objet
IG2I, LE2 – 2023-2024

TP8 – Paint

Diego Cattaruzza, Maxime Ogier, Pablo Torrealba

Contexte et objectif du TP

Dans ce TP nous allons créer une application très innovante et futuriste : le *Paint* !!

Ceci nous permettra d'aborder les bases de la programmation graphique et de la programmation événementielle, ainsi que l'API Java Swing.

Ce TP permet d'illustrer les notions suivantes :

- les composants graphiques **JFrame**, **JComboBox**, **JLabel** et **JPanel** ;
- la gestion des événements et les écouteurs ;
- les interfaces **MouseListener**, **MouseMotionListener**, **ActionListener** et **WindowListener** ;
- le polymorphisme en Java ;
- les types énumérés.

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 Introduction

Dans ce TP nous allons voir ce qui distingue un programme avec une interface console d'un programme avec une interface graphique. Dans le premier cas, c'est le programme qui pilote l'utilisateur en le sollicitant au moment voulu pour qu'il fournisse des informations ; le dialogue se fait en mode texte et de façon séquentielle, dans la console. Dans le second cas, au contraire, l'utilisateur a l'impression de piloter le programme qui réagit à des demandes qu'il exprime en sélectionnant des articles de menu, en cliquant sur des boutons, en remplissant des boîtes de dialogue. Malgré l'adjectif *graphique* utilisé dans l'expression *interface graphique*, la principale caractéristique de ces programmes réside dans la notion de *programmation événementielle*.

Jusqu'ici, pour vous faciliter l'apprentissage des fondements de la programmation orientée objet, nous n'avons réalisé que des programmes à interface console. Ce TP aborde les bases de la programmation graphique avec Swing, API graphique la plus utilisée en Java.

Nous allons commencer par créer une fenêtre graphique afin de voir comment gérer les événements comme un clic dans la fenêtre, ou le déplacement de la souris. Cela permettra de présenter la notion fondamentale *d'écouteur d'événement*.

2 L'API Swing

Swing est une bibliothèque graphique pour le langage de programmation Java. Swing offre la possibilité de créer des interfaces graphiques identiques quel que soit le système d'exploitation sous-jacent. Swing utilise le principe Modèle-Vue-Contrôleur (*design pattern* MVC) : les composants Swing permettent de définir le contrôleur (avec les écouteurs d'événements) et la vue (Swing dispose de plusieurs choix d'apparence pour chacun des composants standards). Le modèle est en général à réaliser dans des classes à part.

2.1 JFrame

Pour créer une fenêtre graphique, il existe une classe nommée **JFrame**, dans le paquetage nommé **javax.swing**. Cette classe possède donc des constructeurs et des méthodes qui permettent de spécifier le contenu que l'on souhaite afficher sur la fenêtre ainsi que la manière dont on veut l'afficher.

Pour créer une fenêtre graphique très basique, on peut par exemple utiliser le code suivant :

```
1  JFrame maFenetre = new JFrame();
2  maFenetre.setSize(300, 150);
3  maFenetre.setTitle("Ma premiere fenetre !");
4  maFenetre.setLocation(100, 200);
5  maFenetre.setVisible(true);
```

Dans le code ci-dessus, on a commencé par créer un objet **maFenetre** de type **JFrame** en appelant le constructeur par défaut. Comme, par défaut, une telle fenêtre est créée avec une taille nulle, nous avons fait appel à la méthode **setSize** pour définir ses dimensions en pixels. Nous avons également défini le titre de la fenêtre (qui sera affiché dans la barre de titre) avec la méthode **setTitle**. De plus, par défaut, la fenêtre est ouverte en haut à gauche de l'écran. Pour définir son emplacement initial, nous avons fait appel à la méthode **setLocation** en passant en paramètre les coordonnées en pixels du point en haut à gauche de la fenêtre (par rapport au point en haut à gauche de l'écran). Notez que pour ouvrir la fenêtre au centre de l'écran, vous pouvez utiliser la méthode **setLocationRelativeTo** en passant **null** en paramètre. Enfin, par défaut la fenêtre n'est pas visible, c'est pourquoi nous avons appelé la méthode **setVisible** an passant **true** en paramètre.

Question 1. Après avoir ouvert IntelliJ IDEA, créez un nouveau projet **Paint**, un paquetage **vuecontrole** dans lequel on ajoutera toutes les classes liées à la vue et au contrôle des objets graphiques. Ajoutez une classe **Fenetre** dans ce paquetage. Pour le moment, dans cette classe, écrivez seulement une méthode principale (**public static void main (String[] args)**), dans laquelle vous créez une fenêtre graphique. Faites en sorte que cette fenêtre soit visible à l'écran, spécifiez sa taille, son titre et sa position. Testez le bon fonctionnement de l'affichage de la fenêtre.

Dans la question précédente, nous avons simplement créé un objet de type **JFrame** et nous avons utilisé quelques méthodes présentes dans cette classe. La classe **JFrame** définit le comportement commun à toutes les fenêtres. Si nous souhaitons créer un type de fenêtre particulier, il est alors préférable d'utiliser la notion d'héritage. Ainsi, nous créons une nouvelle classe qui hérite de **JFrame**, et dans cette classe, nous définissons seulement le comportement lié au type de fenêtre que nous souhaitons créer. Ceci nous permettra de disposer d'un type d'objet qui sera réutilisable.

Question 2. Modifiez la classe **Fenetre** afin qu'elle hérite de la classe **JFrame**. Dans le constructeur par défaut de la classe **Fenetre**, faites appel aux méthodes de la classe **JFrame** pour définir le titre de la fenêtre, définir la taille de la fenêtre, afficher la fenêtre dans une position centrale. Modifiez le contenu de la méthode principale de la classe **Fenetre**, afin de créer un nouvel objet de type **Fenetre**.

2.2 Fermeture d'une fenêtre

Pour fermer la fenêtre que vous avez créée à la question précédente vous avez certainement cliqué sur le bouton rouge en haut à droite de la fenêtre. Elle a disparu de l'écran, mais elle ne s'est pas *fermée*, elle est seulement devenue invisible (comme si l'on avait fait appel à la méthode **setVisible(false)**). Il serait donc possible de la faire réapparaître. D'ailleurs, vous pouvez remarquer que lors de l'appui sur le bouton rouge de la fenêtre, le programme ne s'est pas terminé pour autant.

On peut imposer un autre comportement lors de la fermeture de la fenêtre, en appelant la méthode **setDefaultCloseOperation** de la classe **JFrame** avec l'un des arguments suivants :

- DO_NOTHING_ON_CLOSE : ne rien faire,
- HIDE_ON_CLOSE : cacher la fenêtre (comportement par défaut),
- DISPOSE_ON_CLOSE : détruire l'objet fenêtre,
- EXIT_ON_CLOSE : quitte l'application (fait un appel à **System.exit()**).

Dans les trois premiers cas, la fermeture de la fenêtre ne met pas fin à l'application.

Question 3. Dans le constructeur de votre fenêtre, faites appel à la méthode **setDefaultCloseOperation** avec l'argument qui vous semble le plus approprié. Vous pouvez tester les différentes possibilités.

2.3 Création d'une barre en bas de la fenêtre

Maintenant nous voulons personnaliser notre fenêtre en ajoutant une barre en bas. Cette barre contiendra trois étiquettes : une pour afficher un message de bienvenue, et les deux restantes pour afficher à tout moment les coordonnées de la souris. Nous verrons par la suite comment on peut ajouter une barre dans une fenêtre, mais pour le moment, nous nous concentrons sur la création de la barre.

Une barre est un objet de type **JPanel**. Notez que de manière générale, un objet de type **JPanel** est un panneau qui contient des composants graphiques. On peut dire qu'un panneau est une sorte de "sous-fenêtre", sans titre ni bordure, et qu'il s'agit donc d'un simple rectangle qui contient d'autres composants graphiques. La classe **JPanel** dispose d'une méthode **add** qui permet d'ajouter un composant graphique (passé en paramètre de la méthode) au panneau.

Une étiquette est un objet de type **JLabel**. Il s'agit d'un composant graphique très simple qui affiche un texte court, et ne réagit à des événements externes (comme un clic avec la souris par exemple). Le texte affiché sur un objet de type **JLabel** ne peut donc pas être modifié par l'utilisateur de l'interface graphique ; seul le programme peut faire des modifications. La classe **JLabel** a un constructeur par donnée qui prend en paramètre le message à afficher.

Ainsi, si on souhaite afficher une étiquette sur un panneau, on utilisera un code semblable au code suivant :

```
1  JPanel panneau = new JPanel();
2  JLabel message = new JLabel("Joyeux Noel");
3  panneau.add(message);
```

Par ailleurs, notez que dans notre cas, nous souhaitons définir une barre qui contient trois étiquette. Il semble donc pertinent de créer une classe qui hérite de **JPanel**, et dans laquelle on gérera la création et l'ajout des trois étiquettes.

Question 4. Ajoutez une classe **BarreBasse** dans le paquetage **vuecontrole**. Cette classe hérite de la classe **JPanel**. Elle aura trois attributs de type **JLabel** pour représenter les trois étiquettes sur lesquelles on affichera un message de bienvenue, l'abscisse et l'ordonnée de la souris. Dans le constructeur par défaut de la classe, pensez à créer les trois étiquettes, à les ajouter à votre barre et à rendre cette dernière visible. Ajoutez dans la classe les accesseurs et les mutateurs nécessaires. Utilisez la Javadoc pour regarder plus précisément comment fonctionne la méthode **add** de la classe **JPanel**.

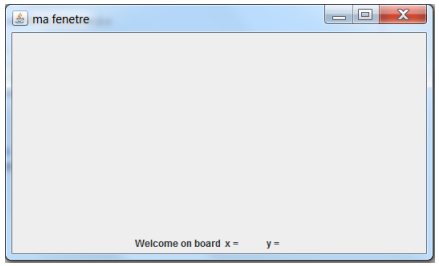
Il est possible de mettre dans une fenêtre (un objet de type **JFrame**) plusieurs panneaux (des objets de type **JPanel**). On dit que la fenêtre est un conteneur, c'est-à-dire un objet susceptible de contenir d'autres composants graphiques. Pour rappel, on peut voir une fenêtre comme un panneau avec un titre et une bordure. Par ailleurs, une fenêtre peut exister de façon autonome, alors qu'un panneau ne peut pas exister de façon autonome, et doit être associé à un conteneur (par exemple une fenêtre).

Il faut utiliser la méthode **add** de la classe **JFrame** pour ajouter un objet de type **JPanel** dans une fenêtre. De plus, un gestionnaire de mise en page (*layout manager*) permet de gérer la manière dont les composants sont agencés dans un conteneur. La méthode **add(Component comp, Object constraint)** ajoute le composant **comp** à la fin du container et avertit le gestionnaire de mise en page (*layout manager*) qu'il faut ajouter le composant en suivant les contraintes spécifiées par l'objet **constraint**. Pour la valeur de cet objet **constraint**, vous pouvez utiliser des constantes de la classe **BorderLayout**, comme par exemple **BorderLayout.NORTH**, **BorderLayout.CENTER**, ou encore **BorderLayout.SOUTH**.

Ainsi, le code suivant permet d'ajouter un panneau sur la partie haute d'une fenêtre.

```
1  JFrame maFenetre = new JFrame();
2  JPanel monPanneau = new JPanel();
3  maFenetre.add(monPanneau, BorderLayout.NORTH);
4  maFenetre.setVisible(true);
```

Question 5. Ajoutez dans votre classe **Fenetre** un attribut de type **BarreBasse**. Initialisez-le dans le constructeur de la classe **Fenetre**, et modifiez également le constructeur de telle sorte que la fenêtre comporte une barre en bas avec les informations voulues. Vous devriez obtenir un résultat similaire à celui de la figure ci-dessous.



Question 6. Utilisez la méthode **setBackground** de la classe **JPanel** afin de colorier le fond de la barre basse. Utilisez également la méthode **setForeground** de la classe **JLabel** afin de colorier les zones de texte.

2.4 Création de la zone graphique

La partie centrale de notre fenêtre sera la zone graphique, c'est-à-dire la zone que nous utiliserons pour dessiner nos œuvres d'art. Les objets de type **JPanel** permettent de réaliser cela. La couleur du panneau est initialisée avec la méthode **setBackground**. Elle prend en paramètre un objet de type **Color**.

Question 7. Ajoutez une classe **ZoneGraphique** qui hérite de la classe **JPanel**. Faites en sorte que le fond de la zone graphique soit blanc. Par ailleurs,

ajoutez dans votre classe **Fenetre** un attribut de type **ZoneGraphique**. Initialisez-le dans le constructeur de la classe **Fenetre** et ajoutez-le à la fenêtre. Testez que tout fonctionne correctement.

3 Événements

3.1 Écouteur d'événements

Ce que l'on appelle *événements de bas niveau* sont des actions physiques de l'utilisateur sur le clavier ou la souris ; c'est par exemple le cas d'un clic dans une fenêtre. Les événements de bas niveau peuvent être classifiés en quatre catégories :

- les événements liés à la souris : distinction entre appui et relâchement des boutons, identification des boutons, double clic, opérations de glisser, etc. ;
- les événements liés au clavier : distinction entre appui et relâchement d'une touche, distinction entre touche et caractère (notion de code de touche virtuelle) ;
- les événements de focalisation ;
- les événements de gestion des fenêtres.

Lorsqu'il y a un événement sur un composant graphique, cela peut déclencher une action exécutée par d'autres composants. Un composant qui crée des événements est appelé *source*. Le composant *source* délègue le traitement de l'événement au composant *écouteur*. Un composant qui traite un événement est appelé *écouteur* (*listener*). Un composant *écouteur* doit s'inscrire auprès du composant *source* des événements qu'il veut traiter.

En Java, chaque *écouteur* implémente une interface particulière correspondant à une catégorie d'événements. On parle bien ici d'interface en tant que concept de la POO et non pas d'interface graphique (vous pouvez vous référer à la Section 1.2.1 du TP 3 si vous ne savez plus ce qu'est une interface). Chaque méthode proposée par l'interface correspond à un événement de la catégorie. Ainsi, il existe une catégorie d'événements souris qu'on peut traiter avec un écouteur de souris, c'est-à-dire un objet d'une classe implémentant l'interface **MouseListener**. Cette dernière comporte cinq méthodes correspondant chacune à un événement particulier : **mousePressed**, **mouseReleased**, **mouseEntered**, **mouseExited** et **mouseClicked**.

3.2 L'interface WindowListener

Nous allons voir ici un premier exemple de gestion d'événements avec les événements sur une fenêtre graphique de type **JFrame**. Pour pouvoir recevoir et traiter les événements associés à une fenêtre, un objet *écouteur* doit implémenter l'interface **WindowListener**. Cette interface définit les méthodes suivantes :

- **windowActivated** qui est invoquée lorsque la fenêtre devient la fenêtre active ;
- **windowClosed** qui est invoquée lorsque la fenêtre a été fermée à la suite de l'appel de la méthode **dispose** ;
- **windowClosing** qui est invoquée lorsque l'utilisateur tente de fermer la fenêtre depuis le menu système de la fenêtre (appui sur la croix rouge par exemple) ;
- **windowDeactivated** qui est invoquée lorsque la fenêtre n'est plus active ;
- **windowDeiconified** qui est invoquée lorsque la fenêtre est changée d'un état d'icône à un état normal ;
- **windowIconified** qui est invoquée lorsque la fenêtre est changée d'un état normal à un état d'icône ;
- **windowOpened** qui est invoquée la première fois que la fenêtre est rendue visible.

Pour rappel, pour qu'une classe puisse implémenter une interface, vous devez le spécifier dans l'en-tête de la classe avec le mot clé **implements**, et ensuite implémenter toutes les méthodes définies dans l'interface. Ci-dessous un exemple du code :

```
1  public class EcouteurFenetre implements WindowListener {
2     // implemntation de toutes les methodes de l'interface
        WindowListener
3     ...
4  }
```

Question 8. Ajoutez une classe **EcouteurFenetre** qui implémente l'interface **WindowListener**. Ainsi, vous allez proposer une implémentation pour chacune des méthodes définies par l'interface. Vous pouvez vous contenter d'afficher sur la console un message décrivant l'événement qui vient de se produire.

Pour traiter les événements liés à la fenêtre, il suffit d'associer à un objet de type **JFrame** un objet de type **WindowListener**. On peut donc par exemple associer un objet de type **EcouteurFenetre** (qui implémente **WindowListener**) à un objet de type **Fenetre** (qui hérite de **JFrame**). Pour ce faire, nous utilisons la méthode **addWindowListener**. Cette dernière figure dans toutes les classes susceptibles de générer des événements liés à une fenêtre, en particulier la classe **JFrame**. Ainsi, le code suivant permet d'ajouter à une fenêtre un écouteur sur les événements de gestion de la fenêtre.

```
1  EcouteurFenetre objetEcouteur = new EcouteurFenetre();
2  JFrame maFenetre = new JFrame();
3  maFenetre.addWindowListener(objetEcouteur);
```

Par ailleurs, notez que, de même qu'il existe la méthode **addWindowListener** pour ajouter à une *source* un *écouteur* sur les événements de gestion de la fenêtre, il existe des méthodes pour ajouter des *écouteurs* sur les événements souris (**addMouseListener**) ou autre (allez regarder la Javadoc).

Question 9. Modifiez le constructeur de la classe **Fenetre** afin qu'un *écouteur* de type **EcouteurFenetre** soit associé aux objets de type **Fenetre**. Testez le bon fonctionnement de l'*écouteur* et vérifiez à quel moment chacune des méthodes de l'interface **WindowListener** est appelée.

3.3 Polymorphisme et écouteur d'événements

En Java, comme généralement en POO, une classe peut *redéfinir* (c'est-à-dire modifier) certaines des méthodes héritées de sa classe mère. Cette possibilité est la clé de ce que l'on nomme le **polymorphisme**, c'est-à-dire la possibilité de traiter de la même manière des objets de types différents, pour peu qu'ils soient issus de classes filles d'une même classe mère.

Plus précisément, on utilise chaque objet comme s'il était de cette classe mère, mais son comportement effectif dépend de sa classe effective (dérivée de cette classe mère), en particulier de la manière dont ses propres méthodes ont été redéfinies. Le polymorphisme permet d'ajouter de nouveaux objets dans un scénario préétabli et, éventuellement, écrit avant d'avoir connaissance du type exact de ces objets.

Java (contrairement à d'autres langages orientés objets comme C++) autorise chaque classe à hériter (avec le mot clé **extends**) d'au plus une seule classe mère. Par contre, une même classe peut implémenter (avec le mot clé **implements**) plusieurs interfaces (autant qu'on le souhaite). Une interface peut être vue comme une classe abstraite dont toutes les méthodes sont abstraites. Par ailleurs, il est important de noter qu'une classe peut hériter d'une classe *et* implémenter des interfaces *en même temps*.

Pour la gestion des *écouteurs*, Java se montre très souple puisque l'objet *écouteur* peut être n'importe quel objet dont la classe implémente l'interface voulue. Dans une situation aussi simple que celle étudiée précédemment, il est même possible de ne pas créer de classe séparée telle que **EcouteurFenetre** en faisant de la fenêtre elle-même son propre écouteur d'événements. Notez que cela est possible car la seule chose qu'on demande à un objet écouteur est que sa classe implémente l'interface voulue (ici **WindowListener**). Ainsi, on aurait très bien pu écrire cela :

```
1  public class Fenetre extends JFrame implements WindowListener {
2     public Fenetre() {
3        // Au lieu de this.addWindowListener(new EcouteurFenetre())
4        this.addWindowListener(this);
5     }
6
7     // Implémentation des méthodes de l'interface WindowListener
8  }
```

3.4 Récupération des coordonnées de la souris : l'interface MouseMotionListener

La barre basse que nous avons créée précédemment doit afficher les coordonnées de la souris (sur la partie graphique). Pour ce faire, on peut utiliser l'écouteur d'événements de mouvements de la souris. L'interface **MouseMotionListener** est l'interface d'écoute pour recevoir les événements de mouvement de la souris sur un composant. Elle définit les méthodes :

- **mouseDragged** qui est invoquée lorsqu'un bouton de la souris est pressé et que la souris est ensuite glissée ;
- **mouseMoved** qui est invoquée lorsque le curseur de la souris a été déplacé sur un composant sans qu'aucun bouton de la souris n'ait été pressé.

Ces deux méthodes prennent en paramètre un objet de type **MouseEvent**. La classe **MouseEvent** dispose de deux méthodes **getX** et **getY** (leur fonctionnalité est facile à comprendre).

Question 10. Ajoutez un attribut de type **BarreBasse** dans la classe **ZoneGraphique**. C'est cet attribut qui gérera l'affichage des coordonnées de la souris sur la zone graphique. Modifiez le constructeur de la classe **ZoneGraphique** afin que ce soit un constructeur par donnée qui prenne en paramètre un objet de type **BarreBasse** (il n'y aura donc plus de constructeur par défaut). Modifiez en conséquence le reste du code.

Question 11. Ajoutez dans la classe **BarreBasse** une méthode **public void deplacementSouris (MouseEvent evt)** qui modifie l'affichage de la barre basse (les valeurs de x et y) en fonction des coordonnées contenues dans l'objet **evt**.

Question 12. Modifiez la classe **ZoneGraphique** de telle sorte qu'elle implémente l'interface **MouseMotionListener** et qu'à chaque déplacement de la souris dans la zone graphique, la barre basse soit mise à jour avec les nouvelles coordonnées. Testez que tout fonctionne correctement et regardez comment sont calculées les coordonnées.

4 Les types énumérés

Nous souhaitons à présent dessiner sur la zone graphique plusieurs types de formes (des segments, des rectangles, des cercles, etc.) avec différentes couleurs. Pour cela, nous proposons d'ajouter une barre en haut de la fenêtre dans laquelle nous listerons les différentes couleurs possibles et les différentes formes possibles. Cette section et la suivante présentent la manière de réaliser cette barre qui liste les couleurs et les formes. La dernière section abordera la manière de réaliser les dessins.

Le langage Java permet de définir ce que l'on nomme classiquement des *types énumérés* (ou encore des *classes d'énumération*). Il s'agit de types dont les valeurs, en nombre fini, sont définies par des identificateurs. Par exemple :

```
1  public enum Jour {
2     LUNDI,
3     MARDI,
4     MERCREDI,
5     JEUDI,
6     VENDREDI,
7     SAMEDI,
8     DIMANCHE;
9  }
```

Cette énumération contient une constante pour chaque jour de la semaine.

Les constantes en Java sont normalement écrites avec toutes les lettres en majuscules. Pensez bien à écrire un code qui respecte cette convention.

Il est possible en Java de réaliser des types énumérés auxquels on associe des propriétés et des méthodes. Il est également possible de doter une classe d'énumération d'un ou plusieurs constructeurs. Dans ce cas, il faut savoir qu'un tel constructeur est appelé au moment de l'instanciation des objets constants du type. S'il nécessite des arguments, ces derniers doivent être mentionnés à la suite du nom de la constante. On notera par ailleurs que ces constructeurs doivent avoir une visibilité privée (**private**), ce qui empêche de créer des instances différentes des constantes de l'énumération. Il est possible de définir des méthodes spécifiques à l'intérieur de la classe constituée par un type énuméré.

Dans l'exemple qui suit, **Planete** est un type énuméré représentant les planètes dans le système solaire. Chaque élément est défini avec deux propriétés : la masse et le rayon de la planète. Ces valeurs sont passées au constructeur quand la constante est créée. Notez que le langage Java oblige à définir les constantes en premier, avant tous les attributs et les méthodes.

```
1  public enum Planete {
2     MERCURE (3.303e+23, 2.4397e6),
3     VENUS   (4.869e+24, 6.0518e6),
4     TERRE   (5.976e+24, 6.37814e6),
5     MARS    (6.421e+23, 3.3972e6),
6     JUPITER (1.9e+27,   7.1492e7),
7     SATURNE (5.688e+26, 6.0268e7),
8     URANUS  (8.686e+25, 2.5559e7),
9     NEPTUNE (1.024e+26, 2.4746e7);
10
11    private double masse;
12    private double rayon;
13    public static final double G = 6.67300e-11;
14
15    private Planete(double masse, double rayon) {
16       this.masse = masse;
17       this.rayon = rayon;
18    }
19
20    public double getMasse() {
21       return mass;
22    }
23
24    public double getRayon() {
25       return rayon;
26    }
27
28    public double gravite() {
29       return G * masse / (rayon * rayon);
30    }
31 }
```

Question 13. Créez un nouveau paquetage **modele**. Dans ce paquetage, créez un type énuméré nommé **EnumCouleur** (avec IntelliJ IDEA, créez une classe, puis remplacez le mot-clé **class** par **enum**). Ajoutez au moins cinq couleurs dans la liste des éléments. Chaque élément devra avoir une propriété de type **Color**.

Toutes les valeurs d'un type énuméré peuvent être récupérées dans un tableau avec la méthode statique **values()**. On peut par exemple afficher tous les noms des planètes (les constantes du type énuméré **Planete**), de la manière suivante :

```
1  Planete [ ] planetes = Planete.values();
```

```
for(int i=0; i<planetes.length; i++) {
   System.out.println(planetes[i]);
}
```

Par ailleurs, avec la méthode statique **valueOf(String val)** vous pouvez récupérer la valeur d'un type énuméré associé à la chaîne de caractères **val**, comme dans l'exemple suivant :

```
Planete maPlanete = Planete.valueOf(TERRE);
Planete autrePlanete = Planete.valueOf(MARS);
System.out.println("Gravite de surface de la terre : " +
     maPlanete.gravite());
System.out.println("Masse de mars : " + autrePlanete.getMasse());
```

5 Les listes déroulantes : JComboBox

Une *ComboBox* (ou *boite de liste déroulante*) associe un champ de texte (par défaut non éditable) et une boîte de liste à sélection simple. Tant que le composant n'est pas sélectionné, seul le champ de texte s'affiche. Lorsque l'utilisateur sélectionne le champ de texte, la liste s'affiche. L'utilisateur peut choisir une valeur dans la liste, qui s'affiche alors dans le champ de texte. L'objet Java correspondant est de type **JComboBox**.

Il existe plusieurs manières d'initialiser les valeurs des champs d'une liste déroulante. Par exemple, il est possible de passer au constructeur de **JComboBox** un tableau d'objets, et la liste sera initialisée en affichant chacun des objets du tableau (en faisant appel à la méthode **toString**) dans l'ordre dans lequel ils apparaissent dans le tableau. Ainsi, on peut écrire :

```
String [ ] couleurs = {"rouge", "bleu", "gris", "vert", "jaune", "noir"
     };
JComboBox combo = new JComboBox(couleurs);
```

Il est aussi possible d'initialiser l'objet **JComboBox** avec son constructeur par défaut, puis d'ajouter les éléments désirés avec la méthode **addItem**. Ce qui donne par exemple :

```
String [ ] couleurs = {"rouge", "bleu", "gris", "vert", "jaune", "noir"
     };
JComboBox combo = new JComboBox();
for(String s : couleurs) {
   combo.addItem(s);
}
```

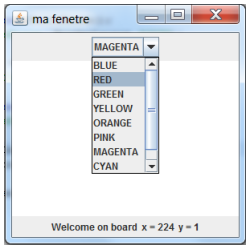
Notez bien que les items de la **JComboBox** ne sont pas nécessairement de type **String**. Comme nous n'avons rien précisé, ils sont de type **Object**. Vous pouvez donc passer en paramètre de la méthode **addItem** n'importe quel type d'objet, et l'affichage sur la liste déroulante sera réalisé en faisant appel à la méthode **toString** de l'objet.

Question 14. Ajoutez dans votre projet une classe **BarreHaute**. Comme la classe **BarreBasse**, elle hérite de **JPanel**. Ajoutez un attribut nommé **couleurs** de type **JComboBox**. Dans le constructeur, initialisez la liste déroulante avec les valeurs de l'énumération **EnumCouleur**. Pensez bien à rajouter la liste déroulante à la barre.

Testez le bon fonctionnement de votre classe **BarreHaute** en écrivant dans la méthode principale de cette classe le code suivant :

```
JFrame fenetre = new JFrame();
BarreHaute barre = new BarreHaute();
fenetre.add(barre);
fenetre.setVisible(true);
```

Question 15. Dans la classe **Fenetre** ajoutez un attribut de type **BarreHaute**. Dans le constructeur de la classe initialisez cet attribut et positionnez la barre en haut de la fenêtre. Vous devez obtenir un résultat similaire à celui dans la figure ci-dessous.



Question 16. Faites à présent une démarche similaire pour avoir dans votre barre haute une deuxième liste déroulante qui montre la liste des formes que vous allez pouvoir dessiner dans la zone graphique de votre projet. Ajoutez au moins les formes suivantes : droite, rectangle, cercle et triangle. Les noms de

ces formes seront définis dans un type énuméré nommé **EnumForme** (dans le paquetage **modele**).

Pour récupérer l'élément actuellement sélectionné dans une liste déroulante, vous pouvez utiliser les méthodes **getSelectedItem** et **getSelectedIndex** de la classe **JComboBox**. La première renvoie un objet de type **Object**. Vous pouvez faire le *cast* de l'objet si vous connaissez le type des éléments mémorisés dans la liste.

Question 17. Ajoutez dans votre classe **BarreHaute** la méthode **getCouleurSelectionnee** qui renvoie un objet de type **EnumCouleur** et la méthode **getFormeSelectionnee** qui renvoie un objet de type **EnumForme**.

Afin de tester si les deux méthodes que vous venez d'implémenter fonctionnent correctement, il faut être capable de détecter le changement de la valeur sélectionnée dans une liste déroulante. Pour ce faire, il faut un *écouteur* sur chacune des listes déroulantes. Toute action sur une liste déroulante déclenche la méthode **actionPerformed** de l'interface **ActionListener**. Il faudra donc que chacune des listes déroulantes (**JComboBox**) ait un *écouteur* sur une classe qui implémente l'interface **ActionListener**.

Question 18. Modifiez la classe **BarreHaute** de telle sorte qu'elle implémente l'interface **ActionListener**, et implémentez la méthode **actionPerformed** afin qu'elle affiche sur la console la couleur et la forme sélectionnées. Pensez bien, dans le constructeur de la classe **BarreHaute**, à doter chaque **JComboBox** d'un écouteur à l'aide de la méthode **addActionListener**. Testez le bon fonctionnement des méthodes codées à la question précédente.

Question 19. Dans la classe **ZoneGraphique**, ajoutez un attribut de type **BarreHaute**. Modifiez le constructeur par données afin qu'il prenne aussi en paramètre un objet de type **BarreHaute** qui permette d'initialiser l'attribut. Modifiez en conséquence le reste du code.

6 Allez, on dessine !

Maintenant que notre fenêtre est en place, nous voulons faire apparaître des dessins sur la zone graphique. Pour pouvoir dessiner n'importe quelle forme, nous avons besoin de déterminer les points où l'utilisateur appuie sur le bouton de la souris pour initier la forme, et où l'utilisateur relâche le bouton de la souris. Par exemple, pour dessiner une droite nous appuyons en un point sur un bouton de la souris, puis nous faisons glisser la souris et enfin nous relâchons le bouton de notre souris en un autre point de la zone graphique. La droite doit donc partir du premier point et arriver jusqu'au second.

Commençons par récupérer les événements liés à la souris dans la zone graphique. Pour ce faire, il est nécessaire d'implémenter l'interface **MouseListener** pour tout ce qui concerne les clics sur la souris, et l'interface **MouseMotionListener** pour tout ce qui concerne les mouvements de la souris. Comme la classe **ZoneGraphique** implémente déjà l'interface **MouseMotionListener**, nous pouvons décider que cette même classe implémentera également l'interface **MouseListener**. L'interface **MouseListener** contient cinq méthodes que vous devez implémenter. Pour rappel, le langage Java permet à une classe d'implémenter plusieurs interfaces (mais une classe ne peut hériter que d'une seule classe).

Question 20. Implémentez l'interface **MouseListener** dans la classe **ZoneGraphique**. Pour le moment, faites en sorte que chacune des méthodes de l'interface affiche sur la console un message descriptif de la méthode appelée (aidez-vous du nom de la méthode). Faites quelques tests afin de vous assurer de bien comprendre quel événement déclenche chacune des méthodes de l'interface.

À présent, occupons nous des objets qui représentent un point dans la zone graphique.

Question 21. Ajoutez dans le paquetage **modele** une classe nommée **Point**. Elle a deux attributs de type **int** représentant les coordonnées x et y d'un point. Ajoutez un constructeur par données des coordonnées et un constructeur par copie. Ajoutez aussi les accesseurs et mutateurs nécessaires et une méthode **toString** qui renvoie les informations sur un point. Testez que votre classe fonctionne correctement.

Question 22. Dans la classe **ZoneGraphique**, ajoutez deux attributs de type **Point** nommés **pInit** et **pFin**. **pInit** représente le point lorsque l'utilisateur appuie sur la souris (mis à jour à chaque nouvel appui). **pFin** représente le point lorsque l'utilisateur relâche la souris (mis à jour à chaque relâchement de la souris). Modifiez les méthodes des interfaces **MouseListener** et **MouseMotionListener** de la manière suivante. Lorsque l'utilisateur appuie sur la souris :

- mémorisez dans **pInit** le point où se situe le curseur de la souris ;
- affichez dans la console les coordonnées de ce point ;
- modifiez le message de la barre basse et affichez le message : "*Relâcher pour voir la forme*".

Lorsque l'utilisateur relâche la souris :

- mémorisez dans **pFin** le point où se situe le curseur de la souris ;
- affichez dans la console les coordonnées de ce point ;

- modifiez le message de la barre basse et affichez le message : "*Cliquez pour initier une forme*".

Testez le bon fonctionnement de vos méthodes.

6.1 La classe Graphics

Pour réaliser nos dessins nous allons utiliser la classe **Graphics** du paquetage **java.awt**. La classe **Graphics** est une classe abstraite. Elle dispose de toutes les méthodes voulues pour dessiner et elle gère également des paramètres courants tels que la couleur de fond, la couleur de trait, le style de trait, la police de caractères, la taille des caractères. Par exemple pour dessiner une droite nous utiliserons la méthode **drawLine**. N'hésitez pas à consulter la Javadoc !

Dans une optique de factorisation du code, nous allons maintenant créer une classe abstraite **Forme**. Elle sera la classe mère de toutes les classes que nous irons créer pour nos dessins.

Question 23. Ajoutez dans la paquetage **modele** une classe abstraite **Forme**. Ajoutez (pour le moment) un seul attribut de type **Color** (qui sera la couleur avec laquelle on dessine la forme). Ajoutez un constructeur par donnée de la couleur et une méthode **seDessiner(Graphics g)**. Elle détermine la couleur à utiliser pour le dessin en faisant appel à la méthode **setColor** sur l'objet de type **Graphics** passé en paramètre.

Question 24. Ajoutez dans le paquetage **modele** une classe **Droite**. Elle hérite de la classe **Forme**. La classe **Droite** a deux attributs de type **Point**. Ajoutez une méthode **seDessiner(Graphics g)**. Elle fait appel à la méthode **seDessiner(Graphics g)** de la classe mère **Forme**. De plus, cette méthode dessine une droite entre les deux points à l'aide de la méthode **drawLine** de la classe **Graphics**.

Question 25. Ajoutez dans la classe **ZoneGraphique** une méthode **private void dessin()**. Cette méthode crée un nouvel objet selon la forme sélectionnée dans la liste déroulante et fait appel à la méthode **seDessiner** sur cet objet afin de visualiser la forme sur la zone graphique. On pourra passer en paramètre de la méthode **seDessiner** un objet de type **Graphics** que l'on récupère grâce à la méthode **getGraphics** de la classe **JPanel** dont **ZoneGraphique** est une classe fille. Utilisez une structure de type **switch** pour créer et dessiner la bonne forme en fonction de celle sélectionnée dans la liste déroulante (pour le moment vous avez seulement la possibilité de dessiner des droites). Pensez également à traiter le cas par défaut.

Cette méthode **dessin** sera appelée lorsque l'utilisateur relâche la souris (et qu'on peut donc récupérer les coordonnées du point **pFin**). Testez que le dessin des droites fonctionne correctement, avec les différentes couleurs.

Vous avez remarqué que la droite n'apparait que lorsque vous relâchez le bouton de votre souris. Nous voulons en fait voir la droite que nous sommes en train de dessiner. Elle doit donc apparaître au cours du mouvement de glissé de la souris. Mais évidemment, seule la droite obtenue lors du relâché de la souris doit être conservée.

Question 26. Dans la classe **ZoneGraphique** implémentez la méthode **mouseDragged** de l'interface **MouseMotionListener**. À chaque appel de cette méthode, l'attribut **pFin** est mis à jour avec les coordonnées de la position de la souris, puis la méthode **dessin** est appelée. Il est très important que vous remarquiez qu'une droite est créée et affichée lors de *chaque* appel à la méthode **mouseDragged**. Testez ces modifications. Que remarquez vous ?

6.2 Mémorisation des formes dessinées

Pour résoudre le problème que vous avez dû identifier à la question précédente, nous allons introduire une sorte de mémoire dans laquelle nous garderons les formes à dessiner sur la zone graphique. Nous allons aussi ajouter dans la classe **ZoneGraphique** un attribut de type booléen qui indique si nous sommes en train de dessiner (glissé de la souris avec le bouton appuyé) ou si le dessin est terminé (bouton relâché). Nous mémoriserons dans la mémoire seulement le dernier dessin que nous avons fait, et pas tous les dessins intermédiaires lors du glissé de la souris. Par contre, si nous sommes en train de dessiner (glissé de la souris, le booléen vaut vrai), nous allons effacer la dernière forme mémorisée et ajouter la nouvelle forme correspondant au nouvel emplacement de la souris.

Ainsi, nous proposons de mémoriser dans une liste seulement les formes que nous avons dessinées lors du relâché de la souris. Nous pouvons utiliser la méthode **repaint** de la classe **Graphics**. À chaque appel, cette méthode appelle automatiquement la méthode **paintComponent**. Nous allons modifier cette dernière comme il nous conviendra.

Question 27. Ajoutez dans la classe **ZoneGraphique** un attribut de type booléen. Ajoutez aussi un attribut qui permette de mémoriser les formes. Pensez-bien à initialiser ces attributs dans le constructeur de la classe.

Question 28. Dans la classe **ZoneGraphique**, redéfinissez la méthode **paintComponent** de la classe **JPanel**. Il faut d'abord faire appel à cette même méthode dans la classe mère, puis indiquer ce qu'il faut dessiner en plus sur le panneau. Dans notre cas, il s'agit uniquement de faire appel à la méthode **seDessiner** de chacune des formes contenues dans l'ensemble des formes mémorisées. Remarquez que la méthode **paintComponent** possède un paramètre de type **Graphics**, et que c'est ce paramètre qui doit être utilisé lors de l'appel de la méthode **seDessiner**.

Question 29. Modifiez la classe **ZoneGraphique** de telle sorte que :

- le booléen qui indique si le dessin est en cours soit modifié quand il faut ;

- les formes mémorisées soient mises à jour ;
- chaque appel à la méthode **dessin** fasse appel, à la fin, à la méthode **repaint**.

Testez que tout fonctionne correctement !

References

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