

Programmation Orientée Objet

TP - 3IRC

Préambule

Méthode de travail

- Un seul sujet sert de fil conducteur à l'ensemble des séances. Il est découpé en plusieurs exercices à enchaîner au fil des séances (1 exercice ≠ 1 séance de TP). Chaque exercice est découpé en étapes.
- Chaque exercice est précédé du ou des cours nécessaires que vous aurez **étudiés avant**.
- Pour réaliser un exercice :
 - Lisez le sujet en entier avant de commencer la 1^{ère} étape pour avoir une vision globale des attendus et bien mesurer la progression,
 - Revoyez les points de cours correspondants au fur et à mesure,
 - Consultez la documentation en ligne du langage.
- Pour capitaliser vos connaissances, notez bien les réponses aux questions posées (en gras).
- Une correction sera proposée à la fin de chaque exercice pour repartir sur de bonnes bases (au cas où...).

Réalisation des TP

- Les TP s'effectuent **en binôme** sous Linux avec l'IDE de votre choix (Eclipse et NetBeans sont installés en salle TP).
- Pour réduire les temps de réponse, définissez le workspace d'Eclipse en local /var/tmp. Dans ce cas, pensez à sauvegarder régulièrement les fichiers sur votre compte perso.

Recommandations

- Faites générer par votre IDE (Eclipse, NetBeans, IntelliJ, etc.) dans chaque classe autant de méthodes que possible (kit de survie prise en main Eclipse et tests unitaires en annexe en fin de document).
- Respectez-bien les consignes et les choix de conception imposés.
- Consultez les documentations recommandées et les trucs et astuces en fin de chaque sujet d'exercice.

Documentation à consulter (outre le poly de cours) :

- <http://docs.oracle.com/javase/7/docs/api/>
- <http://docs.oracle.com/javase/tutorial/java/index.html>
- Une mine d'or avec de très bons tutos et exemples : <http://www.java2s.com/>
- Sobre et efficace, en français : http://www.jmdoudoux.fr/accueil_java.htm

Suivi pendant TP

- Les profs s'occupent des dépannages ponctuels et vérifient l'avancement du TP.
- Ils vous aident à résoudre les problèmes rencontrés en vous renvoyant si nécessaire sur le cours étudié et la documentation en ligne.

Evaluation

- Les TP ne sont pas notés puisqu'ils doivent vous permettre d'acquérir des compétences.
- Cependant, une auto-évaluation de votre niveau vous sera demandée en fin de module et sera confirmée, ou non, par les enseignants.

Introduction

Enjeux

Être capable de concevoir et développer en Java des programmes souples, extensibles et faciles à maintenir. Cela suppose de respecter les principes suivants afin de garantir une forte cohésion et un faible couplage :

- Responsabilité unique : une classe ne doit avoir qu'une seule raison de changer.
- Ouverture fermeture : une classe doit être ouverte aux extensions mais fermée aux modifications.
- Substitution de LISKOV : une méthode utilisant une référence vers une classe de base doit pouvoir référencer des objets de ses classes dérivées sans les connaître (polymorphisme).
- Ségrégation des interfaces : toute classe implémentant une interface doit implémenter chacune de ces fonctions.

Moyens

L'objectif pédagogique du TP est de mettre en œuvre, dans un programme construit au fur et à mesure des séances, les différents concepts de la Programmation Orientée Objet et de l'architecture MVC (Model-View-Controller) en les illustrant en Java :

- Abstraction et encapsulation : classe, objet, constructeur, accesseur, interfaces, etc.
- Héritage et polymorphisme : classes abstraites, classes dérivées, méthodes abstraites.
- Utilisation des collections en Java.
- Programmation événementielle et graphique.

Les algorithmes sous-jacents sont volontairement fort simples de manière à se concentrer essentiellement sur la programmation Objet.

Pédagogie

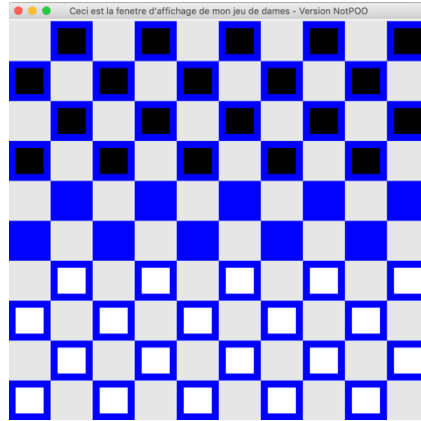
Le but du programme est de simuler un Jeu de Dames.

Contrairement à ce qui se pratique usuellement (conception et développement des classes « métier » puis développement d'une IHM (Interface Homme Machine), nous allons d'abord nous intéresser à la représentation graphique d'un damier (vue) pour aller vers le modèle (classe(s) plus métier/données) en passant par le contrôleur.

Cela nous permettra de bien comprendre l'intérêt de la conception et de la programmation orientées objets en découvrant les limites de notre 1^{er} programme qui ne respectera volontairement ni les principes objet ni l'architecture MVC et en l'améliorant au fur et à mesure des exercices.

Exercice N°1

L'objectif de cet exercice est d'afficher dans une fenêtre graphique un damier de 10*10 cases, composé de cases carrées noires et blanches (bleu et gris clair sur l'exemple), sur lesquelles sont disposées des carrés simulant les pions d'un Jeu de Dames.



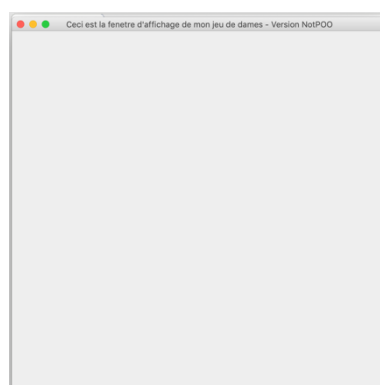
1. Créez un nouveau projet dans votre IDE (Eclipse, etc.) et créez une classe CheckersGameNotPOO dans ce projet. Complétez cette classe comme indiqué ci-dessous. Testez et constatez l'apparition du message « Ceci est mon premier programme » dans la console :

```
public class CheckersGameNotPOO {

    public static void main(String[] args) {
        System.out.println("Ceci est mon premier programme");
    }
}
```

- Le programme pourrait-il compiler/s'exécuter sans fonction main() ?
- De quel type est le paramètre args ? Dessinez la représentation que vous vous en faites avec des valeurs de votre choix.

2. Dans la fonction main() créez une fenêtre carrée de 600 pixels de côté (objet de type JFrame référencé par une variable que vous nommerez frame par exemple), donnez-lui un titre « Ceci est la fenêtre... », positionnez-la au centre de l'écran et faites la afficher. Exécutez l'application : vous obtenez une fenêtre vide (carrée...) que vous pouvez redimensionner, mettre en icône, etc.



- **Combien d'instances de la classe JFrame avez-vous créé ?**
- **Quelle instruction permet de créer un objet ?**
- **Est-ce que frame est un objet ? si non, qu'est-ce que c'est et quelle est sa valeur ? Dessinez la représentation que vous vous en faites.**

3. Complétez la fonction main() avec les lignes suivantes pour créer un damier (JPanel) et l'ajouter à la fenêtre. Testez.

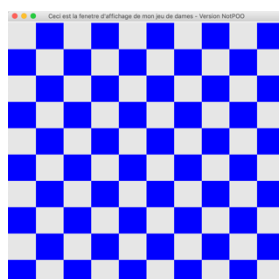
```
JPanel checkersBoard ;           // référence vers le damier
// construction et remplissage du damier
checkersBoard = new JPanel();
// Remplissage du fond du damier avec les carrés noirs et blancs
setBackgroundCheckersBoard(checkersBoard, length);
// Ajouts des pieces sur le damier - des JPanel pour l'instant
setPiecesCheckersBoard(checkersBoard, length);
frame.add(checkersBoard);
```

- **Pourquoi ces nouvelles lignes génèrent elles une erreur ?**
- **add () est-elle une fonction ou une méthode ? Pourquoi ?**
- **setBackgroundCheckersBoard() est-elle une fonction ou une méthode ? Pourquoi ?**

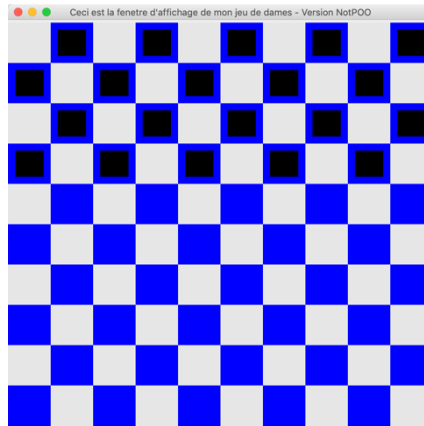
4. Survolez les erreurs avec votre souris et faites générer par votre IDE le squelette des fonctions qui manquent et testez. Le résultat devrait être le même qu'à la question 2.

5. Complétez la fonction setBackgroundCheckersBoard() pour remplir le damier de carrés (JPanel) noirs et blancs et testez (il faudra les distinguer ensuite des pions noirs et blancs, donc choisissez vos propres couleurs).

- Pour se faire, munissez le damier (JPanel) d'un Layout Manager de type GridLayout afin d'obtenir N lignes de N colonnes chacune (N = 10 dans le cas d'un Jeu de Dames).
- Chaque « cellule » est un JPanel noir ou blanc qui est ajouté au damier (JPanel principal). Attention, la « case » [0][0] est en haut à gauche de la fenêtre et non pas en bas à gauche.
- Il existe des constantes pour définir les couleurs standards dans la classe Color.



6. Complétez la fonction `setPiecesCheckersBoard()` pour, dans un 1er temps, ajouter des pions noirs (JPanel) sur les cases noires des 4 premières lignes du damier et testez.
- Les 100 JPanel créés précédemment ont été ajoutés au JPanel principal dans une liste de composants qui peuvent être retrouvés grâce à des indices `j` variant de 0 à $(N*N)-1$: `checkersBoard.getComponent(j)`;
 - Donnez une couleur non noire à vos pions « noirs » si vos cases noires sont noires... ☺



- Voyez-vous une toute petite case de couleur représentant votre pion ou alors ne voyez-vous plus la case noire (bleue) après avoir ajouté le pion ? Pourquoi ?** Jouez sur les LayoutManager pour obtenir des rendus différents mais n'y passez pas trop de temps, nous changerons de stratégie de dessin dans un prochain exercice (pour dessiner des pions ronds par exemple).
7. Complétez la fonction `setPiecesCheckersBoard()` pour ajouter des pions blancs (JPanel) sur les cases noires des 4 dernières lignes du damier et testez. Si vous vous apercevez que vous écrivez plusieurs fois les mêmes lignes de code, n'hésitez pas à écrire une nouvelle fonction avec les paramètres nécessaires pour factoriser le code...
8. Le 1^{er} exercice étant fini :
- Hormis l'objet qui correspond à l'application, créé par l'instruction** `CheckersGameNotP00` `checkersFrame = new CheckersGameNotP00();` **combien d'objets ont été créés dans ce programme ? Listez-les en détail pour les dénombrer** (1 JFrame pour la fenêtre, 1 JPanel pour le damier, etc.).
 - Hormis celle qui correspond à l'application, créé par l'instruction** `CheckersGameNotP00` `checkersFrame = new CheckersGameNotP00();` **combien d'instances de classe ont été créés dans ce programme ? Listez-les en détail pour les dénombrer si besoin.**

- **Hormis la variable créé par l'instruction** `CheckersGameNotPOO checkersFrame = new CheckersGameNotPOO();`
Combien de références ont été créés dans ce programme ?
Listez-les en détail pour les dénombrer.
En existe-t-il autant que d'objets créés ? Pourquoi ?
- **Un objet instance de la classe JFrame est-il un Component ? un Container ?**
- **Un objet instance de la classe JPanel est-il un Component ? un Container ?**
- **A quoi servent les méthodes private ? était-il utile/indispensable d'en utiliser dans ce programme ? pourquoi ?**
-
- **A quoi servent les commentaires ? En avez-vous écrits ? Pourquoi ?**

Trucs et astuces

- Ajouter un Component à un Container : `monContainer.add(monComponent)`
- Changer le Layout Manager d'un Container : `monContainer.setLayout(...)`
- Changer la couleur du fond d'un Component : `monComponent.setBackground(...)`
- Récupérer le i^{ème} composant d'un container : `monComponent = monContainer.getComponent(i)`
- Le Layout Manager d'une JFrame est BorderLayout, celui d'un JPanel est FlowLayout

Exercice N°2

L'objectif de cet exercice est de compléter l'exercice précédent de manière que lorsqu'un utilisateur sélectionne une pièce (par un clic de souris) puis clique sur une case vide, le programme déplace la pièce précédemment sélectionnée en la positionnant sur la case indiquée.

N'entrent aucunement dans ce déplacement des stratégies du Jeu de Dame (déplacement en diagonale, prise, rafles, alternance pion blanc / pion noir, etc.). Tous les déplacements sont donc possibles, sauf sur une case déjà occupée, puisque selon l'algorithme décrit plus haut, si l'utilisateur clique sur une case occupée pour indiquer une destination, en fait notre programme va considérer qu'il sélectionne une nouvelle pièce à déplacer.

Nous allons organiser le code existant de manière à le rendre objet et bénéficier ainsi de la visibilité des attributs, et nous allons créer nos propres classes pour écouter les événements déclenchés lors d'un clic de souris sur les pièces ou les cases.

Nous n'allons pas encore résoudre le pb d'affichage des pièces que l'on ne peut pas dessiner à notre convenance (cercle), ni envisager de changer leur couleur à la volée.

1. Créez un nouveau projet ou un nouveau dossier source ou un nouveau package selon la manière dont vous souhaitez organiser vos fichiers (Sauvez votre travail précédent, repartez de la correction si nécessaire).
 - a. Recopiez et réorganisez le programme de l'exercice 1 de manière à définir une classe (comme ci-dessous) avec plusieurs attributs, 1 seule méthode publique qui est le constructeur et des méthodes privées qui correspondent aux fonctions précédemment créées. Les modifications sont très minimales.
 - b. Préfixez bien tous les noms d'attributs avec « **this**. » lorsque vous les utilisez pour bien apprécier la différence de portée des attributs, des paramètres et des variables locales.
 - c. Testez. Le résultat doit être le même qu'à la fin de l'exercice 1 puisque vous n'avez effectué aucune modification dans les fonctionnalités.

```
public class CheckersGameHalfP00 {

    private JFrame frame ;           // la fenêtre du jeu
    private JPanel checkersBoard ;   // le damier
    private JPanel selectedPieceGUI; // la pièce à déplacer
    private static final int length = 10; // le nb de ligne et de colonne du damier

    public static void main(String[] args) {
        new CheckersGameHalfP00();
    }

    public CheckersGameHalfP00(){

        this.selectedPieceGUI = null;

        // construction et paramétrage de la fenêtre
        this.frame = new JFrame();
        // ...

        // construction et remplissage du damier
        this.checkersBoard = new JPanel();
        // ...

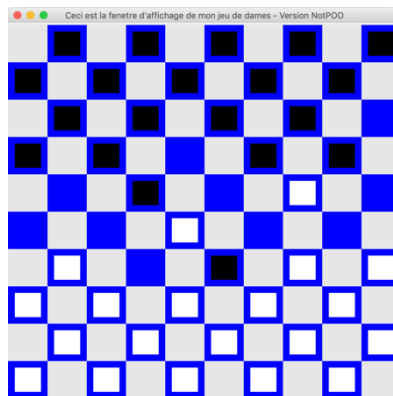
    }
    // ... Autres méthodes privées à modifier très légèrement
}
```

- **Pourquoi avons-nous écrit `new` CheckersGameHalfP00(); pour créer l'application et non pas `CheckersGameHalfP00 monApplication = new CheckersGameHalfP00();` ?**
- **Pourquoi la variable `length` est-elle définie avec les qualificatifs `static` et `final` ?**
- **Combien y-a-t-il d'attributs dans cette classe ?**
- **Pourquoi tous les attributs ont été déclarés avec le qualificatif `private` ? Aurait-on pu utiliser un autre qualificatif ? Justifiez.**
- **Pourquoi parlait-on dans l'exercice 1 de la fonction `setBackgroundCheckersBoard()` et maintenant de la méthode `setBackgroundCheckersBoard()` ?**
- **Les paramètres de vos méthodes privés ont-ils changé. Pourquoi ?**

2. Il s'agit maintenant de définir le comportement approprié pour réagir au clic de souris d'un utilisateur qui clique sur une pièce en vue de la déplacer.

- Ecrivez une méthode privée `setSelectedPiece(JPanel pieceGUI)` pour mettre à jour l'attribut `selectedPieceGUI` (1 ligne). Prévoyez temporairement un affichage de la valeur dans la console (`println()`) avant et après mise à jour de l'attribut, pour tester (à l'étape c.).
 - Ecrivez une classe privée `PieceListener` (à l'intérieur de la classe `CheckersGameHalfP00`) qui implémente l'interface `MouseListener`. Dans la méthode `mouseClicked()` invoquez la méthode `setSelectedPiece()` précédemment écrite en lui passant la valeur qui convient en paramètre (Cf. Trucs et astuces). Inutile de coder le comportement des autres méthodes de cette classe.
 - Complétez la méthode qui crée les pièces (`addPieceOnSquare()` dans la correction de l'exo1) en précisant que les pièces nouvellement créées sont écoutées par un objet du type de l'écouteur d'évènement souris que vous venez d'écrire et testez (`addMouseListener(...)`).
- **A quoi sert la méthode `getSource()` définie dans la classe `MouseEvent`?**
 - **A quoi sert la méthode `addMouseListener()` ?**
 - **Pourquoi avez-vous donné un corps « vide » à toutes les méthodes non utilisées de votre écouteur ?**
 - **A quoi sert une interface ?**
 - **Que se serait-il passé si la classe de votre écouteur avait été écrite dans un autre fichier ?**

3. La pièce à déplacer ayant été sélectionnée et sa référence sauvee dans l'attribut `selectedPieceGUI`, il faut maintenant programmer le comportement du déplacement effectif la pièce. Ce déplacement va être effectué lorsque l'utilisateur clique sur n'importe quelle case vide.
 - a. Ajoutez un écouteur d'événements souris aux cases noires et blanches dans la méthode qui les crée (`setBackgroundCheckersBoard()` dans la correction de l'exo1).
 - b. Créer une classe privée `SquareListener` (à l'intérieur ...) qui implémente l'interface `MouseListener` pour écouter ces événements. Seul le comportement de la méthode `mouseClicked()` est à redéfinir. On se contente dans cette méthode d'invoquer une nouvelle méthode `movePiece()`.
 - c. Écrivez la méthode `movePiece()` qui admet en paramètre la case de destination et qui effectue le déplacement effectif de la pièce. Il ne s'agit pas de supprimer/recréer une nouvelle pièce mais effectivement de la déplacer. La pièce déplacée ne doit plus apparaître là où elle était précédemment.
 - d. Testez successivement plusieurs déplacements.
- **Quelle méthode avez-vous invoqué pour rafraîchir l'affichage après un déplacement ?**
- **Quel enchainement d'invocation de méthodes avez-vous effectué pour réaliser le déplacement effectif de la pièce ?**

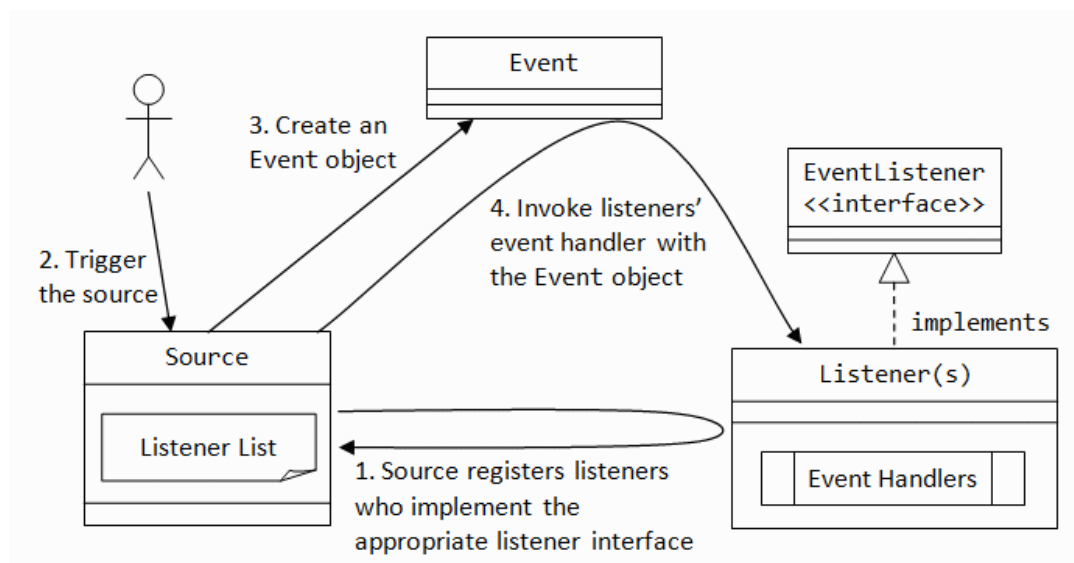


9. Le 2^{ème} exercice étant fini :

- **Finalement, ça sert à quoi un écouteur (listener) ?**
- Vous avez créé 2 classes qui implémentent les méthodes de l'interface `MouseListener`. **Mais combien d'instances de ces classes ont été créées dans le programme ?** (Si > 1, l'instruction `addMouseListener()` n'est pas écrite au bon endroit...).
- **Et en fait, c'est quoi un événement ? Pouvez-vous quantifier le nombre d'événements générés par ce programme ?**

Trucs et astuces

- Récupérer l'objet qui a donné naissance à un évènement : `ev.getSource()`
- Récupérer le Container d'un Component : `monComponent.getParent()`
- Supprimer les Component d'un Container : `monContainer.removeAll()`
- Forcer l'appel à `paintComponent()` pour redessiner un Component après modification : `repaint()`
- Enregistrer l'objet qui gèrera les évènements XXX d'un Component :
`monComponent.addXXXListener(monXXXListener)`

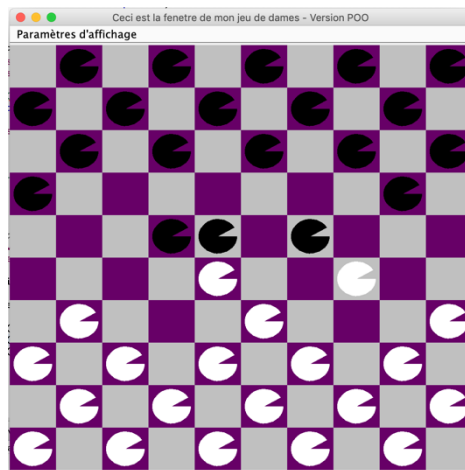


http://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html

Exercice N°3

L'objectif de cet exercice est de compléter l'exercice précédent de manière que l'utilisateur puisse donner la forme de son choix aux pièces et puisse modifier la couleur des cases au cours de l'exécution du programme. On ne travaille toujours pas sur l'algorithme de déplacement des pièces.

Notre programme va cette fois être complètement orienté objet, et nous verrons que c'est grâce à cette conception objet qu'il est très simple de modifier des données d'affichage en cours d'exécution ce qui aurait été plus difficile ou pas très élégant à faire en mode procédural.



1. Créez un nouveau projet ou un nouveau dossier source ou un nouveau package selon la manière dont vous souhaitez organiser vos fichiers. Cette fois, nous aurons plusieurs fichiers pour 1 seul programme. Le lanceur pourrait être écrit de la manière suivante. En survolant les erreurs (avec votre souris), faites générer par votre IDE la classe `CheckersGameGUI` et son constructeur.

```
public class CheckersGamePOOLauncher {

    public static void main(String[] args) {

        // Fenêtre dans laquelle se dessine le damier
        // et qui propose un menu pour changer la couleur des cases
        // et la forme des pions

        JFrame frame = new CheckersGameGUI( );

        frame.setTitle("Ceci est la fenetre de mon jeu de dames - Version POO");
        frame.setSize(600, 600);
        frame.setLocationRelativeTo(null);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

2. La classe `CheckersGameGUI` qui correspond à la fenêtre de notre application (la vue) va créer le damier et le menu de gestion des couleurs. Elle reste un objet de type `JFrame` que nous allons spécialiser : c'est donc une classe dérivée de `JFrame`. Son code pourrait être écrit de la manière suivante. En survolant les erreurs, faites générer à vide par votre IDE les classes `CheckersGameGUIBoard` et `CheckersGameGUIMenu`.

```

public class CheckersGameGUI extends JFrame {

    private JPanel checkersBoard ;           // le damier
    private JMenuBar menuBar;               // la barre de menu

    public CheckersGameGUI() {
        super();

        this.checkersBoard = new CheckersGameGUIBoard();
        this.setContentPane(checkersBoard);

        this.menuBar = new CheckersGameGUIMenu();
        this.setJMenuBar(menuBar);
    }
}

```

- A quoi sert l'instruction `super();`
- A quoi sert la méthode `setJMenuBar()` de la classe `JFrame` ?
- A quoi sert la méthode `setContentPane()` de la classe `JFrame` ?

3. La classe `CheckersGameGUIBoard` correspond au damier de notre vue. Il reste un objet de type `JPanel` que nous allons spécialiser. Son code pourrait être écrit de la manière suivante :

```

public class CheckersGameGUIBoard extends JPanel {

    private MouseListener squareListener; // l'écouteur d'évènements souris sur les carrés
    private MouseListener pieceListener;  // l'écouteur d'évènements souris sur les pièces
    private JPanel selectedPieceGUI;      // la pièce à déplacer
    private int length = 10;              // le nb de ligne et de colonne du damier

    public CheckersGameGUIBoard() {
        super();

        this.squareListener = new SquareListener(this);
        this.pieceListener = new PieceListener(this);

        this.selectedPieceGUI = null;

        // initialisation du damier
        this.setBackGroudCheckersBoard();
        this.setPiecesCheckersBoard();
    }

    // + toutes les méthodes privées
}

```

- Recopiez toutes les méthodes privées définies dans l'exercice 2 dans cette classe ainsi que les 2 classes privées qui implémentent les écouteurs et testez (nous les externaliserons plus tard). Le comportement de l'exécution doit être le même qu'à la fin de l'exo 2.

4. Nous allons maintenant customiser nos pièces :

- a. Créez dans un fichier séparé une enum pour gérer les couleurs des cases et des pièces :

```
public enum PieceSquareColor {
    BLANC, NOIR ;
}
```

- b. Dans la méthode qui crée les pièces sur les cases du damier (addPieceOnSquare()), remplacez les instructions

```
JPanel pieceGUI = new JPanel();
pieceGUI.setBackground(color); // avec color de type Color
```

par l'instruction

```
JPanel pieceGUI = new PieceGUI (pieceColor);
// avec pieceColor de type PieceSquareColor
```

Modifiez l'appel de cette méthode en conséquence dans la méthode addPiecesCheckersBoard()

Faites générer la classe PieceGUI et son constructeur par votre IDE.

Remarque : si vous aviez codé des artifices pour distinguer les pièces des cases, ils peuvent être supprimés...

- **Dans cette nouvelle configuration, pourquoi la variable pieceGUI référence-t-elle une instance de JPanel et non pas de PieceGUI ?**

- **Quel principe Objet avez-vous mis en œuvre ?**

- c. Dans la classe PieceGUI, redéfinissez la méthode paintComponent() de manière à ce que ce soit à la classe PieceGUI de décider comment les pièces sont dessinées (carré, sphère, etc.) et de quelle couleur (de type Color) en fonction de la couleur théorique passée en paramètre au constructeur (de type PieceSquareColor). Pour l'instant ce choix est définitif et l'utilisateur ne peut pas le modifier. Testez. Vérifiez que ces évolutions n'ont eu aucun impact sur le déplacement des pièces sur les cases du damier.

- **Voyez-vous toujours votre carré noir dessous ?** si non, consultez les trucs et astuces.

5. De la même manière, nous allons définir une classe SquareGUI pour ensuite customiser nos cases noires et blanches :

- a. Dans la méthode qui crée les cases du damier setBackgroundCheckersBoard()), remplacez les instructions

```
JPanel square = new JPanel(new BorderLayout());
square.setBackground(Color.WHITE);
```

par l'instruction

```
JPanel square = new SquareGUI(squareColor);
// avec squareColor de type PieceSquareColor
```

Faites générer la classe SquareGUI et son constructeur par votre IDE.

- b. Dans l'immédiat, dans la classe SquareGUI, redéfinissez la méthode paintComponent() de manière à ce que chaque carré soit dessiné par un rectangle (carré) vert entouré par une bordure noire d'1

pixel d'épaisseur. Testez. Mis à part que tous vos carrés sont temporairement verts, l'exécution doit être à iso.

6. Afin de pouvoir changer la couleur des cases du damier ou la forme des pièces en cours d'exécution nous allons utiliser un objet qui stockera ces paramètres ; il sera mis à jour lorsque l'utilisateur sélectionnera les items de menu correspondants. Les objets SquareGUI et PieceGUI devront donc l'interroger pour connaître respectivement la couleur et la forme à utiliser. Cet objet sera une table associative (Map) que l'on initialisera avec une fabrique, en dur dans un 1^{er} temps.

- a. Dans un nouveau fichier codez la fabrique comme ci-dessous.

```
public class CheckersGameGUIDataFactory {

    private CheckersGameGUIDataFactory() {
    }

    public static Map<Object, Object> createCheckersGameGUIData(){

        Map<Object, Object> map = new HashMap<Object, Object> ();
        map.put(PieceSquareColor.NOIR, Color.BLUE); // Couleur cases noires
        map.put(PieceSquareColor.BLANC, Color.LIGHT_GRAY); // Couleur cases blanches
        map.put("Taille", 10); // Taille du damier
        map.put("Forme", "Cercle"); // Forme par défaut des pièces
        return map;
    }
}
```

- Pourquoi le constructeur de cette classe est-il private ?

- b. Dans la classe CheckersGameGUI :

- Déclarez un attribut `checkersGameGUIData` de type `Map<Object, Object>`.
- Dans le constructeur, ajoutez l'instruction
`this.checkersGameGUIData = CheckersGameGUIDataFactory.createCheckersGameGUIData();`
- Passez la map en paramètre lors de la construction du damier et du menu, et modifiez en ce sens le constructeur des 2 classes concernées (+ ajout attribut de ce type dans les classes).

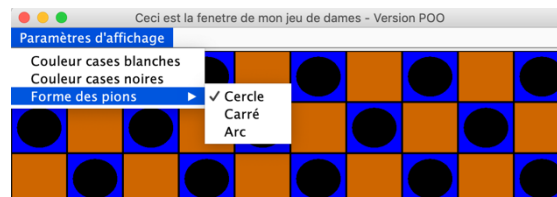
- c. Dans la classe CheckersGameGUIBoard,

- Modifiez la manière d'initialiser l'attribut `length` en allant tirer (pull) l'information de la map. Testez.
- Modifiez les appels aux constructeurs des classes SquareGUI et PieceGUI en leur passant la Map. Modifiez les constructeurs des 2 classes en conséquence. Testez. Vous devez être à iso puisque vous n'avez encore rien modifié de fonctionnel.

- d. Dans la classe SquareGUI modifier la couleur d'affichage du rectangle pour qu'elle soit fonction de sa couleur théorique (son attribut de type PieceSquareColor est l'une des clés de la map). Testez (iso).

- e. Dans la classe PieceGUI modifier la forme d'affichage de la pièce en allant tirer (pull) l'information de la map. Testez (normalement toujours à iso).

7. Il s'agit maintenant de créer un menu à partir duquel l'utilisateur pourra choisir la couleur des cases (on pourrait le faire aussi pour la couleur des pièces mais inutile de faire 4 fois la même chose, déjà 2...) et la forme des pièces :



- a. Dans la classe `CheckersGameGUIMenu` qui est une `JMenuBar`, créez un menu « Paramètres d'affichage » et 3 sous-menus « Couleur cases blanches », « Couleur cases noires » et « Forme des pions ». Ce dernier sous-menu est en fait un menu qui propose 3 options exclusives « Cercle », « Carré » et « Arc ». Testez et vérifiez que vous pouvez dérouler les menus.

- Si l'utilisateur sélectionne un item de menu, se passe-t-il quelque chose ? est-ce normal, pourquoi ?

- b. Ajoutez un écouteur d'évènement `Action` pour mettre à jour la couleur qui correspond à la clé `PieceSquareColor.BLANC` de la map passée en paramètre au constructeur. Faites choisir la bonne couleur à l'utilisateur en utilisant un objet de type `JColorChooser`. Avant de tester, prévoyez un `println()` pour afficher le contenu de la map avant et après le changement de couleur. Testez.

- Les données de la map ont-elles été modifiées ?

- Les couleurs de vos cases sont-elles changées ?

- Si non, déplacez une pièce à l'aide de la souris. Les couleurs de vos cases sont-elles changées ? Que s'est-il passé ?

- c. Avant de corriger ce problème, programmez au moins un écouteur pour modifier la forme des pièces. Modifiez la classe `PieceGUI` en conséquence (if « carré » ... `fillRect()`, if « cercle » ... `fillOval()`, etc. On apprendra une solution plus générique plus tard). Testez (en déplaçant une pièce).

8. Pour corriger le problème, nous allons mettre en place l'un des Design Pattern les plus utilisés, le pattern `Observer`. Étudiez [https://fr.wikipedia.org/wiki/Observateur_\(patron_de_conception\)](https://fr.wikipedia.org/wiki/Observateur_(patron_de_conception)) (c'est très court ☺). Comme dans l'exemple, nous allons nous appuyer sur le mécanisme `observer/observé` mis en place dans le JDK (package `java.util`).

- Auriez-vous déjà utilisé ce pattern sans le savoir ? Si oui quand ?

Dans notre application, l'objet observé est la map qui contient les paramètres d'affichage, le ou les observateurs sont les éléments de la vue. Autant, il va vous être facile d'imposer que des classes que vous avez personnellement écrites implémentent l'interface `java.util.Observer`, autant, il va être impossible d'imposer que la classe `HashMap` (classe concrète de notre map – Cf. Fabrique) définie dans le JDK étende la classe `java.util.Observable`.

Et pourtant, il nous faut bien une « map observable » sinon l'utilisateur ne va pas comprendre pourquoi le programme ne réagit pas à ses actions... Et vous savez quoi, c'est exactement ce que nous allons programmer !

Une map, avec le comportement attendu d'une map et un petit quelque chose en plus pour la rendre observable.

Nous allons simplement « décorer » une Map (mise en œuvre du Design Pattern Decorator) :

- Dans un 1^{er} temps, occupons-nous de l'observer. En toute logique, si vous avez compris l'exemple étudié, les observateurs sont soit les pièces soit les cases. Pour minimiser l'impact sur les classes existantes, il est plus facile de rendre la classe CheckersGameGUI observateur et de dire que la vue complète se redessine à chaque modification (il est clair que ce n'est pas la solution la plus rapide en temps d'exécution ; en général, on l'évite...). Modifiez le programme en ce sens. Testez.
- Dans un nouveau fichier, codez le décorateur comme ci-dessous. Un décorateur est un objet qui implémente la même interface que l'objet qu'il contient (Map).

```
public class CheckersGameGUIData extends Observable implements Map {

    private Map<Object, Object> mapGUIData;

    public CheckersGameGUIData (Map<Object, Object> mapGUIData) {
        super();
        this.mapGUIData = mapGUIData;
    }
}
```

- Survolez les erreurs avec votre souris et faites générer par votre IDE les méthodes nécessaires. Un décorateur appuie son comportement sur celui de l'objet qu'il enveloppe et l'enrichit. Modifier toutes les méthodes suivant l'exemple suivant :

```
public int size() {
    setChanged();
    notifyObservers();
    return mapGUIData.size();
}
```

- Modifiez la fabrique pour qu'elle retourne une instance de CheckersGameGUIData et non plus de HashMap. Testez.

- Pour minimiser l'impact de la mise en place du modèle (Exercice N°4), nous allons achever la vue pour ne pas y revenir par la suite.

- Munissez la classe SquareGUI d'un Id qui correspond à l'ordre de création des objets : le 1^{er} carré créé porte le N°1, le 2^{ème} le N° 2, etc. Réalisez cette évolution sans modifier le constructeur de la classe (Pensez attribut de classe). Prévoyez un accesseur sur cet attribut.

- Pourquoi ne faut-il pas prévoir un accesseur sur cet attribut ?**

- Ce sont les écouteurs qui vont communiquer avec le modèle (Est-il possible de déplacer la PieceGUI sélectionnée lors du clic de la souris, est-il possible de la déposer sur le SquareGUI sélectionné, etc.). Pour éviter de modifier la classe qui s'occupe de l'affichage (CheckersGameGUIBoard), nous allons externaliser les 2 écouteurs d'événements « souris » dans une autre classe.
 - Dans un nouveau fichier, codez une classe CheckersGameGUIBoardListeners. Copiez-y les 2 écouteurs (SquareListener et PieceListener). Inutile de chercher à tester pour vous apercevoir que les méthodes movePiece() et setSelectedPiece() ne sont plus accessibles. Munissez chaque écouteur d'un attribut de type CheckersGameGUIBoard afin d'invoquer ses méthodes comme montré ci-dessous.


```

public class CheckersGameGUIBoardListeners {

    public class SquareListener implements MouseListener {
        private CheckersGameGUIBoard checkersGameGUIBoard;
        public SquareListener(CheckersGameGUIBoard checkersGameGUIBoard) {
            super();
            this.checkersGameGUIBoard = checkersGameGUIBoard;
        }
        public void mouseClicked(MouseEvent ev) {
            this.checkersGameGUIBoard.movePiece((JPanel) ev.getSource());
        }
        // ...
    }
    public class PieceListener implements MouseListener {
        // ...
    }
}

```

- Munissez le constructeur de la classe CheckersGameGUIBoard, d'un paramètre de type CheckersGameGUIBoardListeners et remplacez les instructions

```

this.squareListener = new SquareListener(this);
this.pieceListener = new PieceListener(this);

```

par les instructions

```

this.squareListener = checkersGameGUIBoardListener.new SquareListener(this);
this.pieceListener = checkersGameGUIBoardListener.new PieceListener(this);

```

- Modifiez la création du damier en conséquence dans la classe CheckersGameGUI qui a la responsabilité de créer l'objet de type CheckersGameGUIBoardListeners. Testez.

10. Le 3^{ème} exercice étant fini :

- **Quels sont les différents types d'EventListener utilisés dans ce programme ?**
- **Que fait l'instruction pieceGUI.addMouseListener(monObjetEcouteur) ? Peut-on considérer que l'utilisation d'un EventListener est une mise en œuvre du Design Pattern Observer ?**
- **Qu'est-ce qui justifiait l'usage de l'héritage dans cet exercice ?**
- **Quel(s) intérêt(s) y avait-il à découper le programme en autant de classes ?**

- Êtes-vous capable de donner une définition de : interface, classe, objet, référence, classe dérivée ?
- Quels principes objet avez-vous mis en œuvre dans cet exercice ? Illustrez.
- Êtes-vous capable de dire ce qu'est une abstraction et en quoi consiste l'encapsulation, à quoi ça sert ?

Trucs et astuces

- L'argument de la méthode `paintComponent()` est un objet `Graphics` qui vous fournit la surface sur laquelle dessiner (en réalité `Graphics2D`)
- Dessiner sur cette surface : `g.setColor(...)`; ou `g.setPaint(paint)` pour fixer la couleur ou le motif puis `g.fillRect(...)`; pour dessiner,
Avec : `Paint paint = new GradientPaint(...)`;
- Encadrer un `Component` : `monComponent.setBorder(BorderFactory.createLineBorder(...))`
- Indiquer qu'un `Component` est transparent (et voir celui en dessous) : `monComponent.setOpaque(false)`
- Utiliser des menus : des `JMenu` composés de `JMenuItem` le tout stocké dans une `JMenuBar`. Certains `JMenuItem` sont `JMenu` ou des `JRadioButtonMenuItem`.
- Ouvrir une boîte de dialogue pour choisir une couleur : `JColorChooser.showDialog(...)`
- Ouvrir une boîte de dialogue pour saisir une valeur : `JOptionPane.showInputDialog(...)`
- Comparer les valeurs de 2 objets : `refVersObj1.equals(refVersObj2)` avec `refVersObj1 != null`
- Écrire dans une `Map` : `maMap.put(key, value)`. Si la valeur existait déjà, elle est remplacée.
- Lire dans une `Map` : `maMap.get(key)`. Retourne la valeur si la clé n'existe, `null` sinon.

Exercice N°4

A la fin de l'exercice 3, notre programme est muni d'une vue. Reste à lui ajouter une partie « métier » pour gérer les algorithmes de déplacement et de prises des pièces. Étant donné les faibles nombre et types d'interactions de l'utilisateur (clic de souris) dans cette application, un contrôleur n'était pas vraiment indispensable.

Pour en justifier l'existence nous avons joué sur la façon dont la vue et le modèle gèrent les coordonnées des pièces sur le damier : N° (de 1 à 100) du SquareGUI sur lequel est posée la PieceGUI côté vue, coordonnées de type ligne/colonne côté modèle.

Dans un premier temps, seuls le déplacement en diagonal et la prise d'1 seule pièce du jeu opposé seront implémentés (pas de rafle, pas de vérification qu'il n'y a pas de prise possible pour qu'une pièce puisse être jouée, etc.).

A la fin de l'exercice N°3, notre programme respectait déjà un certain nombre de principes objets :

- Responsabilité unique : une classe ne doit avoir qu'une seule raison de changer.
 - Nous avons organisé la vue (checkersGameGUI) dans des fichiers distincts et ainsi séparé la gestion du damier (CheckersGameGUIBoard) de celle du menu (CheckersGameGUIMenu), puis de la gestion des pièces (PieceGUI) et des cases (SquareGUI). Ces dernières peuvent être réutilisées dans d'autres contextes, sous réserve que leurs données de configuration soient stockées dans une Map...
 - Nous avons séparé la vue des données qu'elle manipule (checkersGameGUIData) même si pour l'instant, les objets de la vue doivent connaître les valeurs des clés de la Map pour fonctionner ; mais cela pourrait être rendu plus générique.
- Substitution de LISKOV : une méthode utilisant une référence vers une classe de base doit pouvoir référencer des objets de ses classes dérivées sans les connaître (polymorphisme).
 - Les objets de type checkersGameGUI, CheckersGameGUIBoard, PieceGUI et SquareGUI restent des JComponent ce qui permet d'invoquer leur méthode paintComponent() et de rafraîchir l'affichage sans tester leur type réel.
- Ouverture/Fermeture : Si l'on ajoute du comportement à la vue, par exemple en créant une nouvelle classe qui afficherait dans un autre panneau, la trace des coups joués, les scores à l'instant t, le nom du joueur gagnant, etc. cela n'aurait aucun impact sur les classes existantes).

Le respect de ces principes nous a permis d'isoler les modifications et de ne pas faire de tests de non-régression sur des parties déjà testées précédemment, en particulier lors de la mise en place du pattern Observer. Notre programme est assez souple et semble extensible et facile à maintenir.

Comme nous avons bien découpé notre application, les classes qui gèrent les couleurs et le damier/carrés/pièces ne devraient subir aucune modification lors de développement du modèle ! Néanmoins, nous ajouterons au damier une méthode qui permettra de supprimer visuellement une pièce en cas de prise.

Seuls les écouteurs liés aux déplacements des pièces seront modifiés puisqu'ils doivent maintenant invoquer les méthodes du modèle par l'intermédiaire du contrôleur. Le code de retour de ces méthodes sera exploité par les écouteurs pour communiquer au damier exactement ce qu'il doit faire (déplacer visuellement la pièce ou non dans le cas où le déplacement ne serait pas « légal »).

1. Créez un nouveau projet ou un nouveau dossier source ou un nouveau package selon la manière dont vous souhaitez organiser vos fichiers.
 - a. Créez plusieurs packages : model, controler, gui, launcher, checkers (pour stocker les classes partagées par plusieurs packages). Copiez temporairement les fichiers de l'exercice 3 dans le package model. Déplacez le lanceur dans le package launcher, la classe PieceSquareColor dans le package checkers et tous les autres fichiers dans le package gui. Testez.

- b. Le lanceur pourrait être écrit de la manière suivante. En survolant les erreurs (avec votre souris), faites générer les classes `CheckersGameModel`, `CheckersGameController` et leurs constructeurs. Modifiez le constructeur de la classe `CheckersGameGUI`. Testez.

```
public class CheckersGameMVCLauncher {

    public static void main(String[] args) {
        // Objet qui gère les aspects métier du jeu de dame :
        // déplacement en diagonale, avec ou sans prise pour l'instant
        CheckersGameModel checkersGameModel = new CheckersGameModel();

        // Objet qui contrôle les actions de la vue et les transmet au model
        // il renvoie les réponses du model à la vue
        CheckersGameController checkersGameController = new
            CheckersGameController(checkersGameModel);

        // Fenêtre dans laquelle se dessine le damier et qui gère le menu
        JFrame frame = new CheckersGameGUI(checkersGameController );

        // configuration habituelle de la fenêtre ...
        frame.setVisible(true);
    }
}
```

2. Nous allons mettre en place le mécanisme de communication entre la vue, le controler et le model.

- a. Munissez la classe `CheckersGameModel` d'une méthode `isPieceMoveable()` comme ci-dessous et faites générer la classe `Coord` que vous rangerez dans le package `checkers`. Cette méthode sera chargée de vérifier que l'utilisateur peut bien déplacer la pièce qui se trouve aux coordonnées passées en paramètre. Dans un 1^{er} temps, nous considérons que oui (`return true`). Recopiez aussi la méthode `getLength()` ci-dessous qui donne le nb de lignes/colonne du model.

```
public int getLength() {
    return 10; // cette ligne sera modifiée ultérieurement
}

public boolean isPieceMoveable(Coord coord) {
    boolean bool = false;
    bool = true; // cette ligne sera modifiée ultérieurement
    return bool;
}
```

- b. L'écouteur de `PieceGUI` (`PieceGUIListener`) ne peut pas invoquer directement cette méthode car il ne parle pas le même langage que le model (`int` vs `Coord` pour gérer l'emplacement des pièces). Il faut donc qu'il s'adresse au controler qui propagera le message au model après transformation des coordonnées. Munissez la classe `CheckersGameController` d'une méthode `isPieceMoveable()` comme ci-dessous.

```
public boolean isPieceMoveable(int squareIndex) {
    boolean bool = false;
    Coord targetCoord =
        this.transformIndexToCoord(squareIndex,
                                   this.checkersGameModel.getLength());
    bool = this.checkersGameModel.isPieceMoveable(targetCoord);
    return bool;
}
```

Avec :

```
private Coord transformIndexToCoord (int squareIndex, int length) {
    Coord coord = null;
    // coord = cette ligne sera modifiée ultérieurement
    return coord;
}
```

- c. Dans la classe PieceGUIListener, modifiez la méthode mouseClicked() comme ci-dessous. Testez.

```
public void mouseClicked(MouseEvent ev) {

    // Recherche PieceGUI sélectionnée
    JPanel piece = (JPanel) ev.getSource();

    // Recherche coordonnée (Id) du carré qui contient la pièce
    SquareGUI square = (SquareGUI) piece.getParent();
    int indexSquare = square.getIdSquareGUI();

    // Si le déplacement est OK pour le model, la PieceGUI de la vue est fixée
    if (checkersGameControler.isPieceMoveable(indexSquare)) {
        this.checkersGameGUIBoard.setSelectedPiece((JPanel) ev.getSource());
    }
}
```

- **Le comportement du programme à l'exécution est-il bien le même que précédemment ?**

3. Pour tester réellement notre mécanisme de communication, et avant de définir les classes métier, il faut définir la classe Coord et programmer la méthode transformIndexToCoord() de la classe CheckersGameControler.

- a. Munissez la classe Coord des 2 attributs ci-dessous, d'un constructeur, de leurs getter et setter et d'une fonction main() pour la tester rapidement :

```
private char colonne;
private int ligne;
```

- b. Dans la fonction main(), saisissez les instructions suivantes et testez.

```
Coord c1 = new Coord('a', 7);
Coord c2 = new Coord('a', 7);
System.out.println("c1 = " + c1);
System.out.println("c2 = " + c2);
System.out.println("c1.equals(c2) ? "+ c1.equals(c2));
```

- **Que pensez-vous des résultats obtenus ?**

- c. Faites générer par votre IDE la méthode toString() qui retourne une représentation des attributs et la méthode equals() qui vérifie que les 2 attributs sont identiques, et testez à nouveau.

- **Que pensez-vous des résultats obtenus ?**

- **A quoi sert la méthode toString() ? l'avez-vous appelée explicitement ?**

- **Pourquoi, lorsque vous avez fait générer la méthode equals(), la méthode hashCode() a-t-elle été générée automatiquement ? à quoi sert-elle ?**

- d. Ajoutez dans cette classe 2 méthodes (signatures ci-dessous), qui vérifient que des coordonnées sont dans les limites de la borne max. Testez.

```
public static boolean coordonnees_valides(char col, int ligne, int max)
public static boolean coordonnees_valides(Coord coord, int max)
```

- **Pourquoi ces méthodes sont-elles définies avec le qualificateur static ?**

- e. Modifiez la méthode transformIndexToCoord() de la classe CheckersGameController pour qu'elle retourne le résultat attendu. Testez (ajoutez un println()).

4. Afin de pouvoir enfin tester réellement nous devons finir de définir la partie métier (classe CheckersGameModel). Nous décidons de ne pas stocker des « cases » vides ou non, car elles n'ont aucun comportement particulier. Nous choisissons de ne manipuler que des pièces (de type PieceModel) qui connaîtront leur couleur et leur position sur le damier (Coord), et surtout qui sauront comment se déplacer sur un damier.

- a. Dans de nouveaux fichiers, codez l'interface PieceModel comme ci-dessous, une classe AbstractPieceModel qui implémente PieceModel et 2 classes dérivées de AbstractPieceModel PawnModel et QueenModel qui auront des comportements différents pour tester si un déplacement et une prise sont autorisés. Munissez la classe AbstractPieceModel de 2 attributs respectivement de type Coord et PieceSquareColor. Complétez le corps des méthodes de la classe AbstractPieceModel. Ajoutez (faites générer) un constructeur qui initialise les attributs dans les 3 classes. Ajoutez une méthode toString() et une méthode equals() là où c'est nécessaire. Dans une fonction main() dans la classe PawnModel vérifiez rapidement que vous êtes capable de créer et d'afficher un PawnModel.

```
public interface PieceModel {
    public Coord getCoord() ;
    public void move(Coord targetCoord);
    public PieceSquareColor getPieceColor() ;
    public boolean isMoveOk(Coord targetCoord, boolean isPieceToTake);
}
```

- **Pourquoi ne pas prévoir de méthode setPieceColor().**
- **Pourquoi appeler la méthode move() plutôt que setCoord() ?**
- **Pourquoi la classe AbstractPieceModel doit-elle être abstraite ?** (ne répondez pas « parce qu'elle contient des méthodes abstraites »...)
- **Était-il obligatoire de la créer ? si non, quels auraient été les conséquences ?**
- **Était-il obligatoire de créer l'interface PieceModel ? si non, quels auraient été les conséquences ?**

- **Qu'est-ce qu'une méthode abstraite ?**

- Dans la classe AbstractPiece l'attribut Coord est déclaré avec le qualificateur private alors que la méthode isMoveOk() des classes dérivées a besoin de travailler avec ses valeurs et doit passer par la méthode getCoord() ? **Aurait-il fallu le déclarer protected ? Justifiez.**

b. Nous allons initialiser les données du modèle à partir d'une fabrique à coder comme ci-dessous.

```
public class CheckersGameModelFactory {

    private CheckersGameModelFactory() {
    }
    /**@return une map de Coord pour alimenter la liste de PieceModel du model*/
    public static Map<PieceSquareColor, List<Coord>> getCheckersGameModelCoords(){

        Map<PieceSquareColor, List<Coord>> map =
            new HashMap<PieceSquareColor, List<Coord>> ();
        Coord[] blackPieceCoords = new Coord[] {
            new Coord('b',10),new Coord('d',10),new Coord('f',10),new Coord('h',10),new Coord('j',10),
            new Coord('a',9), new Coord('c',9), new Coord('e',9), new Coord('g',9), new Coord('i',9),
            new Coord('b',8), new Coord('d',8), new Coord('f',8), new Coord('h',8), new Coord('j',8),
            new Coord('a',7), new Coord('c',7), new Coord('e',7), new Coord('g',7),new Coord('i',7),};
        Coord[] whitePieceCoords = new Coord[] {
            new Coord('b',4), new Coord('d',4), new Coord('f',4), new Coord('h',4), new Coord('j',4),
            new Coord('a',3), new Coord('c',3), new Coord('e',3), new Coord('g',3), new Coord('i',3),
            new Coord('b',2), new Coord('d',2), new Coord('f',2), new Coord('h',2), new Coord('j',2),
            new Coord('a',1), new Coord('c',1), new Coord('e',1), new Coord('g',1),new Coord('i',1),};
        map.put(PieceSquareColor.NOIR, Arrays.asList(blackPieceCoords));
        map.put(PieceSquareColor.BLANC, Arrays.asList(whitePieceCoords));
        return map;
    }
    public static int getlength(){
        return 10;
    }
    public static PieceSquareColor getBeginColor(){
        return PieceSquareColor.BLANC;
    }
}
```

c. Dans la classe CheckersGameModel.

- Ajoutez les attributs ci-dessous.

```
private List<PieceModel> pieceList; // la liste de pièces noires et blanches
private int length; // le nombre de lignes et colonnes du damier
private PieceSquareColor currentColor; // la couleur du joueur courant
```

- Dans le constructeur, initialisez les attributs length et currentColor à partir de la fabrique.
- Dans le constructeur, implémentez la liste à l'aide d'une LinkedList<PieceModel>. Créez autant de PawnModel que le suggère la fabrique et stockez les dans la liste.
- Recopiez la méthode toString() suivante. Ajoutez l'instruction System.out.println(this); à la fin du constructeur et testez l'application : par la suite, il faudra faire afficher le damier en mode console pour chaque déplacement et vérifier ainsi que vos données métiers sont bien cohérentes avec ce qui s'affiche sur la GUI.

```

public String toString() {
    String[][] damier = new String[this.length][this.length];

    // création d'un tableau 2D avec les noms des pièces
    for(PieceModel piece : this.pieceList) {
        PieceSquareColor color = piece.getPieceColor();
        String stColor =
            (PieceSquareColor.BLANC.equals(color) ? "--B--" : "--N--" );

        int col = piece.getCoord().getColonne()-'a';
        int lig = piece.getCoord().getLigne() -1;
        damier[lig][col ] = stColor ;
    }
    // Affichage du tableau formaté
    String st = "   a   b   c   d   e   f   g   h   i   j\n";
    for ( int lig = 9; lig >=0 ; lig--) {
        st += (lig+1) + " ";
        for ( int col = 0; col <= 9; col++) {
            String stColor = damier[lig][col];
            if (stColor != null) {
                st += stColor + " ";
            }
            else {
                st += "----- ";
            }
        }
        st += "\n";
    }
    return "Damier du model \n" + st;
}

```

- Aurait-on pu implémenter la liste de PieceModel dans une ArrayList ? Justifiez.
- Aurait-on pu choisir un Set pour stocker les PieceModel ? Justifiez.
- Ce Set aurait-il pu être implémenté dans un HashSet, un TreeSet ? Justifiez.
- Ce Set aurait-il pu être implémenté dans un TreeSet ? Justifiez.

- d. Dans la classe CheckersGameModel modifiez la méthode isPieceMoveable() de manière à ce qu'elle retourne true uniquement dans le cas où la couleur de la PieceGUI sélectionnée par l'utilisateur est la même que celle du joueur courant (ses Coord sont passées en paramètre). Testez. Normalement, sur la GUI, vous ne devriez pouvoir déplacer que des pièces blanches (uniquement celles que vous n'avez pas encore déplacées...).

5. Nous allons maintenant nous occuper du déplacement effectif.

- a. Dans la méthode `mouseClicked()` de la classe `SquareListener`, l'invocation de la méthode `movePiece()` sur la vue (`CheckersGameGUIBoard`) ne doit se faire que s'il est possible de déplacer la pièce aux coordonnées correspondantes dans le model. Comme précédemment, l'écouteur interroge le controler, qui interroge le model. Si le model répond favorablement, l'écouteur indique à la vue et au controler que le déplacement peut être effectif. Ce dernier prévient alors le model. Commencez par tester le workflow avant de travailler sur les méthodes du model :

- Munissez la classe `CheckersGameModel` des méthodes suivantes en leur faisant respectivement retourner, pour l'instant, `true` et `-1`. La valeur de retour de la méthode `movePiece()` sera exploitée par la vue pour effacer la pièce capturée.

```
public boolean isMovePieceOk(Coord initCoord, Coord targetCoord)
public Coord movePiece(Coord initCoord, Coord targetCoord)
```

- Munissez la classe `CheckersGameController` des méthodes suivantes :

```
public boolean isMoveTargetOk(int squareIndex)
public int movePiece(int squareIndex)
private int transformCoordToIndex (Coord coord)
```

- Ajoutez une méthode `void removePiece(JPanel removePieceSquare)` dans la classe `CheckersGameGUIBoard`.
 - Programmez l'enchaînement des invocations de méthodes dans les différentes classes. Testez. Vous devriez avoir le même résultat que précédemment (déplacement uniquement des pièces blanches).
- b. Programmez la méthode `isMovePieceOk()` de la classe `CheckersGameModel` qui s'appuie sur la méthode `isMoveOk()` de la classe `PawnModel`. Intéressez-vous pour l'instant uniquement aux déplacements d'1 case en diagonale sans prise (vers le haut pour les pions blancs, vers le bas pour les noirs). Testez avec un `println()` pour vérifier la valeur de retour des fonctions.
 - c. Programmez la méthode `movePiece()` de la classe `CheckersGameModel` qui s'appuie sur la méthode `move()` de la classe `PawnModel`. Pensez au changement de joueur. Testez. Normalement vous devriez pouvoir déplacer des pions noirs et blancs en diagonale, dans le bon sens, sans prise et sans retour en arrière.
 - d. Complétez vos méthodes dans les classes `CheckersGameModel` et `PawnModel` pour gérer les prises. Propagez les évolutions jusqu'à la vue. Testez.

6. Le 4^{ème} exercice étant fini.

- **Finalement, ça voulait dire quoi « Encapsulation » ? Justifiez en vous appuyant sur l'architecture complète du programme.**

Trucs et astuces

- Supprimer un `Component` d'un `Container` : `monContainer.remove(monComponent)`
- Écrire dans une `List` : `maList.add(monObjet)`
- Supprimer d'une `List` : `maList.remove(monObjet)`
- Vérifier que 2 objets sont égaux : `monObjet1.equals(monObjet2)` avec `monObjet1 != null`
- Un `char` peut être « vu » comme un `int` (par son code ASCII). Pour le manipuler comme un `int` :
`monChar - 'a'`

Exercice N°5

A la fin de l'exercice 4, notre programme respecte bien l'architecture MVC et nous avons pu constater que l'encapsulation nous a permis d'effectuer des mises à jour très localisée à chaque évolution.

Notre programme est suffisamment souple pour qu'il soit facile/rapide de programmer le comportement des « Dames » au niveau métier et de propager un look de pièce différente sur la vue.

Il est également possible, sans changer le modèle, de programmer une vue complètement différente (on le voit d'ailleurs avec notre affichage en mode console qui aurait pu être déporté du model dans une vue à part). À noter, qui dit nouvelle vue, dit nouveau contrôleur, ce dernier effectuant les traitements de la vue, il lui est intimement lié.

Vous allez pouvoir maintenant améliorer les algorithmes métiers (rafles, test du meilleur coup, etc.) et/ou gérer les joueurs et leurs gains à votre guise.

Identifiez bien avant chaque modification les classes/méthodes concernées en essayant de ne pas détruire l'architecture mise en place...

Annexes

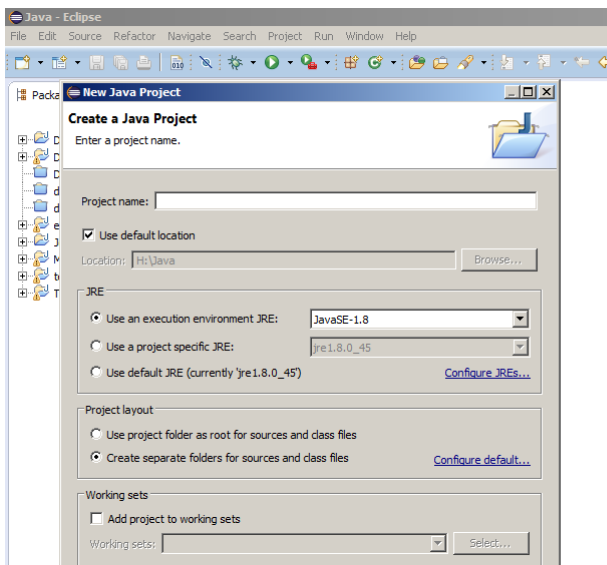
Prise en main d'Eclipse :

Extrait de : <http://www.eclipse totale.com/articles/premierPas.html>

et de : <http://www.jmdoudoux.fr/java/dejae/index.htm>

Une première application Java :

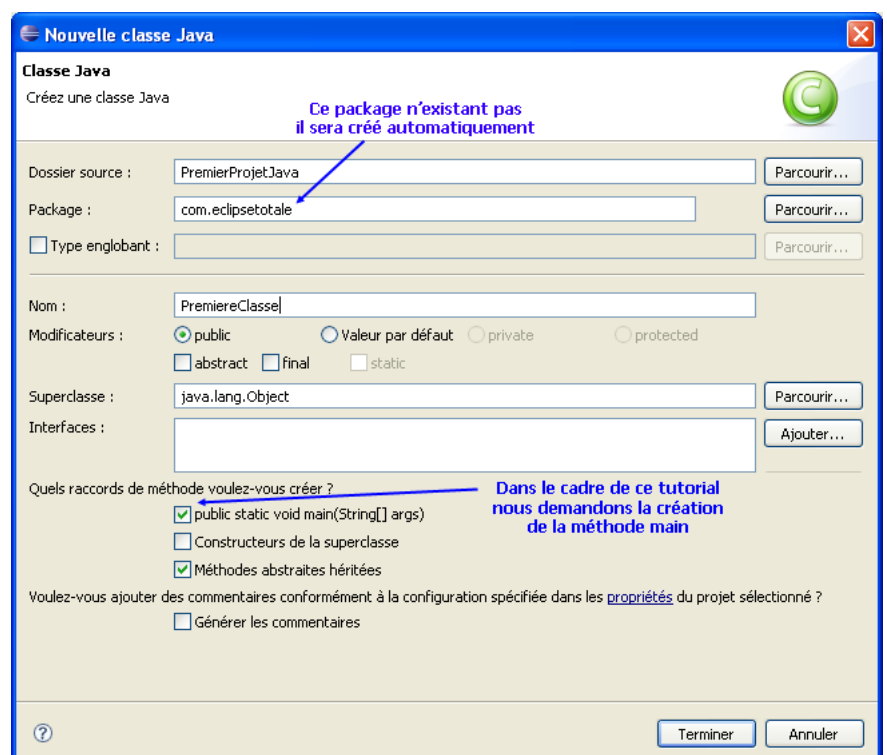
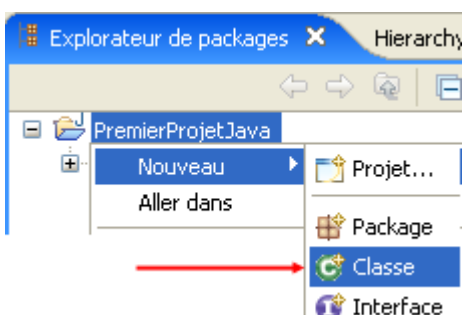
- lancer Eclipse et sélectionner le répertoire qui contiendra le répertoire de travail. Pour réduire les temps de réponse le workspace d'Eclipse sera placé en local dans le répertoire /var/tmp.
- créer un 'Projet Java' (Onglet "New", choisir "Java Project").



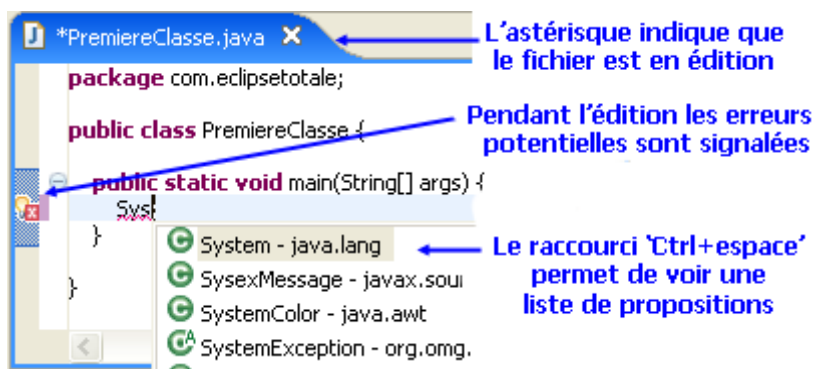
La seule information nécessaire pour l'instant est le nom du projet.


Après la création, le projet apparaît dans la vue 'Explorateur de projets' :

- créer une Classe :
Les informations importantes sont le nom de la classe et le nom de son package.



- Dans l'éditeur, compléter la classe (consultez <http://thierry-leriche-dessirier.developpez.com/tutoriels/eclipse/raccourcis/>) :



- Le raccourci Ctrl+Shift+F déclenche le formatage soit de tout le fichier soit des lignes sélectionnées. (Les options de formatage sont configurables dans Préférences->Java->Style de Code->Formater.)
- Une fois le code saisi, demander l'enregistrement (Ctrl+S ou menu Fichier), la classe est enregistrée et compilée.
- Pour demander l'exécution de la classe : déplier le menu associé au bouton  et sélectionner l'option 'Exécuter en tant que...' -> 'Application Java'.

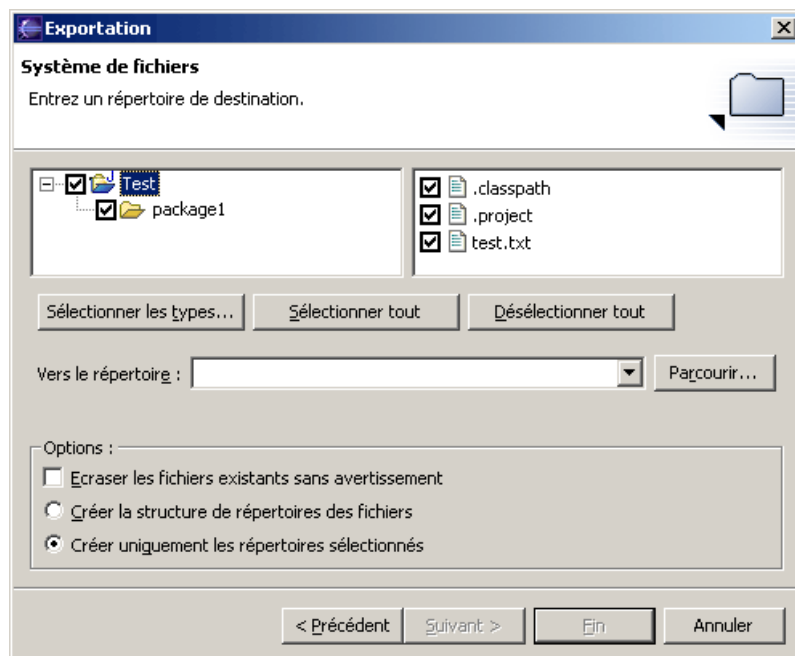
Sauvegarder vos fichiers :

Pour exporter un projet vers votre espace personnel, utiliser le menu "Fichier/Exporter".

L'assistant demande de sélectionner le format d'exportation. Choisir le format "Système de fichiers".

Sélectionner TOUT le projet et indiquer le répertoire de destination.

Attention : pour que la structure des répertoires soit conservée dans la cible, il faut obligatoirement que les répertoires soient sélectionnés.



Importer un projet (ou la copie d'un projet) :

Pour restaurer un projet dans le workspace d'Eclipse, utiliser le menu "Fichier/Importer/General/Existing Projects into workspace". L'assistant vous demande d'indiquer le répertoire source où se trouve le projet à importer (bouton "Browse" (Parcourir)). Pour travailler sur une copie de votre projet, cocher la case "copy projects into ws" (conseillé!).

Tests unitaires

- Pour paramétrer le chemin d'accès vers les bibliothèques de test JUnit, utiliser le menu Project/Properties/Java_Build_Path/Libraries et cliquez sur le bouton "Add Library". Sélectionnez JUnit.
- Indiquez le chemin de la classe de test dans l'instruction d'appel des tests unitaires :

```
public static void main(String[] args){
    org.junit.runner.JUnitCore.main("tp1.HommePolitiqueTest");
}
```

Ajout de nouvelles librairies dans le « Build Path » d'un projet

1. Récupérez les librairies nécessaires (Exemple avec les bibliothèques graphiques).
 - Récupérez sur le e-campus les archives « jcommon-1.0.23.zip » et « jfreechart-1.0.19.zip » et les extraire dans le répertoire de votre projet (sur disque).
 - Ou bien, téléchargez la dernière version des librairies sur le site <http://sourceforge.net/projects/jfreechart/files/>.
2. Intégrez les librairies dans Eclipse.
 - Lancez Eclipse.
 - Sélectionnez Window puis Preferences. Dans le menu organisé sous forme d'arbre à droite de la fenêtre, sélectionnez Java > Build Path > User Libraries
 - Appuyez ensuite sur le bouton New... et entrez JCommon 1.0.23 comme nom de nouvelle librairie puis validez.
 - La librairie devrait désormais apparaître dans la liste à l'écran. Appuyez ensuite sur le bouton Add JARs... et sélectionnez l'archive jcommon-1.0.23.jar dans le répertoire JCommon téléchargé précédemment.
 - Double-cliquez sur la ligne Source attachment : (None) puis sur External Folder... et sélectionnez le sous-répertoire « source » dans le répertoire JCommon.
 - Procédez de même pour la librairie Jfreechart : Appuyer donc sur le bouton New... et entrez JFreeChart 1.0.19 comme nom de nouvelle librairie puis validez.
 - La librairie devrait désormais apparaître dans la liste à l'écran. Appuyez ensuite sur le bouton Add JARs... et sélectionnez l'archive jfreechart-1.0.19.jar dans le sous-répertoire « lib » du répertoire JFreeChart téléchargé précédemment.
 - Double-cliquez sur la ligne Source attachment : (None) puis sur External Folder... et sélectionnez le sous-répertoire « source » dans le répertoire JFreeChart.
 - Validez cette configuration en appuyant sur ok.
3. Ajoutez les librairies au Build Path de votre projet.
 - Positionnez-vous sur votre projet (dans le navigateur d'Eclipse) et faites un clic droit pour obtenir le menu contextuel.
 - Sélectionnez le menu Build Path puis Add Library...
 - Choisissez ensuite User Library puis cliquez sur Next...
 - Cochez les 2 librairies JCommon 1.0.23 et JFreeChart 1.0.19.
 - Cliquez sur Finish pour valider l'ajout des librairies.
 - Elles devraient apparaître dans votre projet (dans le navigateur d'Eclipse) sous « JRE System Library ».