Cahier de TD
CPOO2
4INFO, 2024-2025

Exercice 1. Optional

```java
Optional<Foo> opt = Optional.of(foo);
if(opt.isPresent()) {
   //...
}
String str = opt.map(foo -> foo.toString()).orElse("");
```

Q1. Donner le code Java de la classe `Optional` en fonction du code ci-dessus (avec donc les méthodes `of`, `isPresent`, `map`, `orElse`).

Exercice 2. Injection de dépendances

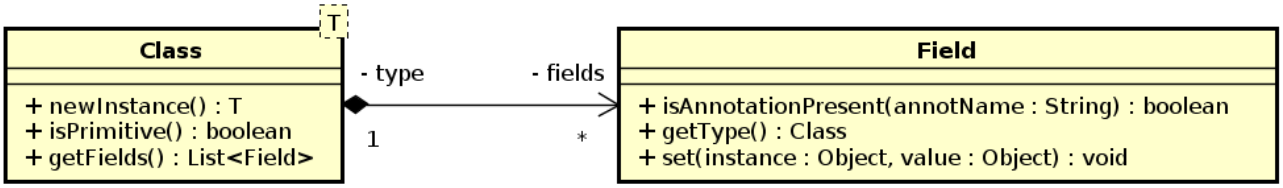
L'injection de dépendances est un patron de conception très utilisé. Il permet d'injecter automatiquement une valeur dans un attribut ou en paramètre d'une méthode. Par exemple, le code Java suivant utilise l'annotation *Inject* pour spécifier les attributs dont la valeur doit être injectée à la Création de l'objet :

```java
class A {
  @Inject
  B b;
}

class B {}

class C {
  @Inject
  A a;
}
```

Le fonctionnement de l'injection de dépendances (simplifiée pour cet exercice) est le suivant : on demande à une fabrique spécifique (l'injecteur) une instance d'un certain type. La fabrique, instancie la classe donnée (par exemple *A*), puis identifie les attributs annotés avec *Inject* (l'attribut *b* dans *A*). La fabrique injecte alors dans cet attribut une valeur (pour *b* dans *A* on insère une nouvelle instance de *B*) (la méthode *set* de la classe *Field*).

Proposer une fabrique *Injector* contenant au moins une méthode *createInstance* permettant d'instancier un objet et de réaliser de l'injection de dépendances dans les attributs de cet objet annotés par *Inject*. Votre solution doit être générique (i.e. pas liée à *A*, *B*, *C* qui sont données ici à titre d'illustration). Pour cela, vous devrez utiliser l'introspection dont voici les classes et méthodes qui devraient vous intéresser :

Pour rappel, l'accès l'instance *Class* d'une classe (par exemple *A*) s'effectue ainsi : *A.class*. L'opération *newInstance* créer une nouvelle instance de la classe (votre solution ne devrait pas utiliser les constructeurs des classes). *isPrimitive* retourne *true* si la classe est celle d'un type primitif. *isAnnotationPresent* retourne *true* si l'annotation donnée en paramètre est présente sur l'attribut. *getType* retourne le type de l'attribut. *set* place la valeur *value* dans l'attribut courant de l'objet *instance*. **Votre code ne doit rien faire lors de la Création d'un objet primitif (exemple : *String.class*). Veillez à gérer les dépendances cycliques** : lorsqu'une dépendance est détectée (comme dans le code suivant) lors de l'injection, l'attribut n'est pas injecté.




```
class E {
  @Inject
  F f;
}

class F {
  @Inject
  E e;
}
```

Je vous conseille de faire une première solution sur brouillon sans la gestion des dépendances cycliques et de l'améliorer ensuite si vous avez le temps.

Le squelette de l'injecteur à créer est le suivant :

```
public class Injector {
  public <T> Optional<T> createInstance(final Class<T> cl) {
  }
}
```

Exercice 3. *Observable* et *Promise*

```
export interface Bar {
  attr1: number;
  attr2?: string;
}

export class Foo {
  private http: HttpClient;
  private errMsg: string | undefined;
  private data: Array<Bar>;

  public foo(): void {
    this.http... //TODO Q1

    console.log(this.data); // Q6
  }

  public foo2(p: string): void {
    // TODO Q5
  }
}
```

Q1. Soit une route REST *'api/v1/foo/bar'* GET qui retourne en JSON une liste d'objets de type *Bar*. Donner le code TypeScript/Angular exécutant une telle requête à l'aide de l'attribut *http*. Vous utiliserez l'API standard Angular pour faire cela à savoir *Observable*. Le traitement de la réponse affectera le résultat obtenu à l'attribut *data*. Traitez le cas d'une erreur dans la requête en modifiant l'attribut *errMsg*.

Q2. Changer ensuite le code pour utiliser *Promise* à l'aide de *lastValueFrom()*.

Q3. Pourquoi ne faut-il pas écrire : this.http.get('http ://www.mon.service.web/api/foo')...;?

Q4. Quelle est la différence entre *Observable* et *Promise* ? Et dans le cas des requêtes REST, dans quel rare cas *Observable* est pertinent ?

Q5. Soit une route REST *'api/v1/foo/yolo/param'* GET qui ne retourne rien. Donnez le code TypeScript/Angular exécutant une telle requête en utilisant le paramètre *p* pour le paramètre *param*.

Q6. Qu'affiche la ligne *console.log(this.data)* ?

Exercice 4. *Fabrique*

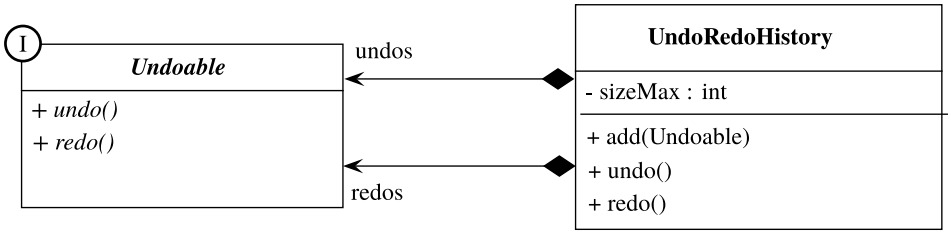
Voici une énumération définissant les quatre couleurs d'un jeu de carte.

```
1 public enum ColourCard {
2   SPADE, CLUB, HEART, DIAMOND;
3 }
```

Q.1 Développer en Java un moyen pour obtenir une couleur à partir d'une chaîne de caractères correspondant au nom de la couleur recherchée (le nom d'un élément d'une énumération s'obtient grâce à la méthode `name()`, par ex. `SPADE.name();`, tandis que la méthode `values()` retourne un tableau contenant tous les éléments de l'énumération, par exemple `ColourCard[] items = values();`).

Exercice 5. Undo / Redo

Lorsqu'un utilisateur réalise une action, il est souvent recommandé que celle-ci soit annulable. Par exemple avec l'éditeur de dessin, ajouter une forme devrait être annulable. Cela demande : de considérer une action comme un objet pouvant être stocké et manipulé ; de sauvegarder l'état des objets modifiés par l'action pour pouvoir revenir à l'état précédent.



Donner le code Java de la classe *UndoRedoHistory* qui collecte les objets annulables.

Exercice 6. *Fabrique* et *Commande*

En entreprise, le code que vous allez produire sera normalement analysé pour y détecter des erreurs et des défauts (anti-patrons, design smells, *etc.*). Une des métriques les plus utilisées – et très certainement la plus pénible – est la complexité cyclomatique. Dans le cas de certaines fabriques cela peut devenir problématique : certaines fabriques peuvent avoir une complexité cyclomatique élevée due aux tests analysant le paramètre donné à la méthode. Par exemple la méthode de fabrique suivante crée une instance d'un sous-type de l'interface *ExpArithm* en fonction d'une chaîne de caractères.

```
1  public class FabriqueExp {
2    public Optional<ExpArithm> creerExpression(final String txtExpression) {
3      ExpArithm exp = null;
4      if("mult".equals(txtExpression)) {
5        exp = new Mult();
6      }
7      if("plus".equals(txtExpression)) {
8        exp = new Plus();
9      }
10     if("mod".equals(txtExpression)) {
11       exp = new Mod();
12     }
13     if("min".equals(txtExpression)) {
```

```
14        exp = new Min();
15      }
16      return Optional.ofNullable(exp);
17    }
18  }
```

Q.1 Utilisez le patron de conception *Commande* pour coder en Java une nouvelle version de la fabrique diminuant drastiquement la complexité cyclomatique de la fonction `créerExpression`.

Exercice 7. *Stream*

Récrivez le code suivant en utilisant les *Streams* Java.

```
1 Optional<Bar> getBar(final List<Bar> src) {
2   for(final Bar bar : src) {
3     if(bar.isBabar()) {
4       return Optional.of(bar);
5     }
6   }
7   return Optional.empty();
8 }
```

```
1 public List<String> foo(final List<List<String>> ls) {
2   final List<String> res = new ArrayList<>();
3   for(final List<String> l : ls) {
4     for(final String s : l) {
5       res.add(s);
6     }
7   }
8   return res;
9 }
```

```
1   public List<String> yolo(final List<Integer> data) {
2     final int size = data.size();
3     final List<String> res = new ArrayList<>();
4     int i = 0;
5     while(i < size) {
6       if(data.get(i) < 10) {
7         res.add(data.get(i).toString());
8       }
9       i++;
10    }
11    return res;
12  }
```

Exercice 8. Log

Q1. Pour rappel, si le niveau DEBUG est utilisé alors seuls le niveau de sévérité DEBUG et ceux plus élevés (SEVERE, etc.) provoquerons le logging des informations. Pour les autres niveaux (exemple : INFO), l'information à loguer sera ignorée. Donc, lorsque le niveau de log du programme est Level.DEBUG (et que donc les messages de niveau INFO seront ignorés), quelle est la différence de comportement entre :

```
LOG.log(Level.INFO, () -> "process CtClass: " + clazz);
```

et

```
LOG.log(Level.INFO, "process CtClass: " + clazz);
```

Quelle est le patron de conception sous-jacent ? Quel est l'avantage de la première méthode par rapport à la seconde ?

Exercice 9. *Stratégie*

Un jeu vidéo du type PACMAN peut avoir une partie en cours. Une partie possède un unique niveau de difficulté mais qui évoluant dans le temps. Au début de la partie, le méchant de PACMAN se déplace de manière aléatoire (*aléatoire*). Après 2 minutes, ce méchant se déplace vers PACMAN de manière verticale ou horizontale (*ligne*). Enfin, après à nouveau 2 minutes, il se déplacer vers PACMAN également en diagonale (*diagonale*).

Q.1 Donner un diagramme de classes décrivant le jeu, la partie et les niveaux de difficultés.

Exercice 10. *Fabrique Abstraite*

Soient les concepts suivants de *MutableList*, *MutableHashMap*, *MutableSet*, *ImmutableList*, *ImmutableMap* et *ImmutableSet*. Nous nous ne préoccuperons pas des méthodes de ces concepts dans cette exercice.

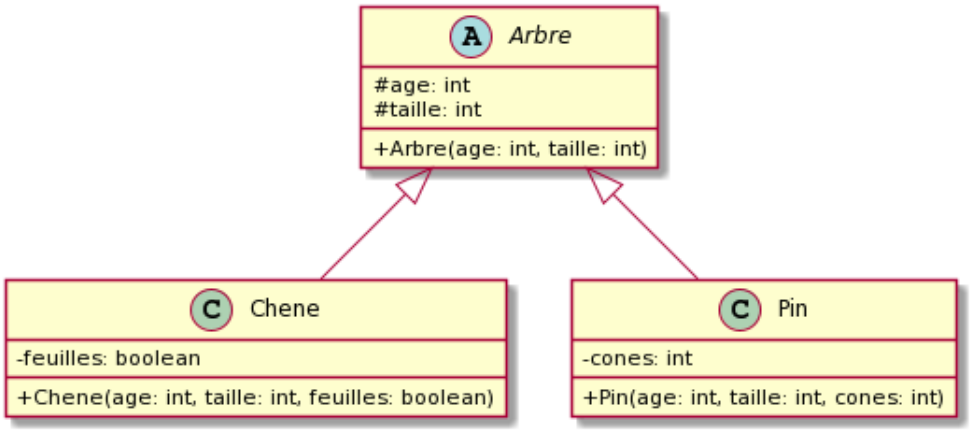
Dans un premier temps mettez en place la patron de concept *Pont* (API) pour définir une librairie permettant de manipuler ces concepts. Votre solution aura une unique implémentation de l'API. Pour l'API prenez les noms des concepts ci-dessus et pensez à l'héritage.

Dans un second temps, complétez votre solution pour y ajouter le patron de conception *fabrique abstraite*.

Enfin, donnez du code Java illustrant comment s'utilise votre solution (créez quelles instances).

Exercice 11. *Builder*

Voici le diagramme de classes décrivant deux types d'arbres : le *Chêne* et le *Pin*. Ces deux types d'arbres ont un age (années) et une taille (mètres). Un chêne peut avoir des feuilles ou non (attribut *feuilles*). Un pin peut avoir un certain nombres de cônes (le fruit du pin, attribut *cones*).



Nous voulons que le code client configure et crée des pins et des chênes à l'aide de **monteurs fonctionnels**. Avec ces **monteurs fonctionnels**, l'utilisateur n'est pas obligé de spécifier tous les attributs de l'arbre pour pouvoir le créer. Ainsi, les valeurs par défaut à utiliser sont : *age = 1*, *taille = 1*, *feuilles = false*, *cônes = 10*.

1. Donnez le diagramme de classes de vos **monteurs fonctionnels**.
2. Donnez le code Java du monteur de chênes (uniquement ce monteur).
3. Donnez un exemple d'utilisation de votre monteur de chênes pour créer un chêne de 10 ans, sans feuille.

Exercice 12. Fluent API

Lorsque nous écrivons des tests front-end, nous utilisons généralement ce que nous appelons un 'robot' pour interagir avec l'interface utilisateur. Deux exemples en Java :

```
public void test(RobotFactory factory) {
  factory.newRobot()
    .click(10, 2)
    .type("yolo");

  factory.newRobot()
    .press(10, 2)
    .release()
    .execute(() -> {
      System.out.println("coucou");
    })
    .press(10, 2)
    .release();
}
```

Le style de ce code est ce que l'on appelle une API fluide (*fluent API*), qui ressemble fortement au patron *monteur*. Cependant, notre API fluide ne construit rien et n'a pas de méthode terminale *build*.

La méthode *type* prend un *string* en paramètre correspondant au texte à écrire au clavier. La méthode *execute* prend un argument (à vous de trouver son type) correspondant au code à exécuter au moment voulu. Les méthodes *click* et *press* ont deux paramètres x et y (*double*) correspondant à la position de la souris.

Plus compliqué, nous voulons que certaines méthodes soient accessibles que dans certains cas : la méthode *release* doit être accessible uniquement après un appel à la méthode *press*. Inversement, les méthodes *press* et *click* ne sont plus accessibles après un appel à la méthode *press* tant qu'un appel à *release* n'est pas fait.

Deux exemples de code qui ne doivent pas compiler :

```
factory.newRobot()
  .press(10, 20)
  .press(30, 10); // ne compile pas

factory.newRobot()
  .press(10, 20)
  .click(30, 10); // ne compile pas

factory.newRobot()
  .release(); // ne compile pas
```

Q1. Sur la page suivante, donnez le diagramme de classes permettant de résoudre notre problème (qui contiendra donc au moins la classe *RobotFactory*). Ne vous préoccupez pas des attributs des classes.

Exercice 13. Lazy object

Donner le code Java correspondant (pour cela, vous ne tiendrez pas compte du mot-clé *val* qui équivaut au mot-clé *final* de Java). Le mot-clé *lazy* fait référence à l'instanciation paresseuse. La méthode *getBar* est un accesseur en lecteur sur l' objet *bar*.

```
class Foo {
  private lazy val bar : Object = new Object()
  def getBar() : Object {
    return bar
  }
}
```

Exercice 14. Typage

Étant donné le code suivant écrit en langage Go définissant une interface *Canard* et une classe *Loup* :

```go
type Canard interface { // public interface Canard
  coincoin() string     // String coincoin();
  danser() string       // String danser();
}

type Loup struct { }    // public class Loup

// definition et implementation des methodes de la classe Loup.
// En Go, les fonctions sont definies et implementees a l'exterieur de la definition de la classe.
// (loup Loup) est une reference a l'objet courant (i.e, le "this").
func (loup Loup) coincoin() string { return "COIN COIN OUUUH" }
func (loup Loup) danser() string  { return "\_()_/" }
func (loup Loup) manger(c Canard) string  { return "\_(@@)_/" }
```

Un objet *Loup* peut alors se faire passer pour un *Canard*, comme le montre le code Go suivant :

```go
func main() { // le "public static void main"
  var loup Loup // Creation un Loup ("Loup loup = new Loup();")
  var canard Canard // null car Canard n'est pas instanciable ("Canard canard = null;")

  // Le Loup prend la forme d'un Canard
  canard = &loup  // En java, "canard = loup;"

  // Affiche "COIN COIN OUUUH" en console :
  fmt.Printf("%s\n", canard.coincoin()) // System.out.println(canard.coincoin());
}
```

1. Quel est le système de typage utilisé par Go expliquant pourquoi le code ci-dessus compile ?

2. Pourquoi n'est-il pas possible de faire cela en Java ?

3. En Java, quel patron de conception faut-il utiliser pour rendre compatible *Loup* avec *Canard* ? Un indice : le loup devra porter un cosplay (déguisement) de canard. Pour rappel, nous sommes dans le cas où ne voulons ou pouvons pas modifier la classe *Loup* pour la rendre explicitement compatible.

4. Donnez le code Java équivalent (ou presque) aux deux morceaux de code Go donnés ci-dessus en utilisant le patron précédemment trouvé.

Exercice 15. Questions

1. Donner le nom de deux patrons de conception utilisés dans le code suivant :

```java
angle.textProperty().addListener((o, oldValue, newValue) -> {
  if(newValue.isEmpty()) {
    System.out.println("Not a good value.");
    angle.setText(oldValue);
  }
});
```

2. Est-ce que la ligne `map.getOrDefault(str, () -> null).myOperation();` peut produire un *NullPointerException* ?

3. Un *layout* est un objet permettant de disposer les composants graphiques d'un container selon un algorithme particulier. Il est possible de changer le layout d'un container, comme le montre le code suivant (où *StackPane* est un container et *HBox* et *VBox* des layouts). Quel est le patron de conception sous-jacent ?

```java
StackPane pane = new StackPane();
pane.setLayout(new HBox());
pane.setLayout(new VBox());
```

4. L'interface *javax.swing.Action* permet d'attacher à des composants graphiques Swing une action à exécuter lors de leur utilisation. D'aprés la Javadoc, cette interface possède notamment l'opération suivante :
`void actionPerformed(ActionEvent e)` *: appelée lorsqu'une action est déclenchée.*
À quel patron de conception cette interface *Action* fait référence ?

5. La classe *Container* (qui hérite de *Component*) peut contenir des instances de *Component*. Elle possède l'opération suivante :
public void add(Component component) : ajoute le *Component* donné en paramètre à la fin du *Container*.
De quel patron de conception s'agit-il ?

Exercice 16. *Visiteur / Visitor*

Dans cet exercice nous allons modéliser des expressions arithmétiques sous la forme d'un arbre binaire. Seules l'addition et la soustraction devront être considérées. Un *Arbre* possède un *Noeud racine* et un *nom*. Un *Noeud* est soit un *NoeudPlus*, soit un *NoeudMoins*, soit un *NoeudValeur*. *NoeudPlus* et *NoeudMoins* possède chacun un *noeud droit* et un *noeud gauche*. *NoeudValeur* possède une *valeur* (un entier). Vous modéliserez *Noeud* sous la forme d'une interface.
Q.1 créez le diagramme de classes de l'arbre.
Q.2 Ajoutez le patron de conception *Visiteur* à l'arbre : ajoutez les méthodes *accept(VisiteurArbre)* à ce diagramme de classes et définissez l'interface *VisiteurArbre* et ses méthodes comme vu en cours.
Q.3 Donnez le code Java de chacune des méthodes *accept* ajoutée.
Q.4 Pourquoi les méthodes *accept* sont-elles nécessaires ?
Q.5 Implémentez en Java un visiteur permettant d'afficher dans la console et en notation post-fixée la formule arithmétique que représente l'arbre.
Q.6 Implémentez en Java un visiteur permettant de calculer la formule arithmétique que représente l'arbre. Un indice : le visiteur possédera une pile stockant les valeurs visitées. Vous utilisez alors le patron *Visiteur* pour faire un *Interpréteur* de manière propre.
Q.7 Implémentez en Java un visiteur permettant d'afficher dans la console sous la forme de balises XML la formule arithmétique que représente l'arbre.

```
<Formula name="foo">
  <Plus>
    <Value value="2" />
    <Value value="3" />
  </Plus>
</Formula>
```