Développement Orienté Objets
Année 2024-2025

IUT Montpellier-Sète
Département Informatique
M. Bougeret, M. Gasquet, S. Nabitz, C. Nadal, V. Poupet, P. Valicov

Examen : durée 3 heures

CONSIGNES :

Document autorisé : une feuille A4 manuscrite recto/verso. Matériel électronique interdit.
Le barème (excédentaire) est donné à titre indicatif.
Votre écriture doit être claire et lisible.
Chaque partie est à rendre sur une copie séparée.

Partie 1 – Tour de notions

Durée estimée : 45 minutes

1. (*2 points*) Soit le code suivant :

```java
public class GestionVilles {
    public static void main(String[] args) {
        Ville c1 = new Ville("Montpellier");
        Ville c2 = new Ville("Sète");
        Ville c3 = new Ville("Nîmes");
        Ville c4 = new Ville("Béziers");

        c1 = c2;
        c2 = c3;
        c3 = null;

        System.out.println(c1 + " " + c2 + " " + c3 + " " + c4);

        changerVille(c1, c3);

        System.out.println(c1 + " " + c2 + " " + c3 + " " + c4);

        c3 = c4;

        System.out.println(c1 + " " + c2 + " " + c3 + " " + c4);

        c2 = c1;

        System.out.println(c1 + " " + c2 + " " + c3 + " " + c4);

        c2.setNom("Paris");

        System.out.println(c1 + " " + c2 + " " + c3 + " " + c4);
    }

    public static void changerVille(Ville v1, Ville v2) {
        Ville temp = v1;
        v1 = v2;
        v2 = temp;
    }
}
```

```java
class Ville {
    private String nom;

    public Ville(String nom) {
        this.nom = nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    @Override
    public String toString() {
        return nom;
    }
}
```

Y a-t-il des erreurs de compilation ou exécution dans ce programme ? Si oui, expliquez brièvement lesquelles. Si non, donnez le résultat d'affichage produit à l'exécution de la méthode principale.

2. (*2 points*) Soient les 2 classes suivantes. En invoquant le contrat des fonctions `equals(Object o)` et `hashCode()`, expliquez l'affichage produit par le code ci-dessous.

```java
import java.util.Objects;

public class Personne {
    private String nom;
    private int age;

    public Personne(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null ||
                getClass() != o.getClass())
            return false;
        Personne personne = (Personne) o;
        return age == personne.age;
    }

    @Override
    public int hashCode() {
        return Objects.hash(nom, age);
    }
}
```

```java
import java.util.Arrays;
import java.util.HashSet;

public class AppPersonnes {
    public static void main(String[] args) {
        Personne p1 = new Personne("Lolo", 25);
        Personne p2 = new Personne("Fifi", 19);
        Personne p3 = new Personne("Lolo", 25);
        Personne p4 = new Personne("Riri", 25);

        HashSet<Personne> personnes = new HashSet<>();
        personnes.add(p1);
        personnes.add(p2);
        personnes.add(p3);
        personnes.add(p4);

        System.out.println(personnes.size());

    }
}
```

3. (*1 point*) Voici une suite de commandes exécutées dans un terminal Bash :

```bash
# création d'un répertoire et déplacement vers celui-ci :
mkdir mon_projet ; cd mon_projet

# initialisation du répertoire comme un dépôt Git
git init

# création d'un fichier :
echo "hello" > fichier.txt

git commit -m "sauvegarde du fichier"

git push
```

Expliquez que s'est-il passé après chacune des deux dernières commandes.

Partie 2 – Fabrication de ressources

Durée estimée : 45 minutes

On considère une application d'un jeu, où le joueur peut construire des bâtiments/zones qui produisent des ressources. Pour chaque bâtiment/zone fabricant de ressources, on souhaite pouvoir connaître son nom et calculer son rendement.

Pour l'instant, il existe deux types de fabriques de ressources :

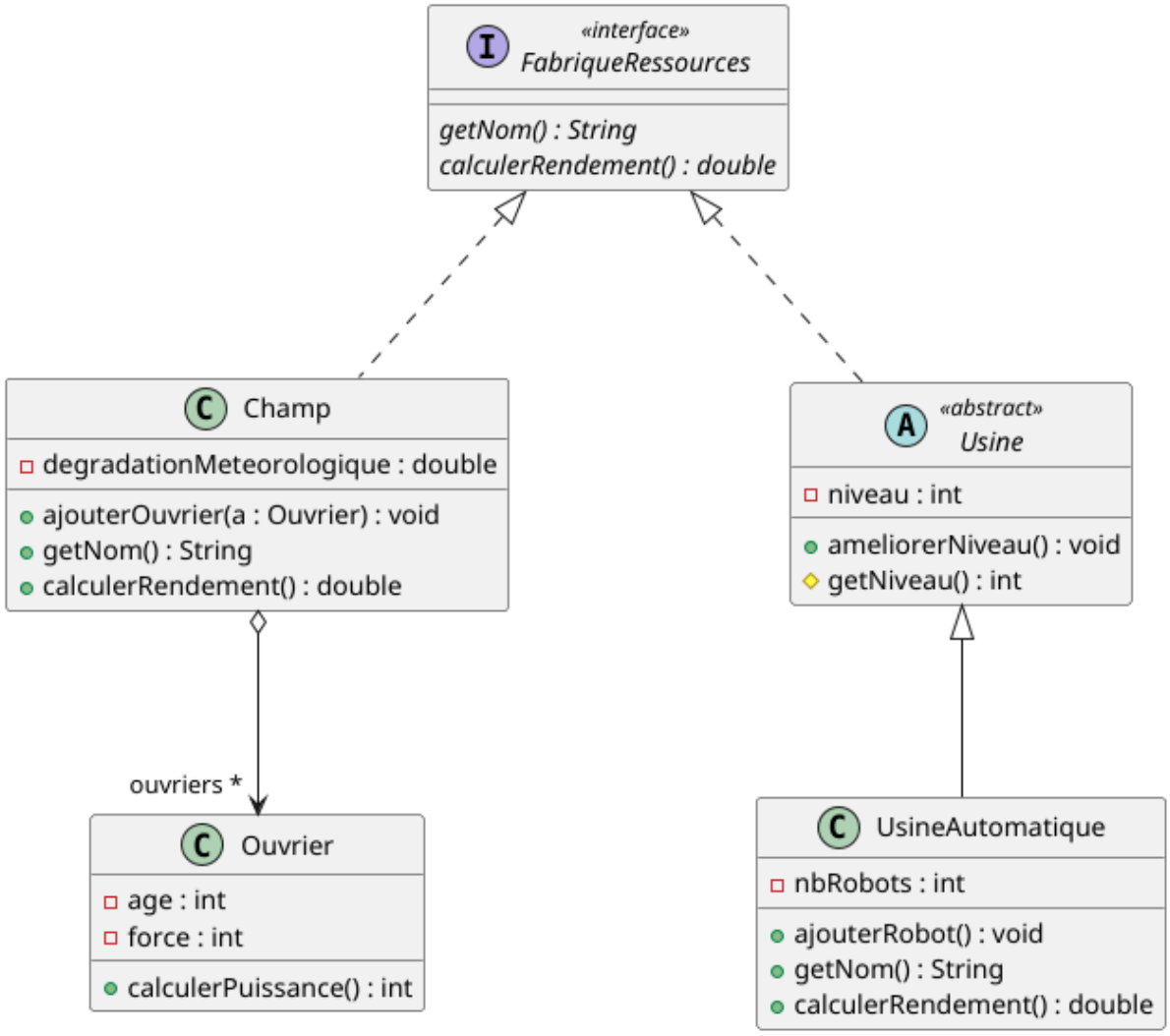
— **Usines**, qui possèdent un niveau et le rendement dépend du niveau de l'usine.

— **Champs** qui incluent des **Ouvriers** qui ont une certaine puissance qui dépend de leur âge et de leur force. Le rendement d'un champ dépend de la puissance de l'ensemble de ses ouvriers et de la dégradation météorologique aléatoire.

Dans le futur, on prévoit d'ajouter de nouvelles usines dont le point commun est d'avoir un niveau, on a donc inclut une classe abstraite **Usine** dont dérive **UsineAutomatique**. L'usine automatique est une usine qui fait intervenir des robots et son rendement dépend à la fois de son niveau en tant qu'**Usine**, mais aussi du nombre de robots.

On a également prévu d'ajouter d'autres types de champs, mais pour l'instant on ne s'intéresse qu'aux champs avec une dégradation météorologique aléatoire.

Voici le diagramme de classes du code existant :



Voici également le code partiel de quelques classes :

```java
import java.util.ArrayList;
import java.util.List;

public class Champ implements FabriqueRessources {
    private final List<Ouvrier> ouvriers;
    private final double degradationMeteorologique;

    public Champ() {
        ouvriers = new ArrayList<>();
        degradationMeteorologique = Math.random();
    }

    public void ajouterOuvrier(Ouvrier o) {
        ouvriers.add(o);
    }

    @Override
    public String getNom() {
        return "Champ";
    }

    @Override
    public double calculerRendement() {
        double rend = 0;
        for (Ouvrier o : ouvriers)
            rend += o.calculerPuissance();
        return rend * (1 - degradationMeteorologique);
    }
}
```

```java
public abstract class Usine implements
    FabriqueRessources {
    private int niveau;

    public Usine(int niveau) {
        this.niveau = niveau;
    }

    public void ameliorerNiveau() {
        niveau++;
    }

    protected int getNiveau() {
        return niveau;
    }
}
```

```java
public class UsineAutomatique extends Usine {
    // du code de la classe ici

    @Override
    public String getNom() {
        return "Usine Automatique";
    }

    @Override
    public int calculerRendement() {
        return getNiveau() * nbRobots;
    }

    // le reste du code de la classe ici
}
```

On souhaite maintenant ajouter une nouvelle usine, l'**UsineManuelle**. C'est une usine qui possède un niveau (comme toutes les usines) et inclut des ouvriers. A l'initialisation, il n'y a aucun ouvrier dans l'usine. Il doit donc être possible d'en ajouter via une fonction. Le rendement d'une usine manuelle est calculée ainsi : somme de la puissance de chaque ouvrier * niveau de l'usine * 0.5. On a donc un algorithme en partie identique à celui de **Champ**.

1. (*3 points*) Écrivez la nouvelle classe **UsineManuelle** en faisant un refactoring du code existant afin d'éviter la duplication de code. Vous pouvez ajouter d'autres classes nouvelles si besoin. **Attention**, refactorer signifie que le comportement des classes données doit rester le même qu'auparavant, mais la *structure* et le code peuvent changer.

   Indication : Le code qui vous est fourni devrait suffire pour déduire ce qu'il faut écrire pour la nouvelle classe **UsineManuelle**.

2. (*2 points*) Dessinez le nouveau diagramme de classes après refactorisation.

Partie 3 – Enveloppe de messages

Durée estimée : 40 minutes

On souhaite créer une application de messagerie avec différents types de contenu. Pour simplifier, on suppose que les types de contenus (**Texte**, **Image**, **Pdf** etc.) n'ont aucun rapport entre eux, donc pas d'héritage entre ces classes, leur seule classe-mère commune étant **Object**.

Voici le code d'une classe utilitaire qui permet de gérer les messages. Son code vous est donné pour que vous puissiez comprendre l'utilisation du code que vous aurez à écrire.

```
import java.util.ArrayList;
import java.util.List;

public class UtilitaireMessagerie {
    public static <T> boolean contient(List<Enveloppe<T>> enveloppes, T message) {
        for (Enveloppe<T> enveloppe : enveloppes)
            if (message.equals(enveloppe.getMessage()))
                return true;
        return false;
    }

    public static <T> void echangerContenu(Enveloppe<T> e1, Enveloppe<T> e2) {
        T temp = e1.getMessage();
        e1.setMessage(e2.getMessage());
        e2.setMessage(temp);
    }

    public static <T> List<Enveloppe<T>> getCopieListeEnveloppes(List<Enveloppe<T>> source) {
        List<Enveloppe<T>> resultat = new ArrayList<>();
        for (Enveloppe<T> enveloppe : source)
            resultat.add(new Enveloppe<>(enveloppe.getMessage()));
        return resultat;
    }
    // D'autres méthodes utilitaires ici manipulant des enveloppes
}
```

On vous rappelle que le principe des méthodes génériques est le même que des classes génériques, à la différence près qu'on ne paramétrise pas toute la classe, mais uniquement la fonction correspondante. La portée du type générique est donc limitée à la fonction.

1. (*2 points*) À partir du code qui vous est donné ci-dessus, et uniquement à partir de celui-ci, ecrivez la classe **Enveloppe**.

2. (*1 point*) Ajoutez à la classe **Enveloppe** un attribut de type **LocalDate** qui représente la date de création de l'enveloppe. Sans modifier les fonctions déjà écrites, ajoutez un nouveau constructeur approprié qui construit l'enveloppe en y intégrant l'initialisation de la date.

3. (*2 points*) On vous donne maintenant une nouvelle fonction de la classe **UtilitaireMessagerie** qui permet d'ordonner les enveloppes en fonction de leur date de création :

```
public static <T> PriorityQueue<Enveloppe<T>> trierEnveloppesParDate(List<Enveloppe<T>> enveloppes) {
    PriorityQueue<Enveloppe<T>> courrierOrdonne = new PriorityQueue<>();
    courrierOrdonne.addAll(enveloppes);
    return courrierOrdonne;
}
```

Complétez le code de la classe **Enveloppe** pour que la fonction **trierEnveloppesParDate** puisse fonctionner.

Partie 4 – Parking

Durée estimée : 50 minutes

Une application gérant le parking de véhicules à longue durée (un jour minimum) vous est proposée. Le principe de fonctionnement est le suivant : lorsqu'un véhicule arrive dans le parking, le système lui attribue une place. Une place de parking a un coût par jour et une capacité (sa surface totale). Lorsqu'un véhicule est affecté à une place de parking, l'attribut **estLibre** de cette dernière devient **false**. Dans la classe principale on vous propose un scénario où 4 véhicules arrivent à une certaine date. Le problème est que par flemme, par insouciance, ou tout simplement parce qu'il n'a pas été recruté pour, le développeur n'a pas pu ajouter certaines fonctionnalités...

Dans les questions qui vont suivre vous êtes libres (et encouragés même !) d'ajouter des nouvelles méthodes/attributs/classes là où cela vous paraît nécessaire. En revanche, le code qui vous est fourni doit rester inchangé.

```java
public class Place {
    private Vehicule vehiculeGaré;
    private boolean libre = true;
    private double coutJour;
    private double capacité;

    public Place(double cp, double cj) {
        capacité = cp; coutJour = cj;
    }

    public boolean estLibre() {
        return libre;
    }

    public double getCoutJour() {
        return coutJour;
    }

    public double getCapacité() {
        return capacité;
    }

    public void affecterVehicule(Vehicule v){
        // à compléter
    }

    public double liberer() {
        // à compléter
    }

    /* méthode qui retourne le nombre de jours
    *  à partir d'une date passée en paramètres
    *  jusqu'à maintenant
    */
    private int getNbJours(LocalDate date) {
        return (int)
            ChronoUnit.DAYS.between(date,
            LocalDate.now());
    }
}
```

```java
import java.time.LocalDate;

public class Vehicule {
    private LocalDate dateEntree;
    private String nrImmatriculation;

    public Vehicule(String nrImmat) {
        nrImmatriculation = nrImmat;
    }

    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;
        Vehicule vehicule = (Vehicule) o;
        return Objects.equals(nrImmatriculation,
            vehicule.nrImmatriculation);
    }

    public int hashCode() { return 42; }
}
```

```java
import java.util.ArrayList;

public class Parking {
    private ArrayList<Place> places = new ArrayList<>();

    public void ajouter(Place place) {
        places.add(place);
    }

    public double sortirVehicule(Vehicule v) {
        /* à compléter */
    }
}
```

```java
import java.time.LocalDate;
import java.util.Random;

public class AppParking {
    public static void main(String[] args) {
        Parking monParking = new Parking();
        // On crée des places de parking avec capacités et prix aléatoires
        for (int i = 0; i < 100; i++) {
            double capacite = new Random().nextDouble() * 10;
            double cout = new Random().nextDouble() * 25;
            monParking.ajouter(new Place(capacite, cout));
        }
        Vehicule v1, v2, v3, v4;
        v1 = new Vehicule("PP-989-KK");
        v2 = new Vehicule("BZ-201-CA");
        v3 = new Vehicule("AO-167-DP");
        v4 = new Vehicule("PP-989-KK");
    }

}
```

Les places

On se propose d'abord de compléter la classe `Place` :

1. (*0.5 points*) Complétez la méthode `void affecterVehicule(Vehicule v)` qui affecte un véhicule sur la place uniquement si celle-ci est libre en mettant à jour la date d'entrée du véhicule à **LocalDate.now()**. Si la place n'est pas libre alors la méthode ne fait rien.

   Remarque : vous supposerez qu'un véhicule ne peut occuper qu'une seule place à un moment donné. Vous n'avez donc pas à le vérifier.

2. (*1 point*) Complétez la méthode `double liberer()` qui libère la place si celle-ci est occupée et retourne le coût total facturé pour le véhicule en fonction du nombre de jours qu'il a stationné sur la place. Vous utiliserez la méthode `getNbJours(...)` pour cela. Si la place est libre, alors la méthode retournera 0.

Le parking

La classe `Parking` va gérer les entrées et sorties des véhicules en les affectant aux bonnes places.

1. (*0.5 points*) Complétez la méthode `double sortirVehicule(Vehicule v)` qui prend en paramètre un véhicule, libère la place sur laquelle celui-ci est garé et retourne le cout final. Si le véhicule n'est pas garé dans le parking, cette méthode retournera 0.

2. (*3 points*) Ajoutez à la classe `Parking` une méthode `void garerVehicule(...)` qui affecte un véhicule à une place de parking en fonction d'un *critère* défini par l'utilisateur. Au moment de l'appel de cette méthode, on passe en paramètre un véhicule et le critère de l'utilisateur. On peut supposer que le critère a été préalablement choisi dans la fonction `public static void main(String[] args)` de la classe `App`.

   Pour le moment 2 critères sont à prévoir : la première place disponible ou bien la place la moins chère. Mais plus tard, une fois que vous aurez terminé ce travail et en fonction de l'évolution du marché, 59 autres critères devraient pouvoir être ajoutés **sans toucher au code des classes déjà écrites** (notamment `Vehicule`, `Place` et `Parking`). À vous de proposer le code modélisant les critères utilisateur.

   Remarque : Il n'est pas demandé de prévoir un traitement particulier s'il n'y a pas de place dans le parking. En effet, dans ce cas précis, la recherche d'une place correspondant au critère demandé ne réussira pas et donc le véhicule ne sera pas affecté à une place.

   Remarque (rappel) : Il ne vous est pas demandé d'implémenter la saisie du choix de l'utilisateur.