Nom&prénom de l'étudiant : DENEUBOURG Florence

INSTITUT PROVINCIAL D'ENSEIGNEMENT DE PROMOTION SOCIALE DE WALLONIE PICARDE

SIÈGE : RUE PAUL PASTUR, 2 – 7500 TOURNAI
IMPLANTATION : RUE PAUL PASTUR, 49 – LEUZE-EN-HAINAUT

Programmation Orientée Objet

Projet

Nom&prénom de l'étudiant : DENEUBOURG Florence

Section : BAC informatique de gestion

Année académique : 2023-2024

Nous remercions tout d'abord M. Degand professeur à l'I.P.E.P.S. Tournai pour ses conseils, sa disponibilité et ses encouragements.

Nous remercions également notre père pour son soutien et ses idées.

Table des matières

Classe AffichagePers :

```
package Console;


import java.util.ArrayList;
import java.util.Iterator;


/**
 * La classe AffichagePers fournit des méthodes pour afficher les informations du personnel.
 */
public class AffichagePers {


    /**
     * Affiche la liste de tout le personnel.
     *
     * @param Person la liste du personnel à afficher
     */
    protected void AffichageListe(ArrayList<Personnel> Person) {
        // Vérifie si la liste est vide
        if (Person.isEmpty()) {
            System.out.println("Affichage impossible, pas de personnel !");
        } else {
            // Affiche les en-têtes du tableau
             System.out.println("+----------------------+------------------+--------------------+--------+-------------+------------------------+");
            System.out.println("| Département          | Prénom           | Nom                | Sexe   | Naissance   | Email                  |");
            System.out.println("+----------------------+------------------+--------------------+--------+-------------+------------------------+");
```

```
        // Utilise StringBuilder pour construire la chaîne de caractères
        StringBuilder sb = new StringBuilder();

        // Utilise un itérateur pour parcourir la liste du personnel
        Iterator<Personnel> iterator = Person.iterator();

        while (iterator.hasNext()) {
            Personnel Pers = iterator.next();
            // Vérifie si le membre du personnel est actif
            if (Pers.isActif()) {
                // Ajoute les informations formatées au StringBuilder
                sb.append(" ");
                sb.append(FixedLenght.setFixedLength(Pers.getDepartement(), 25));
                sb.append(FixedLenght.setFixedLength(Pers.getPrenom(), 20));
                sb.append(FixedLenght.setFixedLength(Pers.getNom(), 20));
                sb.append(FixedLenght.setFixedLength(Pers.getSexe(), 10));
                sb.append(FixedLenght.setFixedLength(Pers.getDateddMMyyyy(), 15));
                sb.append(FixedLenght.setFixedLength(Pers.getEmail(), 25));
                sb.append("\n");
            }
        }
        // Affiche la chaîne de caractères construite
        System.out.println(sb.toString());
    }
}

/**
 * Affiche les informations d'une seule personne.
```

```
     *
     * @param pers l'objet Personnel contenant les informations à afficher
     */
    protected void
AffichageUnePersonne(Personnel pers) {
        // Vérifie si l'objet Personnel est null
        if (pers == null) {
            System.out.println("Affichage impossible, pas de personnel !");
        } else {
            // Affiche les en-têtes du tableau
            System.out.println("+----------------------+------------------+-------------------+-----
---+-------------+------------------------+");
            System.out.println("| Département          | Prénom           | Nom               | Sexe   |
Naissance    | Email                  |");
            System.out.println("+----------------------+------------------+-------------------+-----
---+-------------+------------------------+");

            // Utilise StringBuilder pour construire la chaîne de caractères
            StringBuilder sb = new StringBuilder();

            // Ajoute les informations formatées au StringBuilder
            sb.append(" ");
            sb.append(FixedLenght.setFixedLength(pers.getDepartement(), 25));
            sb.append(FixedLenght.setFixedLength(pers.getPrenom(), 20));
            sb.append(FixedLenght.setFixedLength(pers.getNom(), 20));
            sb.append(FixedLenght.setFixedLength(pers.getSexe(), 10));
            sb.append(FixedLenght.setFixedLength(pers.getDateddMMyyyy(), 15));
            sb.append(FixedLenght.setFixedLength(pers.getEmail(), 25));
            sb.append("\n");
```

```
            // Affiche la chaîne de caractères construite
            System.out.println(sb.toString());
        }
    }
}
```

Remarques

- Les méthodes utilisent la classe `FixedLenght` pour formater les longueurs des champs. Assurez-vous que cette classe est correctement définie dans votre projet.
- La classe `Personnel` doit également être définie avec les méthodes `isActif()`, `getDepartement()`, `getPrenom()`, `getNom()`, `getSexe()`, `getDateddMMyyyy()`, et `getEmail()`.

Classe ClassMain :

```
package Console;

import java.io.IOException;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import javax.swing.plaf.nimbus.NimbusLookAndFeel;
import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.SAXException;



/**
 * La classe principale qui initialise l'interface graphique de l'application.
 * Elle configure le look and feel Nimbus et affiche la fenêtre principale.
 */
public class ClassMain {

    /**
     * Point d'entrée principal de l'application.
     *
     * @param args les arguments de la ligne de commande (non utilisés)
     * @throws SAXException si une erreur se produit lors de l'analyse SAX
     * @throws IOException en cas d'erreur d'entrée/sortie lors du traitement XML
     * @throws ParserConfigurationException en cas d'erreur de configuration du parseur XML
     */
public static void main(String[] args) throws SAXException, IOException, ParserConfigurationException {
```

```
        try {
            // Définit le look and feel Nimbus pour l'interface graphique
            UIManager.setLookAndFeel(new NimbusLookAndFeel());
        } catch (UnsupportedLookAndFeelException e) {
            // En cas d'échec de configuration du look and feel, affiche l'erreur
            e.printStackTrace();
        }
        // Crée une nouvelle instance de la fenêtre principale GestionJFrame
        GestionJFrame fenetre = new GestionJFrame();
        // Rend la fenêtre visible à l'écran
        fenetre.setVisible(true);
    }
}
```

Remarques

- Assurez-vous que la classe `GestionJFrame` est correctement définie et qu'elle hérite probablement de `JFrame` ou d'une classe similaire pour représenter la fenêtre principale de l'application.
- Les exceptions `SAXException`, `IOException`, et `ParserConfigurationException` sont incluses dans la signature de la méthode `main`, ce qui signifie que le code qui les déclenche doit être correctement géré ou propagé.

Classe ControleSaisie :

```
package Console;


import java.io.File;
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;


/**
 * La classe ControleSaisie fournit des méthodes de validation pour différents types de saisie utilisateur.
 */
public class ControleSaisie {


    // Définition des motifs regex pour les différents types de validations
    private static final String EMAIL_PATTERN = "^[\\w_+&*-]+(?:\\.[\\w_+&*-]+)*@(?:[\\w-]+\\.)+[\\w]{2,7}$";
    private static Pattern patternEmail = Pattern.compile(EMAIL_PATTERN);


    private static final String NAME_PATTERN = "^([ \\u00c0-\\u01ffa-zA-Z'\\-])+$";
    private static Pattern patternNom = Pattern.compile(NAME_PATTERN);


    private static final String DATE_PATTERN = "^(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[012])/((19|2[0-9])[0-9]{2})$";
    private static Pattern patternDate = Pattern.compile(DATE_PATTERN);


    private static final String DEPT_PATTERN = "^([1-4]{1})$";
    private static Pattern patternDept = Pattern.compile(DEPT_PATTERN);
```

```
    /**
     * Valide un nom selon un motif et une taille maximale.
     *
     * @param text le texte à valider
     * @param taille la taille maximale autorisée pour le nom
     * @return true si le nom est valide, sinon false
     */
    public static boolean valideNom(String text, int taille) {
        String nullString = null;
        String vide = new String();
        boolean estNull = text.equals(nullString); // Vérifie si le texte est null
        boolean estVide = text.equals(vide); // Vérifie si le texte est vide

        // Si le texte n'est ni null ni vide
        if (!estNull && !estVide) {
            Matcher matcher = patternNom.matcher(text); // Applique le motif regex pour les noms
            // Vérifie si le texte correspond au motif et sa longueur est inférieure ou égale à la taille spécifiée
            if (matcher.matches() && text.length() <= taille) {
                return true;
            } else {
                System.out.println("Valeur hors contraintes !");
                return false;
            }
        } else {
            System.out.println("Valeur null ou vide !");
            return false;
        }
    }
```

```
    /**
     * Valide si le nom de fichier est valide.
     *
     * @param file le nom du fichier à valider
     * @return true si le nom de fichier est valide, sinon false
     */
    public static boolean valideNomFichier(String file) {
        File fichier = new File(file);
        boolean isValid = true;

        try {
            // Essaie de créer un nouveau fichier et de le supprimer immédiatement après
            if (fichier.createNewFile()) {
                fichier.delete();
            }
        } catch (IOException var4) {
            isValid = false; // Si une exception est levée, le fichier n'est pas valide
        }

        return isValid;
    }

    /**
     * Valide une adresse email.
     *
     * @param hex l'email à valider
     * @return true si l'email est valide, sinon false
     */
```

```
    public static boolean valideEmail(String hex) {
        Matcher matcher = patternEmail.matcher(hex);
        return matcher.matches(); // Retourne vrai si l'email correspond au motif
    }

    /**
     * Valide une réponse oui/non (o/n).
     *
     * @param result la réponse à valider
     * @return true si la réponse est "o" ou "n", sinon false
     */
    protected static boolean oui_non(String result) {
        return result.trim().equals("o") || result.trim().equals("n");
    }

    /**
     * Valide une réponse un/deux (1/2).
     *
     * @param result la réponse à valider
     * @return true si la réponse est "1" ou "2", sinon false
     */
    protected static boolean un_deux(String result) {
        return result.trim().equals("1") || result.trim().equals("2");
    }


    /**
     * Valide une date.
     *
```

```
     * @param hex la date à valider
     * @return true si la date est valide, sinon false
     */
    public static boolean valideDate(String hex) {
        Matcher matcher = patternDate.matcher(hex);
        return matcher.matches(); // Retourne vrai si la date correspond au motif
    }

    /**
     * Valide un département.
     *
     * @param hex le département à valider
     * @return true si le département est valide, sinon false
     */
    public static boolean valideDept(String hex) {
        Matcher matcher = patternDept.matcher(hex);
        return matcher.matches(); // Retourne vrai si le département correspond au motif
    }
}
```

Remarques

- Les motifs regex sont définis comme des constantes statiques pour la validation de différents types de saisie utilisateur.
- Les méthodes de validation retournent `true` si la saisie correspond au motif défini et respecte les contraintes spécifiées, sinon elles retournent `false`.
- Les méthodes `oui_non` et `un_deux` sont protégées, ce qui signifie qu'elles ne sont accessibles que dans le même package ou par des sous-classes.

Classe Departement :

```
package Console;

/**
* L'énumération Departement définit les différents départements d'une organisation
* avec leurs noms complets respectifs.
*/
public enum Departement {

  /** Département de la Comptabilité */
  Compta("Comptabilité"),

  /** Département des Ressources Humaines */
  HR("Ressources Humaines"),

  /** Département de la Production */
  Prod("Production"),

  /** Département de la Sécurité */
  SEC("Sécurité");

  /** Nom complet du département */
  private String deptNom;
  /**
   * Constructeur privé pour initialiser le nom complet du département.
   *
   * @param param3 le nom complet du département
   */
```

```
  private Departement(String param3) {
    this.deptNom = param3;
  }

  /**
   * Obtient le nom complet du département.
   *
   * @return le nom complet du département
   */
  public String getDeptNom() {
    return this.deptNom;
  }

  /**
   * Méthode synthétique générée par le compilateur pour obtenir toutes les valeurs de l'énumération.
   * Cette méthode n'est généralement pas visible dans le code source.
   *
   * @return un tableau contenant toutes les valeurs de l'énumération Departement
   */
  private static Departement[] $values() {
    return new Departement[]{Compta, HR, Prod, SEC};
  }
}
```

Remarques

- Chaque constante de l'énumération `Departement` est associée à un nom complet du département.
- Le constructeur de l'énumération est privé et initialise la variable d'instance `deptNom`.

- La méthode `getDeptNom` retourne le nom complet du département.
- La méthode `$values` est une méthode synthétique, générée par le compilateur pour obtenir toutes les valeurs de l'énumération. Cette méthode n'est généralement pas incluse dans la documentation utilisateur, mais elle est présente pour la complétude du code.

Classe Emprunt :

```
package Console;


/**
* La classe Emprunt représente un emprunt de matériel par un membre du personnel.
* Elle contient des informations sur l'emprunteur, le matériel emprunté et le nombre
        d'articles empruntés.
*/
public class Emprunt {
  /** L'emprunteur du matériel */
  protected Personnel emprunteur;

  /** Le matériel emprunté */
  protected Produit materiel;

  /** Le nombre d'articles empruntés */
  protected int nombre;

  /**
   * Constructeur pour initialiser les variables d'instance.
   *
   * @param emprunteur le membre du personnel qui emprunte le matériel
   * @param materiel le matériel emprunté
   * @param nombre le nombre d'articles empruntés
   */
  public Emprunt(Personnel emprunteur, Produit materiel, int nombre) {
    this.setEmprunteur(emprunteur); // Appel de la méthode setEmprunteur pour
        initialiser l'emprunteur
    this.materiel = materiel; // Initialisation du matériel emprunté
```

```
    this.nombre = nombre; // Initialisation du nombre d'articles empruntés
  }

  /**
   * Obtient l'emprunteur.
   *
   * @return l'emprunteur
   */
  public Personnel getEmprunteur() {
    return this.emprunteur;
  }

  /**
   * Définit l'emprunteur.
   *
   * @param emprunteur le membre du personnel qui emprunte le matériel
   */
  public void setEmprunteur(Personnel emprunteur) {
    this.emprunteur = emprunteur;
  }
  /**
   * Convertit l'objet Emprunt en une chaîne de caractères.
   *
   * @return une chaîne de caractères représentant l'emprunt
   */
  public String toString() {
    int var10000 = this.nombre;
    return var10000 + " " + this.materiel.toString() + " emprunté par " +
            String.valueOf(this.getEmprunteur());
```

```
  }

  /**
   * Obtient le matériel emprunté.
   *
   * @return le matériel emprunté
   */
  public Produit getMateriel() {
    return this.materiel;
  }

  /**
   * Obtient une description de l'article emprunté.
   *
   * @return une chaîne de caractères décrivant l'article emprunté
   */
  public String getArticle() {
    String var10000 = this.materiel.getNom();
    return var10000 + "  " + this.materiel.getDescription();
  }
  /**
   * Obtient le nombre d'articles empruntés.
   *
   * @return le nombre d'articles empruntés
   */
  public int getNombre() {
    return this.nombre;
  }
}
```

Remarques

- La classe `Emprunt` représente un emprunt de matériel par un membre du personnel.
- Les variables d'instance `emprunteur`, `materiel`, et `nombre` stockent respectivement l'emprunteur, le matériel emprunté, et le nombre d'articles empruntés.
- Le constructeur initialise ces variables d'instance.
- Les méthodes `getEmprunteur`, `setEmprunteur`, `getMateriel`, `getArticle`, et `getNombre` permettent de récupérer ou définir les informations concernant l'emprunt.
- La méthode `toString` convertit l'objet `Emprunt` en une chaîne de caractères, affichant le nombre d'articles empruntés, le matériel et l'emprunteur.

Classe FixedLenght :

```
package Console;


/**
* La classe FixedLenght fournit une méthode utilitaire pour ajuster la longueur d'une chaîne de caractères.
*/
public class FixedLenght {


  /**
   * Ajuste la longueur d'une chaîne de caractères à une longueur spécifiée en ajoutant des espaces à la fin.
   *
   * @param source la chaîne de caractères à ajuster
   * @param longueur la longueur souhaitée de la chaîne
   * @return la chaîne de caractères ajustée avec des espaces ajoutés pour atteindre la longueur spécifiée
   */
  protected static String setFixedLength(String source, int longueur) {
    // Ajoute des espaces à la chaîne jusqu'à ce qu'elle atteigne la longueur souhaitée
    while (source.length() < longueur) {
      char espace = ' ';
      source = source + espace;
    }


    return source; // Retourne la chaîne de caractères avec la longueur ajustée
  }
}
```

Remarques

- La classe `FixedLenght` contient une méthode utilitaire pour ajuster la longueur d'une chaîne de caractères en ajoutant des espaces à la fin.
- La méthode `setFixedLength` prend en paramètres la chaîne de caractères à ajuster (`source`) et la longueur souhaitée (`longueur`).
- La méthode ajoute des espaces à la chaîne de caractères jusqu'à ce qu'elle atteigne la longueur spécifiée, puis retourne la chaîne ajustée.

Classe GestionEmprunt :

```
package Console;

import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Iterator;

/**
* La classe GestionEmprunt gère les emprunts de matériel par le personnel.
* Elle permet d'ajouter des produits au magasin, de gérer les emprunts et les retours d'emprunts.
*/
public class GestionEmprunt {
  // Variable statique pour la saisie des entrées utilisateur
  private static Lire saisie = new Lire();

  /**
   * Méthode statique pour gérer le magasin.
   * Ajoute des produits au magasin et affiche la liste des produits disponibles.
   *
   * @param mag l'objet Magasin représentant le magasin de produits
   */
  static void Magasin(Magasin mag) {
    mag.AjouterProduit(); // Ajoute des produits au magasin
    mag.listeMap(); // Affiche la liste des produits du magasin
  }

  /**
```

```
    * Méthode protégée pour gérer les emprunts de matériel par le personnel.
    * Ajoute des emprunts à la liste des prêts et les affiche.
    *
    * @param mag l'objet Magasin contenant les produits disponibles
    * @param Person la liste du personnel
    * @param pret la liste des emprunts
    */
   protected void Emprunt(Magasin mag, ArrayList<Personnel> Person,
ArrayList<Emprunt> pret) {
      // Vérifie si la liste du personnel est vide
      if (Person.isEmpty()) {
         System.out.println("Emprunt impossible, pas de personnel !");
      // Vérifie si le magasin est vide
      } else if (mag.isEmpty()) {
         System.out.println("Emprunt impossible, magasin est vide !");
      } else {
         // Ajoute des emprunts à la liste des prêts
         pret.add(new Emprunt(Person.get(0), mag.getProduit("Id_1"), 1));
         pret.add(new Emprunt(Person.get(1), mag.getProduit("Id_3"), 1));
         pret.add(new Emprunt(Person.get(2), mag.getProduit("Id_4"), 1));
         pret.add(new Emprunt(Person.get(3), mag.getProduit("Id_6"), 1));
         pret.add(new Emprunt(Person.get(6), mag.getProduit("Id_7"), 1));

         // Itère sur la liste des emprunts et les affiche
         Iterator<Emprunt> var4 = pret.iterator();
         while (var4.hasNext()) {
            Emprunt emprunt = var4.next();
            PrintStream var10000 = System.out;
            String var10001 = String.valueOf(emprunt.getEmprunteur());
```

```
            var10000.println(var10001 + "\t" + String.valueOf(emprunt.getMateriel()));
        }
    }
}

/**
 * Méthode protégée pour gérer le retour des emprunts.
 * Affiche la liste des emprunts et permet à l'utilisateur de supprimer un emprunt.
 *
 * @param pret la liste des emprunts
 */
protected void RetourEmprunt(ArrayList<Emprunt> pret) {
    // Vérifie si la liste des emprunts est vide
    if (pret.isEmpty()) {
        System.out.println("Aucun emprunt disponible !");
    } else {
        System.out.println("liste des emprunts");
        int i = 1;

        // Itère sur la liste des emprunts et les affiche avec un numéro
        for (Iterator<Emprunt> var3 = pret.iterator(); var3.hasNext(); ++i) {
            Emprunt emprunt = var3.next();
            System.out.println("N° " + i + " " + emprunt.getEmprunteur().getNom() + "\t" +
emprunt.getArticle());
        }

        // Demande à l'utilisateur d'introduire le numéro d'emprunt à annuler
        System.out.println("Introduire le numéro d'emprunt à annuler : ");
        int nuEmprunt = saisie.saisieInt();
```

```
        // Vérifie si le numéro d'emprunt est valide et supprime l'emprunt correspondant
        if (nuEmprunt > 0 && nuEmprunt <= i - 1) {
          pret.remove(nuEmprunt - 1);
        } else {
          System.out.println("Le numéro d'emprunt indiqué n'existe pas !");
        }

        int y = 1;

        // Affiche la liste mise à jour des emprunts
        for (Iterator<Emprunt> var5 = pret.iterator(); var5.hasNext(); ++y) {
          Emprunt emprunt = var5.next();
          System.out.println("N° " + y + " " + emprunt.getEmprunteur().getNom() + "\t" +
emprunt.getArticle());
        }
      }
    }
}
```

Remarques

- La classe `GestionEmprunt` gère les emprunts de matériel par le personnel.
- La méthode `Magasin` ajoute des produits au magasin et affiche la liste des produits disponibles.
- La méthode `Emprunt` ajoute des emprunts à la liste des prêts et les affiche, vérifiant d'abord si la liste du personnel et le magasin ne sont pas vides.
- La méthode `RetourEmprunt` gère le retour des emprunts, affichant la liste des emprunts et permettant à l'utilisateur de supprimer un emprunt après avoir entré son numéro.

Classe GestionJFrame :

```
package Console;


import java.awt.Component;
import java.awt.GridLayout;
import java.io.IOException;
import java.util.ArrayList;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;


/**
* La classe GestionJFrame représente l'interface graphique de gestion du personnel et des emprunts de matériel.
* Elle permet d'afficher et de manipuler les données de personnel, de gérer les emprunts et retours de matériel, et de sauvegarder les informations.
*/
public class GestionJFrame extends JFrame {
  // Déclaration des boutons de l'interface utilisateur
  protected JButton btnLoad;
  protected JButton btnAffichage;
  protected JButton btnMag;
  protected JButton btnPret;
  protected JButton btnRetour;
  protected JButton btnModifPersonnel;
  protected JButton btnSupprimerPersonnel;
  protected JButton btnAjoutPersonnel;
```

```
protected JButton btnSauvegarde;
protected JButton btnClose;

// Déclaration des listes de personnel et d'emprunts, et des objets pour la gestion
private ArrayList<Personnel> Person = new ArrayList<>();
private Magasin mag = new Magasin();
private ArrayList<Emprunt> pret = new ArrayList<>();
private Sauvegarde save = new Sauvegarde();
private AffichagePers affichePers = new AffichagePers();
private GestionPersonnel gestpers = new GestionPersonnel();
private GestionEmprunt gestemprunt = new GestionEmprunt();

/**
 * Constructeur de la classe GestionJFrame.
 * Initialise l'interface graphique, configure les boutons et définit les actions à exécuter lors de l'interaction avec les boutons.
 */
public GestionJFrame() {
    super("Gestion Personnel & prêt matériel.");
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Définir l'opération de fermeture par défaut
    JPanel contentPane = (JPanel) this.getContentPane();

    // Création des textes pour les boutons avec HTML pour le formatage
    String texteBtnAfficher = "<html><Div Align=Center> Affichage <BR> données personnel </Div></html>";
    String texteBtnModifPersonnel = "<html><Div Align=Center> Modification <BR> données personnel </Div></html>";
    String texteBtnLoad = "<html><Div Align=Center> Charger <BR> données personnel </Div></html>";
```

```
// Initialisation des boutons avec les textes correspondants
this.btnLoad = new JButton(texteBtnLoad);
this.btnAffichage = new JButton(texteBtnAfficher);
this.btnMag = new JButton("Création du magasin");
this.btnPret = new JButton("Prêt de matériel");
this.btnRetour = new JButton("Retour de matériel");
this.btnModifPersonnel = new JButton(texteBtnModifPersonnel);
this.btnSupprimerPersonnel = new JButton("Supprimer une personne");
this.btnAjoutPersonnel = new JButton("Ajouter une personne");
this.btnSauvegarde = new JButton("Sauvegarde");
this.btnClose = new JButton("Fermer");

// Définir la disposition du panneau principal
contentPane.setLayout(new GridLayout(3, 1));
// Ajouter les boutons au panneau principal
contentPane.add(this.btnLoad);
contentPane.add(this.btnAffichage);
contentPane.add(this.btnMag);
contentPane.add(this.btnPret);
contentPane.add(this.btnRetour);
contentPane.add(this.btnModifPersonnel);
contentPane.add(this.btnSupprimerPersonnel);
contentPane.add(this.btnAjoutPersonnel);
contentPane.add(this.btnSauvegarde);
contentPane.add(this.btnClose);

// Définir les actions à exécuter lorsque les boutons sont cliqués
this.btnLoad.addActionListener((e) -> {
  try {
```

```
        this.gestpers.LoadPersonnel(this.Person); // Charger les données du personnel
    } catch (SAXException | IOException | ParserConfigurationException ex) {
        throw new RuntimeException(ex); // Gérer les exceptions
    }
});

this.btnAffichage.addActionListener((e) -> {
    this.affichePers.AffichageListe(this.Person); // Afficher la liste du personnel
});

this.btnMag.addActionListener((e) -> {
    GestionEmprunt.Magasin(this.mag); // Gérer le magasin
});

this.btnPret.addActionListener((e) -> {
    this.gestemprunt.Emprunt(this.mag, this.Person, this.pret); // Gérer les emprunts
});

this.btnRetour.addActionListener((e) -> {
    this.gestemprunt.RetourEmprunt(this.pret); // Gérer le retour des emprunts
});

this.btnModifPersonnel.addActionListener((e) -> {
    this.gestpers.ModifPersonnel(this.Person); // Modifier les données du personnel
});

this.btnSupprimerPersonnel.addActionListener((e) -> {
    this.gestpers.SuppressionPersonnel(this.Person); // Supprimer une personne
});
```

```
        this.btnSauvegarde.addActionListener((e) -> {
            this.save.Sauvegarde(this.Person, this.pret); // Sauvegarder les données
        });

        this.btnAjoutPersonnel.addActionListener((e) -> {
            this.gestpers.AjoutPersonnel(this.Person); // Ajouter une personne
        });

        this.btnClose.addActionListener((e) -> {
            this.dispose(); // Fermer la fenêtre
        });

        // Définir la taille de la fenêtre et la positionner au centre de l'écran
        this.setSize(800, 600);
        this.setLocationRelativeTo(null);
    }
}
```

Remarques

- La classe `GestionJFrame` représente l'interface graphique pour la gestion du personnel et des emprunts de matériel.
- Le constructeur initialise l'interface, configure les boutons et définit les actions à exécuter lors de l'interaction avec les boutons.
- Les actions incluent le chargement des données du personnel, l'affichage des données du personnel, la gestion du magasin, des emprunts et des retours de matériel, la modification et la suppression des données du personnel, l'ajout de nouvelles personnes et la sauvegarde des données.

Classe GestionPersonnel :

```
package Console;


import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.xml.sax.SAXException;


/**
* La classe GestionPersonnel gère les opérations liées au personnel, telles que le
        chargement, la modification,
* la suppression et l'ajout de membres du personnel à partir d'un fichier XML ou via des
        entrées utilisateur.
*/
public class GestionPersonnel {
   private AffichagePers affichePers = new AffichagePers();
   private String nom;
   private int tailleNom = 30;
   private ControleSaisie ctlSaisie = new ControleSaisie();
   private InputData inputData = new InputData();
   private WordUtils wordutils = new WordUtils();


   /**
    * Charge les données du personnel à partir d'un fichier XML et les ajoute à la liste
        donnée.
    *
```

```
   * @param Person La liste de personnel à laquelle les données chargées seront ajoutées.
   * @throws SAXException Si une erreur survient lors de l'analyse du fichier XML.
   * @throws IOException Si une erreur d'entrée/sortie survient.
   * @throws ParserConfigurationException Si une erreur de configuration du parseur
        survient.
   */
  protected void LoadPersonnel(ArrayList<Personnel> Person) throws SAXException,
      IOException, ParserConfigurationException {
    if (!Person.isEmpty()) {
      System.out.println("Le personnel est déjà chargé !");
    } else {
      for (int temp = 0; temp < PersonnelXml.Lecture().getLength(); ++temp) {
        Node nNode = PersonnelXml.Lecture().item(temp);
        if (nNode.getNodeType() == Node.ELEMENT_NODE) {
          Element courant = (Element) nNode;
          String nom =
        courant.getElementsByTagName("nom").item(0).getTextContent();
          String prenom =
        courant.getElementsByTagName("prenom").item(0).getTextContent();
          String sex =
        courant.getElementsByTagName("sexe").item(0).getTextContent().trim();
          Sexe tempSex;
          if (sex.equals("HOMME")) {
            tempSex = Sexe.HOMME;
          } else {
            tempSex = Sexe.FEMME;
          }
          String dateNais =
        courant.getElementsByTagName("dateNaissance").item(0).getTextContent();
          int jour = Integer.parseInt(dateNais.substring(0, dateNais.indexOf("/")));
```

```
            int mois = Integer.parseInt(dateNais.substring(dateNais.indexOf("/", 0) + 1,
        dateNais.indexOf("/", 3)));
            int annee = Integer.parseInt(dateNais.substring(dateNais.length() - 4));
            String mail =
        courant.getElementsByTagName("mail").item(0).getTextContent();
            String Depart =
        courant.getElementsByTagName("departement").item(0).getTextContent().trim();
            Departement tempdep;
            switch (Depart) {
              case "Compta":
                tempdep = Departement.Compta;
                break;
              case "HR":
                tempdep = Departement.HR;
                break;
              case "Prod":
                tempdep = Departement.Prod;
                break;
              default:
                tempdep = Departement.SEC;
            }

            Person.add(new Personnel(nom, prenom, tempSex, new MyDate(jour, mois,
        annee), mail, tempdep));
          }
        }
      }
    }

    /**
```

```
    * Modifie les données d'un membre du personnel dans la liste donnée.
    *
    * @param Person La liste de personnel contenant le membre à modifier.
    */
  protected void ModifPersonnel(ArrayList<Personnel> Person) {
    this.affichePers.AffichageListe(Person);
    boolean egal = false;
    System.out.println("Introduire le nom de la personne à modifier :");
    this.nom = InputData.inputNomPrenom("nom", this.tailleNom).trim();

    for (int count = 0; count < Person.size(); ++count) {
      if (this.nom.equalsIgnoreCase(Person.get(count).getNom().trim())) {
        this.affichePers.AffichageUnePersonne(Person.get(count));
        System.out.println("Introduire les nouvelles valeurs : ");

        Person.get(count).setNom(WordUtils.mettreEnMajscule(InputData.inputNomPren
        om("nom", this.tailleNom)));

        Person.get(count).setPrenom(WordUtils.mettreEnMajscule(InputData.inputNomPr
        enom("prenom", this.tailleNom)));

        Person.get(count).setEmail(WordUtils.mettreEnMinusculeMail(InputData.inputE
        mail()));
        egal = true;
      }
    }

    if (!egal) {
      System.out.println("Pas de correspondance !");
    }
  }
```

```
  /**
   * Supprime un membre du personnel de la liste donnée en le marquant comme inactif.
   *
   * @param Person La liste de personnel contenant le membre à supprimer.
   */
  protected void SuppressionPersonnel(ArrayList<Personnel> Person) {
    this.affichePers.AffichageListe(Person);
    boolean egal = false;
    System.out.println("Introduire le nom de la personne à supprimer :");
    this.nom = InputData.inputNomPrenom("nom", this.tailleNom).trim();
    Iterator<Personnel> var3 = Person.iterator();

    while (var3.hasNext()) {
      Personnel personnel = var3.next();
      if (personnel.getNom().contains(this.nom)) {
        this.affichePers.AffichageUnePersonne(personnel);
        if (InputData.suppression()) {
          personnel.setActif(false);
          System.out.println("La suppression a eu lieu !");
          egal = true;
          break;
        }
      }
    }

    if (!egal) {
      System.out.println("La suppression n'a pas eu lieu !");
    }
```

```
}

/**
 * Ajoute un nouveau membre du personnel à la liste donnée.
 *
 * @param person La liste de personnel à laquelle le nouveau membre sera ajouté.
 */
public void AjoutPersonnel(ArrayList<Personnel> person) {
    System.out.println("Introduire les données du nouveau membre du personnel.");
    String nom = WordUtils.mettreEnMajscule(InputData.inputNomPrenom("nom",
        this.tailleNom));
    String prenom =
        WordUtils.mettreEnMajscule(InputData.inputNomPrenom("prenom",
        this.tailleNom));
    String sex = InputData.inputSexe();
    Sexe tempSex = sex.equals("2") ? Sexe.HOMME : Sexe.FEMME;
    String email = WordUtils.mettreEnMinusculeMail(InputData.inputEmail());
    String dateNaissance = InputData.inputDateNaissance();
    int jour = Integer.parseInt(dateNaissance.substring(0, dateNaissance.indexOf("/")));
    int mois = Integer.parseInt(dateNaissance.substring(dateNaissance.indexOf("/", 0) +
        1, dateNaissance.indexOf("/", 3)));
    int annee = Integer.parseInt(dateNaissance.substring(dateNaissance.length() - 4));
    String depart = InputData.InputDepartement();
    Departement tempdep;
    switch (depart) {
      case "1":
        tempdep = Departement.Compta;
        break;
      case "2":
        tempdep = Departement.HR;
```

```
            break;
        case "3":
            tempdep = Departement.Prod;
            break;
        default:
            tempdep = Departement.SEC;
    }

    Personnel pers = new Personnel(nom, prenom, tempSex, new MyDate(jour, mois,
        annee), email, tempdep);
    person.add(pers);
  }
}
```

Remarques

- La classe `GestionPersonnel` gère les opérations liées au personnel, telles que le chargement des données depuis un fichier XML, la modification, la suppression et l'ajout de membres du personnel.
- Les méthodes de la classe utilisent diverses classes utilitaires pour gérer les saisies et les opérations sur les données du personnel.
- Les méthodes sont protégées (protected) pour limiter l'accès aux classes du même package ou aux sous-classes.

Classe Individu :

```
package Console;


/**
* La classe Individu représente une personne avec des attributs tels que le nom, le prénom, le sexe et la date de naissance.
*/
public class Individu {
  private String nom;
  private String prenom;
  private Sexe sexe;
  private MyDate dateNaissance;

  /**
   * Constructeur de la classe Individu.
   *
   * @param nom Le nom de l'individu.
   * @param prenom Le prénom de l'individu.
   * @param sexe Le sexe de l'individu (HOMME ou FEMME).
   * @param dateNaissance La date de naissance de l'individu.
   */
  public Individu(String nom, String prenom, Sexe sexe, MyDate dateNaissance) {
    this.nom = nom;
    this.prenom = prenom;
    this.sexe = sexe;
    this.dateNaissance = dateNaissance;
  }
```

```
    /**
     * Obtient le nom de l'individu.
     *
     * @return Le nom de l'individu.
     */
    public String getNom() {
        return this.nom;
    }

    /**
     * Définit le nom de l'individu.
     *
     * @param nom Le nouveau nom de l'individu.
     */
    public void setNom(String nom) {
        this.nom = nom;
    }

    /**
     * Obtient le prénom de l'individu.
     *
     * @return Le prénom de l'individu.
     */
    public String getPrenom() {
        return this.prenom;
    }

    /**
     * Définit le prénom de l'individu.
```

```
    *
    * @param prenom Le nouveau prénom de l'individu.
    */
   public void setPrenom(String prenom) {
      this.prenom = prenom;
   }

   /**
    * Obtient le sexe de l'individu.
    *
    * @return Le sexe de l'individu.
    */
   public String getSexe() {
      return this.sexe.getLabel();
   }
   /**
    * Définit le sexe de l'individu.
    *
    * @param sexe Le nouveau sexe de l'individu (HOMME ou FEMME).
    */
   public void setSexe(Sexe sexe) {
      this.sexe = sexe;
   }

   /**
    * Obtient la date de naissance de l'individu.
    *
    * @return La date de naissance de l'individu.
    */
```

```
  public MyDate getDateNaissance() {
    return this.dateNaissance;
  }

  /**
   * Définit la date de naissance de l'individu.
   *
   * @param dateNaissance La nouvelle date de naissance de l'individu.
   */
  public void setDateNaissance(MyDate dateNaissance) {
    this.dateNaissance = dateNaissance;
  }

  /**
   * Obtient la date de naissance formatée de l'individu.
   *
   * @return La date de naissance formatée (jj-mm-aaaa).
   */
  public String getDateddMMyyyy() {
    return String.format("%d-%d-%d", this.dateNaissance.getJour(),
this.dateNaissance.getMois(), this.dateNaissance.getAnnee());
  }

  /**
   * Obtient la représentation sous forme de chaîne de caractères de l'individu.
   *
   * @return Une chaîne de caractères représentant l'individu.
   */
  @Override
```

```
  public String toString() {
    return String.format("%s %s sexe : %s date de naissance : %d-%d-%d.", this.nom, this.prenom, this.getSexe(), this.dateNaissance.getJour(), this.dateNaissance.getMois(), this.dateNaissance.getAnnee());
  }
}
```

Remarques

- La classe `Individu` représente une personne avec des attributs basiques : nom, prénom, sexe et date de naissance.
- Le constructeur permet d'initialiser ces attributs lors de la création d'un objet `Individu`.
- Les méthodes `get` et `set` permettent d'accéder et de modifier ces attributs.
- La méthode `getDateddMMyyyy` retourne la date de naissance formatée en chaîne de caractères.
- La méthode `toString` fournit une représentation textuelle complète de l'objet `Individu`.

Classe InputData :

```
package Console;

/**
* La classe InputData gère les saisies de données avec validation.
*/
public class InputData {
  private static Lire saisie = new Lire(); // Instance de la classe Lire pour la saisie

  /**
   * Méthode pour saisir le nom ou le prénom avec validation de la longueur maximale.
   *
   * @param libelle Le libellé (nom ou prénom).
   * @param tailleNom La longueur maximale du nom ou prénom.
   * @return La chaîne de caractères saisie et validée.
   */
  protected static String inputNomPrenom(String libelle, int tailleNom) {
    boolean arret = false; // Booléen pour contrôler la boucle
    String result;

    for(result = null; !arret; arret = false) { // Boucle de saisie jusqu'à ce que la saisie soit valide
      if (libelle.equals("nom")) {
        result = saisie.SaiseTexte("Veuillez introduire le nom : "); // Saisie du nom
      } else {
        result = saisie.SaiseTexte("Veuillez introduire le prénom : "); // Saisie du prénom
      }
```

```
        if (ControleSaisie.valideNom(result, tailleNom)) { // Validation du nom ou prénom saisi
            return result; // Retourne le résultat si valide
        }

        System.out.print("Erreur de saisie !  "); // Message en cas de saisie invalide
    }

    return result; // Retourne le résultat
  }

  /**
   * Méthode pour saisir une adresse e-mail avec validation.
   *
   * @return L'adresse e-mail saisie et validée.
   */
  protected static String inputEmail() {
    boolean arret = false; // Booléen pour contrôler la boucle
    String email;

    for(email = null; !arret; arret = false) { // Boucle de saisie jusqu'à ce que l'e-mail soit valide
        email = saisie.SaiseTexte("Veuillez introduire l'adresse mail : "); // Saisie de l'adresse e-mail
        if (ControleSaisie.valideEmail(email)) { // Validation de l'adresse e-mail
            return email; // Retourne l'e-mail si valide
        }

        System.out.print("Saisie non valide !  "); // Message en cas de saisie invalide
    }
```

```
    return email; // Retourne l'e-mail
  }

  /**
   * Méthode pour saisir un nom de fichier avec validation du chemin.
   *
   * @return Le chemin complet du fichier validé.
   */
  protected static String inputNomFichier() {
    boolean arret = false; // Booléen pour contrôler la boucle
    String result;

    for(result = null; !arret; arret = false) { // Boucle de saisie jusqu'à ce que le nom de fichier soit valide
      result = saisie.SaiseTexte(" "); // Saisie du nom de fichier
      if (ControleSaisie.valideNomFichier("./Fichier/" + result)) { // Validation du nom de fichier avec chemin
        return "./Fichier/" + result; // Retourne le chemin complet si valide
      }

      System.out.print("Saisie non valide !  "); // Message en cas de saisie invalide
    }

    return "./Fichier/" + result; // Retourne le chemin complet du fichier
  }

  /**
   * Méthode pour confirmer une suppression avec validation de la réponse.
   *
```

```
    * @return true si la suppression est confirmée, sinon false.
    */
   protected static boolean suppression() {
      boolean arret = false; // Booléen pour contrôler la boucle
      boolean result = false; // Résultat de la confirmation de suppression

      while(!arret) { // Boucle de saisie jusqu'à ce que la réponse soit valide
         String texte = saisie.SaiseTexte("Voulez-vous valider la suppression ? o/n "); // Saisie de la confirmation
         if (ControleSaisie.oui_non(texte)) { // Validation de la réponse (oui ou non)
            result = true; // Confirme la suppression
            arret = true; // Termine la boucle
         } else {
            System.out.print("Erreur de saisie !  "); // Message en cas de réponse invalide
         }
      }

      return result; // Retourne true si la suppression est confirmée, sinon false
   }

   /**
    * Méthode pour saisir le sexe avec validation du choix.
    *
    * @return Le choix (1 pour femme, 2 pour homme).
    */
   protected static String inputSexe() {
      boolean arret = false; // Booléen pour contrôler la boucle
      String texte = ""; // Texte pour le choix du sexe
```

```
    while(!arret) { // Boucle de saisie jusqu'à ce que le choix soit valide
      texte = saisie.SaiseTexte("Veuillez introduire 1. pour une FEMME ou 2. pour un HOMME : "); // Saisie du choix
      if (ControleSaisie.un_deux(texte)) { // Validation du choix (1 pour femme, 2 pour homme)
        arret = true; // Termine la boucle
      } else {
        System.out.print("Erreur de saisie !  "); // Message en cas de choix invalide
      }
    }

    return texte; // Retourne le choix (1 ou 2)
  }

  /**
   * Méthode pour saisir le département avec validation du choix.
   *
   * @return Le choix du département (1, 2, 3 ou 4).
   */
  protected static String InputDepartement() {
    boolean arret = false; // Booléen pour contrôler la boucle
    String texte = ""; // Texte pour le choix du département

    while(!arret) { // Boucle de saisie jusqu'à ce que le choix soit valide
      System.out.println("Département : ");
      System.out.println("Pour la comptabilité, \t\t\ttapez 1 ");
      System.out.println("Pour les ressources humaines, \ttapez 2 ");
      System.out.println("Pour la production, \t\t\ttapez 3");
      System.out.println("Pour la sécurité, \t\t\t\ttapez 4");
```

```
        texte = saisie.SaiseTexte("Veuillez introduire votre choix : "); // Saisie du choix du département
        if (ControleSaisie.valideDept(texte)) { // Validation du choix du département
          arret = true; // Termine la boucle
        } else {
          System.out.print("Erreur de saisie !  "); // Message en cas de choix invalide
        }
      }

      return texte; // Retourne le choix du département (1, 2, 3 ou 4)
    }

    /**
     * Méthode pour saisir la date de naissance avec validation.
     *
     * @return La date de naissance saisie et validée.
     */
    public static String inputDateNaissance() {
      boolean arret = false; // Booléen pour contrôler la boucle
      String date;

      for(date = null; !arret; arret = false) { // Boucle de saisie jusqu'à ce que la date soit valide
        date = saisie.SaiseTexte("Veuillez introduire la date de naissance : (jj/mm/aaaa)"); // Saisie de la date de naissance
        if (ControleSaisie.valideDate(date)) { // Validation de la date de naissance
          return date; // Retourne la date si valide
        }

        System.out.print("Saisie non valide !  "); // Message en cas de saisie invalide
```

```
    }

    return date; // Retourne la date de naissance
  }
}
```

Remarques

- La classe `InputData` gère les différentes saisies de données utilisateur avec validation.
- Les méthodes utilisent une instance de la classe `Lire` pour lire les entrées utilisateur.
- Chaque méthode boucle jusqu'à ce que la saisie soit validée par les méthodes de la classe `ControleSaisie`.
- Les méthodes retournent la valeur saisie une fois validée.
- Les messages d'erreur sont affichés en cas de saisie invalide.

Classe Lire :

```
package Console;

import java.util.NoSuchElementException;
import java.util.Scanner;

/**
* La classe Lire gère la saisie utilisateur via le clavier.
*/
public class Lire {
  private Scanner clavier; // Scanner pour la saisie au clavier

  /**
   * Constructeur de la classe Lire.
   * Initialise le Scanner avec l'entrée standard.
   */
  public Lire() {
    this.clavier = new Scanner(System.in); // Initialise le Scanner avec l'entrée standard
  }
  /**
   * Méthode pour saisir du texte avec gestion des exceptions.
   *
   * @param libelle Le libellé de saisie à afficher.
   * @return La chaîne de caractères saisie.
   */
  protected String SaiseTexte(String libelle) {
    String texte = null;
    System.out.print(libelle); // Affiche le libellé de saisie
```

```
    boolean loop = false;

    while (!loop) {
      try {
        texte = this.clavier.nextLine(); // Saisie du texte
        if (texte != null && !texte.isBlank()) { // Vérifie si la saisie n'est pas vide
          loop = true; // Sort de la boucle si la saisie est valide
        } else {
          System.out.println("Aucune valeur introduite !"); // Message en cas de saisie vide
        }
      } catch (NoSuchElementException var5) {
        System.out.println("Erreur, Aucune ligne n'a été trouvée"); // Exception si aucune ligne trouvée
      } catch (IllegalStateException var6) {
        System.out.println("Erreur, le Scanner est fermé"); // Exception si le Scanner est fermé
      }
    }

    return texte; // Retourne le texte saisi
  }

  /**
   * Méthode pour saisir un entier avec gestion des exceptions.
   *
   * @return L'entier saisi et validé.
   */
  protected int saisieInt() {
    boolean valid = false;
```

```
    int intValue = 0;


    while (!valid) {
      try {
        String strtValue = this.clavier.nextLine(); // Saisie de la valeur sous forme de chaîne
        intValue = Integer.parseInt(strtValue); // Conversion en entier
        valid = true; // Sort de la boucle si la conversion est réussie
      } catch (NumberFormatException var5) {
        System.out.println("Erreur veuillez saisir un nombre entier !"); // Exception si la conversion échoue
      } catch (NoSuchElementException var6) {
        System.out.println("Saisir un entier, aucune donnée trouvée !"); // Exception si aucune donnée trouvée
      } catch (IllegalStateException var7) {
        System.out.println("Saisir un entier, aucune saisie possible !"); // Exception si aucune saisie possible
      }
    }


    return intValue; // Retourne l'entier saisi
  }


  /**
   * Méthode privée pour vider le scanner en cas de problèmes récurrents.
   */
  private void vider() {
    try {
      this.clavier.nextLine(); // Vide le scanner en lisant la ligne suivante
    } catch (NoSuchElementException var2) {
```

```
            this.vider(); // Appel récursif pour vider en cas d'erreur
            System.out.println("if no line was found "); // Message en cas d'absence de ligne trouvée
        } catch (IllegalStateException var3) {
            this.vider(); // Appel récursif pour vider en cas d'erreur
            System.out.println("if this scanner is closed"); // Message si le scanner est fermé
        }
    }
}
```

Remarques

- La classe `Lire` utilise un `Scanner` pour lire les entrées utilisateur depuis le clavier.
- La méthode `SaiseTexte` permet de saisir du texte en affichant un libellé et en gérant les exceptions.
- La méthode `saisieInt` permet de saisir un entier en gérant les exceptions de conversion et de saisie.
- La méthode `vider` est utilisée pour vider le scanner en cas de problèmes récurrents avec la saisie.

Classe Magasin :

```
package Console;

import java.io.PrintStream;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;

/**
* La classe Magasin gère un ensemble de produits sous forme de Map.
*/
public class Magasin {
   static Map<String, Produit> produits; // Déclaration de la map de produits

   /**
    * Constructeur de la classe Magasin.
    * Initialise la map de produits.
    */
   public Magasin() {
      produits = new HashMap<>(); // Initialisation de la map de produits dans le
constructeur
   }

   /**
    * Méthode pour ajouter des produits prédéfinis dans le magasin.
    */
   public void AjouterProduit() {
```

```
        produits.put("Id_1", new Produit("HP", "Elitebook 850 G7"));
        produits.put("Id_2", new Produit("HP", "Elitebook 830 G7 X360"));
        produits.put("Id_3", new Produit("Dell", "Inspiron 15 3000"));
        produits.put("Id_4", new Produit("Dell", "XPS 13"));
        produits.put("Id_5", new Produit("Dell", "XPS 15"));
        produits.put("Id_6", new Produit("Lenovo", "Thinkpad E15 G2"));
        produits.put("Id_7", new Produit("Lenovo", "IdeaPad 3 14IIL05 81WD00B2MH"));
    }

    /**
     * Méthode pour afficher la liste des produits du magasin.
     */
    public void listeMap() {
        System.out.println("Le magasin est composé de " + produits.size() + " articles."); // Affiche le nombre d'articles dans le magasin
        Iterator<Entry<String, Produit>> iterator = produits.entrySet().iterator();

        while(iterator.hasNext()) {
            Entry<String, Produit> entry = iterator.next();
            System.out.println(entry.getKey() + " --- " + entry.getValue()); // Affiche chaque clé et valeur de la map
        }
    }

    /**
     * Méthode pour obtenir un produit à partir de son nom.
     *
     * @param nom Le nom du produit.
     * @return Le produit correspondant au nom ou null s'il n'existe pas.
```

```
    */
    public Produit getProduit(String nom) {
        return produits.containsKey(nom) ? produits.get(nom) : null; // Retourne le produit correspondant au nom ou null s'il n'existe pas
    }

    /**
     * Méthode pour vérifier si le magasin est vide.
     *
     * @return true si la map de produits est vide, sinon false.
     */
    public boolean isEmpty() {
        return produits.isEmpty(); // Retourne true si la map de produits est vide, sinon false
    }
}
```

Remarques

- La classe `Magasin` gère un ensemble de produits en utilisant une `Map`.
- Le constructeur initialise cette `Map` vide.
- La méthode `AjouterProduit` ajoute des produits prédéfinis à la `Map`.
- La méthode `listeMap` affiche la liste des produits présents dans le magasin.
- La méthode `getProduit` retourne un produit à partir de son nom ou `null` si le produit n'existe pas.
- La méthode `isEmpty` vérifie si la `Map` de produits est vide.

Classe MyDate :

```
package Console;

/**
* La classe MyDate représente une date avec jour, mois et année.
*/
public class MyDate {
   // Déclaration des variables membres privées pour jour, mois et année
   private int jour;
   private int mois;
   private int annee;

   /**
    * Constructeur de la classe MyDate.
    *
    * @param jour  Le jour de la date.
    * @param mois  Le mois de la date.
    * @param annee L'année de la date.
    */
   public MyDate(int jour, int mois, int annee) {
      // Initialisation des variables membres avec les valeurs passées en paramètres
      this.jour = jour;
      this.mois = mois;
      this.annee = annee;
   }

   /**
    * Méthode pour récupérer le jour de la date.
```

```
    *
    * @return Le jour de la date.
    */
   public int getJour() {
      return this.jour;
   }

   /**
    * Méthode pour récupérer le mois de la date.
    *
    * @return Le mois de la date.
    */
   public int getMois() {
      return this.mois;
   }

   /**
    * Méthode pour récupérer l'année de la date.
    *
    * @return L'année de la date.
    */
   public int getAnnee() {
      return this.annee;
   }
}
```

Remarques

- La classe `MyDate` représente une date avec trois attributs : jour, mois et année.

- Le constructeur `MyDate(int jour, int mois, int annee)` initialise ces attributs avec les valeurs fournies en paramètres.
- Les méthodes `getJour()`, `getMois()` et `getAnnee()` permettent d'accéder respectivement au jour, mois et année de l'objet `MyDate`.

Classe Personnel :

```
package Console;


/**
* La classe Personnel représente un individu appartenant à un département spécifique, héritant des caractéristiques de la classe Individu.
*/
public class Personnel extends Individu {
   private Departement departement; // Référence vers le département auquel le personnel appartient
   private int IdPersonnel; // Identifiant unique du personnel
   private static int id = 1; // Variable statique pour générer des identifiants uniques
   private String email; // Adresse email du personnel
   private boolean actif = true; // Indicateur d'activité du personnel, par défaut actif


   /**
    * Constructeur de la classe Personnel.
    *
    * @param nom          Le nom du personnel.
    * @param prenom       Le prénom du personnel.
    * @param sexe         Le sexe du personnel (FEMME ou HOMME).
    * @param dateNaissance La date de naissance du personnel.
    * @param email        L'adresse email du personnel.
    * @param dept         Le département auquel le personnel appartient.
    */
   public Personnel(String nom, String prenom, Sexe sexe, MyDate dateNaissance, String email, Departement dept) {
      // Appel au constructeur de la classe parent Individu avec les informations de base
      super(nom, prenom, sexe, dateNaissance);
```

```
    // Initialisation des variables membres avec les valeurs passées en paramètres
    this.departement = dept;
    this.IdPersonnel = id++; // Attribution d'un identifiant unique en incrémentant la variable statique id
    this.email = email;
  }

  /**
   * Méthode toString pour obtenir une représentation textuelle de l'objet Personnel.
   *
   * @return Une chaîne de caractères représentant l'objet Personnel.
   */
  @Override
  public String toString() {
    // Construction de la chaîne de caractères représentant l'objet Personnel
    return " Id : " + this.getIdPersonnel() + " " + super.toString() + " Email : " + this.getEmail() + " Département : " + this.getDepartement();
  }

  /**
   * Méthode pour récupérer l'identifiant unique du personnel.
   *
   * @return L'identifiant unique du personnel.
   */
  public int getIdPersonnel() {
    return this.IdPersonnel;
  }

  /**
```

```
    * Méthode pour récupérer le nom du département du personnel.
    *
    * @return Le nom du département du personnel.
    */
   public String getDepartement() {
      return this.departement.getDeptNom();
   }
   /**
    * Méthode pour récupérer l'email du personnel.
    *
    * @return L'adresse email du personnel.
    */
   public String getEmail() {
      return this.email;
   }

   /**
    * Méthode pour définir l'email du personnel.
    *
    * @param email La nouvelle adresse email du personnel.
    */
   public void setEmail(String email) {
      this.email = email;
   }

   /**
    * Méthode pour vérifier l'état d'activité du personnel.
    *
    * @return true si le personnel est actif, false sinon.
```

```
    */
    public boolean isActif() {
        return this.actif;
    }

    /**
     * Méthode pour définir l'état d'activité du personnel.
     *
     * @param actif Le nouvel état d'activité du personnel (true pour actif, false pour inactif).
     */
    public void setActif(boolean actif) {
        this.actif = actif;
    }
}
```

Remarques

- La classe `Personnel` représente un individu avec des attributs supplémentaires tels que l'identifiant unique, l'email, l'état d'activité et le département auquel il appartient.
- Le constructeur `Personnel(String nom, String prenom, Sexe sexe, MyDate dateNaissance, String email, Departement dept)` initialise les variables membres avec les valeurs fournies en paramètres, en plus de faire appel au constructeur de la classe parent `Individu`.
- La méthode `toString()` permet d'obtenir une représentation textuelle détaillée de l'objet `Personnel`, incluant son identifiant, ses informations de base (nom, prénom, sexe, date de naissance), son email et son département.
- Les méthodes getters et setters permettent d'accéder et de modifier les attributs de l'objet `Personnel` de manière contrôlée.

Classe PersonnelXML :

```
package Console;

import java.io.File;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

/**
* La classe PersonnelXml permet de lire et récupérer les éléments "Personnel" à partir d'un fichier XML.
*/
public class PersonnelXml {

  // Déclaration des variables statiques pour le document XML et l'élément racine
  public static Document document;
  public static Element racine;

  /**
   * Méthode pour lire et récupérer les éléments "Personnel" depuis le fichier XML.
   *
   * @return Une NodeList contenant tous les éléments "Personnel" du fichier XML.
   * @throws SAXException en cas d'erreur lors du parsing SAX.
```

```
    * @throws IOException en cas d'erreur d'entrée/sortie lors de la lecture du fichier XML.
    * @throws ParserConfigurationException en cas d'erreur de configuration du parseur XML.
    */
   static NodeList Lecture() throws SAXException, IOException, ParserConfigurationException {
       // Chemin vers le fichier XML
       File inputFile = new File("./Fichier/Personnel.xml");

       // Création des constructeurs de documents et de l'usine
       DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
       DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

       // Parsing du fichier XML pour créer un Document
       Document doc = dBuilder.parse(inputFile);
       // Normalisation du document XML pour s'assurer qu'il est correctement formaté
       doc.getDocumentElement().normalize();

       // Récupération de l'élément racine du document XML
       racine = doc.getDocumentElement();

       // Récupération de tous les éléments "Personnel" dans une NodeList
       NodeList listPers = racine.getElementsByTagName("Personnel");

       // Retourne la NodeList contenant tous les éléments "Personnel"
       return listPers;
   }
}
```

Remarques

- La classe `PersonnelXml` permet de lire un fichier XML contenant des éléments "Personnel" et de récupérer ces éléments sous forme d'une `NodeList`.
- La méthode statique `Lecture()` lit le fichier XML à partir du chemin spécifié (`./Fichier/Personnel.xml`), utilise un parseur DOM pour créer un document XML, le normalise pour s'assurer qu'il est correctement formaté, puis récupère tous les éléments "Personnel" dans une `NodeList`.
- Les exceptions `SAXException`, `IOException`, et `ParserConfigurationException` sont levées et doivent être gérées par l'appelant de la méthode en cas d'erreurs lors de la lecture ou du parsing du fichier XML.
- Les variables statiques `document` et `racine` sont utilisées pour stocker le document XML et son élément racine respectivement, afin qu'elles puissent être accessibles à d'autres parties du programme si nécessaire.

Classe Produit :

```
package Console;

/**
* La classe Produit représente un produit avec un nom et une description.
*/
public class Produit {
  // Attributs privés de la classe Produit
  private String nom;
  private String description;

  /**
   * Constructeur de la classe Produit.
   *
   * @param nom Le nom du produit.
   * @param description La description du produit.
   */
  public Produit(String nom, String description) {
    this.nom = nom;                 // Initialisation du nom du produit
    this.description = description;    // Initialisation de la description du produit
  }

  /**
   * Méthode pour obtenir le nom du produit.
   *
   * @return Le nom du produit.
   */
  public String getNom() {
```

```
    return this.nom;
  }
  /**
   * Méthode pour définir le nom du produit.
   *
   * @param nom Le nouveau nom du produit.
   */
  public void setNom(String nom) {
    this.nom = nom;
  }

  /**
   * Méthode pour obtenir la description du produit.
   *
   * @return La description du produit.
   */
  public String getDescription() {
    return this.description;
  }

  /**
   * Méthode pour définir la description du produit.
   *
   * @param description La nouvelle description du produit.
   */
  public void setDescription(String description) {
    this.description = description;
  }
  /**
```

```
    * Méthode toString pour obtenir une représentation textuelle du produit.
    *
    * @return Une chaîne de caractères représentant le produit avec son nom et sa description.
    */
   @Override
   public String toString() {
     return "Produit [nom = " + this.nom + ", description = " + this.description + "]";
   }
}
```

Remarques

- La classe `Produit` représente un produit avec deux attributs privés : `nom` et `description`.
- Le constructeur `Produit(String nom, String description)` permet d'initialiser un produit avec un nom et une description.
- Des méthodes getter (`getNom()` et `getDescription()`) et setter (`setNom()` et `setDescription()`) sont fournies pour accéder et modifier les attributs privés `nom` et `description`.
- La méthode `toString()` est redéfinie pour obtenir une représentation textuelle du produit sous forme de chaîne de caractères contenant son nom et sa description.
- Cette classe encapsule les informations relatives à un produit et permet de les manipuler de manière sécurisée à l'aide des méthodes fournies.

Classe Sauvegarde :

```
package Console;


import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;


/**
* La classe Sauvegarde permet d'effectuer la sauvegarde des données du personnel et des emprunts dans un fichier texte.
* Elle utilise les listes d'objets Personnel et Emprunt pour générer un fichier de sauvegarde au format spécifié.
*/
public class Sauvegarde {


   /**
    * Effectue la sauvegarde des données du personnel et des emprunts dans un fichier texte.
    *
    * @param Person La liste des objets Personnel à sauvegarder.
    * @param pret   La liste des objets Emprunt à sauvegarder.
    */
   protected void Sauvegarde(ArrayList<Personnel> Person, ArrayList<Emprunt> pret) {
      // Demande à l'utilisateur de saisir le nom du fichier de sauvegarde avec extension .txt
```

```
System.out.println("Introduire le nom du fichier extension .txt ! ");
String fichier = InputData.inputNomFichier();  // Supposons qu'il s'agit d'une méthode pour récupérer le nom du fichier


try {
  // Création d'un BufferedWriter pour écrire dans le fichier
  BufferedWriter bufWriter = new BufferedWriter(new FileWriter(new File(fichier), true));


  try {
    // Récupération de la date et heure actuelles
    Date date = new Date();
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy h:mm:ss a");
    String formattedDate = sdf.format(date);

    // Écriture de la date de la sauvegarde dans le fichier
    bufWriter.newLine();
    bufWriter.write("DTG de la sauvegarde : " + formattedDate);
    bufWriter.newLine();

    // Écriture d'une ligne de séparation dans le fichier pour les données du personnel
    bufWriter.write("+-------------------------+------------------+--------------------+----------+--------------+-------------------------+");
    bufWriter.newLine();
    bufWriter.write("| Département             | Prénom           | Nom                |  Sexe    | Naissance    |  Email                  |");
    bufWriter.newLine();
    bufWriter.write("+-------------------------+------------------+--------------------+----------+--------------+-------------------------+");
    bufWriter.newLine();
```

```
        // Construction des lignes de données pour chaque personnel dans la liste
        StringBuilder sb = new StringBuilder();
        Iterator<Personnel> iteratorPerson = Person.iterator();

        while (iteratorPerson.hasNext()) {
          Personnel Pers = iteratorPerson.next();
          sb.append(" ");
          sb.append(FixedLenght.setFixedLength(Pers.getDepartement(), 25));  // Supposons que FixedLenght.setFixedLength() est une méthode pour formater la longueur fixe des chaînes
          sb.append(FixedLenght.setFixedLength(Pers.getPrenom(), 20));
          sb.append(FixedLenght.setFixedLength(Pers.getNom(), 20));
          sb.append(FixedLenght.setFixedLength(Pers.getSexe(), 10));
          sb.append(FixedLenght.setFixedLength(Pers.getDateddMMyyyy(), 15));
          sb.append(FixedLenght.setFixedLength(Pers.getEmail(), 25));
          sb.append("\n");
        }

        // Écriture des données du personnel dans le fichier
        bufWriter.write(sb.toString());
        bufWriter.newLine();
        bufWriter.newLine();

        // Écriture d'une ligne de séparation pour les données d'emprunt
        bufWriter.write("+------+-----------------------+-------------------------------------+");
        bufWriter.newLine();
        bufWriter.write("| N°   | Nom - Prénom          | Matèriel                            |");
        bufWriter.newLine();
```

```
        bufWriter.write("+------+----------------------+-------------------------------------
+");
        bufWriter.newLine();

        // Construction des lignes de données pour chaque emprunt dans la liste
        StringBuilder sb1 = new StringBuilder();
        int i = 1;
        Iterator<Emprunt> iteratorEmprunt = pret.iterator();

        while (iteratorEmprunt.hasNext()) {
          Emprunt emprunt = iteratorEmprunt.next();
          sb1.append(" ");
          sb1.append(FixedLenght.setFixedLength(String.valueOf(i), 8));
          sb1.append(FixedLenght.setFixedLength(emprunt.getEmprunteur().getNom() +
" " + emprunt.getEmprunteur().getPrenom(), 26));
          sb1.append(FixedLenght.setFixedLength(emprunt.getArticle(), 30));
          sb1.append("\n");
          i++;
        }

        // Écriture des données d'emprunt dans le fichier
        bufWriter.write(sb1.toString());
      } catch (Throwable var14) {
        // Fermeture du BufferedWriter en cas d'exception
        try {
          bufWriter.close();
        } catch (Throwable var13) {
          var14.addSuppressed(var13);
        }
```

```
            throw var14;
        }

        // Fermeture du BufferedWriter après avoir écrit toutes les données avec succès
        bufWriter.close();
    } catch (IOException var15) {
        // Gestion des erreurs d'entrée/sortie en cas de problème lors de l'écriture dans le fichier
        System.err.println("Une erreur est survenue : " + var15.getMessage());
    }
  }
}
```

Remarques

- La classe `Sauvegarde` permet de sauvegarder les données des objets `Personnel` et `Emprunt` dans un fichier texte.
- La méthode `Sauvegarde(ArrayList<Personnel> Person, ArrayList<Emprunt> pret)` est responsable de cette tâche.
- Elle utilise un `BufferedWriter` pour écrire dans le fichier spécifié par l'utilisateur.
- Les données sont écrites avec une mise en forme spécifique pour les différentes sections (personnel et emprunts).
- Les exceptions d'entrée/sortie (`IOException`) sont gérées et affichent un message d'erreur en cas de problème lors de l'écriture dans le fichier.
- La classe suppose l'existence de certaines méthodes telles que `InputData.inputNomFichier()` pour obtenir le nom du fichier de sauvegarde et `FixedLenght.setFixedLength()` pour formater les chaînes avec une longueur fixe.

Classe Sexe :

```
package Console;

/**
* L'énumération Sexe représente les genres homme et femme avec leurs étiquettes respectives.
*/
public enum Sexe {
  // Définition des constantes enum avec leur étiquette respective
  HOMME("Garçon"),
  FEMME("Fille");

  // Attribut privé pour stocker l'étiquette associée à chaque constante enum
  private final String label;


  /**
   * Constructeur privé de l'énumération Sexe.
   *
   * @param label L'étiquette associée à chaque constante enum (homme ou femme).
   */
  private Sexe(String label) {
    this.label = label;  // Initialisation de l'étiquette lors de la création de chaque constante
  }

  /**
   * Méthode pour obtenir l'étiquette associée à la constante enum.
   *
```

```
    * @return L'étiquette associée à la constante enum.
    */
   public String getLabel() {
      return this.label;
   }

   // Méthode synthétique pour obtenir un tableau contenant toutes les constantes enum
   // Cette méthode est générée par le compilateur et est utilisée en interne
   private static Sexe[] $values() {
      return new Sexe[]{HOMME, FEMME};  // Retourne un tableau contenant toutes les constantes enum de Sexe
   }
}
```

Remarques

- L'énumération `Sexe` définit deux constantes enum : `HOMME` et `FEMME`, avec leurs étiquettes respectives ("Garçon" et "Fille").
- Chaque constante enum est associée à une étiquette grâce à un attribut privé `label`.
- Le constructeur privé `Sexe(String label)` est utilisé pour initialiser l'étiquette lors de la création de chaque constante enum.
- La méthode `getLabel()` permet d'obtenir l'étiquette associée à une constante enum.
- Une méthode synthétique `$values()` est générée par le compilateur pour obtenir un tableau contenant toutes les constantes enum de `Sexe`.
- Cette énumération permet de représenter de manière structurée et typée les genres homme et femme, en associant à chaque genre une étiquette descriptive.

Classe WordUtils :

```
package Console;


/**
* La classe WordUtils fournit des méthodes utilitaires pour manipuler les chaînes de caractères,
* notamment pour la capitalisation et la mise en minuscule.
*/
public class WordUtils {

    /**
     * Capitalise la première lettre et met le reste de la chaîne en minuscules.
     *
     * @param value la chaîne à capitaliser
     * @return la chaîne avec la première lettre en majuscule et le reste en minuscules
     */
    private static String capitalize(String value) {
        String firstLetter = value.substring(0, 1).toUpperCase();  // Capitalise la première lettre
        String restOfString = value.substring(1).toLowerCase();    // Met le reste en minuscules
        return firstLetter + restOfString;  // Retourne la chaîne capitalisée
    }

    /**
     * Met toute la chaîne en minuscules.
     *
     * @param value la chaîne à mettre en minuscules
     * @return la chaîne en minuscules
```

```
     */
    private static String minimize(String value) {
        return value.toLowerCase();  // Retourne la chaîne en minuscules
    }

    /**
     * Met en majuscule les parties d'une chaîne en fonction des délimiteurs spécifiques (apostrophe, tiret, espace).
     *
     * @param inputString la chaîne à traiter
     * @return la chaîne avec chaque partie capitalisée selon les délimiteurs spécifiques
     */
    protected static String mettreEnMajscule(String inputString) {
        String outString;
        String[] partString;

        // Vérifie s'il y a un apostrophe (')
        if (inputString.indexOf("'") > 0) {
            outString = "";
            partString = inputString.split("'");

            for (String part : partString) {
                outString = outString + "'" + capitalize(part);  // Capitalise chaque partie séparée par un apostrophe
            }
            return outString.substring(1);  // Retourne la chaîne sans le premier apostrophe ajouté en trop
        }
        // Vérifie s'il y a un tiret (-)
        else if (inputString.indexOf("-") > 0) {
```

```
            outString = "";
            partString = inputString.split("-");

            for (String part : partString) {
                outString = outString + "-" + capitalize(part);  // Capitalise chaque partie séparée par un tiret
            }
            return outString.substring(1);  // Retourne la chaîne sans le premier tiret ajouté en trop
        }
        // Vérifie s'il y a un espace ( )
        else if (inputString.indexOf(" ") > 0) {
            outString = "";
            partString = inputString.split(" ");

            for (String part : partString) {
                outString = outString + " " + capitalize(part);  // Capitalise chaque partie séparée par un espace
            }
            return outString.substring(1);  // Retourne la chaîne sans le premier espace ajouté en trop
        }
        // Si aucun des délimiteurs spécifiques n'est trouvé, capitalise simplement la chaîne entière
        else {
            return capitalize(inputString);  // Capitalise toute la chaîne
        }
    }

    /**
     * Met en minuscule la partie avant le '@' d'une adresse email.
```

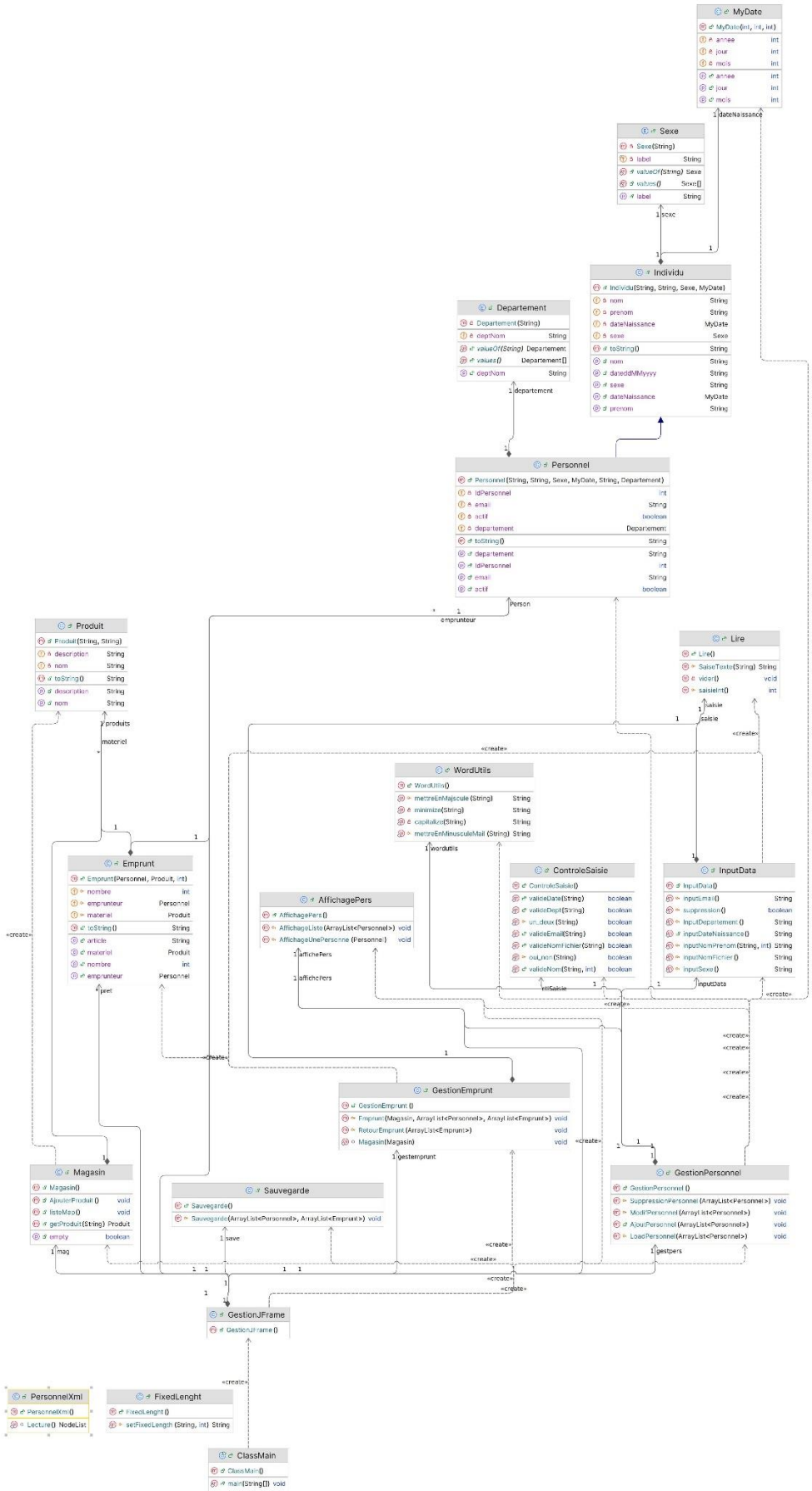
```
     *
     * @param inputString l'adresse email à traiter
     * @return l'adresse email avec la partie avant le '@' en minuscules
     */
    protected static String mettreEnMinusculeMail(String inputString) {
        String outString = "";
        String[] partString = inputString.split("@");  // Sépare la chaîne à partir du caractère '@'


        for (String part : partString) {
            outString = outString + "@" + minimize(part);  // Met en minuscule chaque partie séparée par '@'
        }
        return outString.substring(1);  // Retourne la chaîne sans le premier '@' ajouté en trop
    }
}
```

Remarques

- Les méthodes `capitalize` et `minimize` sont privées, elles ne sont donc accessibles que dans la classe `WordUtils`.
- Les méthodes `mettreEnMajscule` et `mettreEnMinusculeMail` sont protégées, elles sont accessibles dans le même package ou par des sous-classes.
- La méthode `mettreEnMajscule` capitalise les sous-chaînes séparées par des apostrophes, des tirets ou des espaces.
- La méthode `mettreEnMinusculeMail` met en minuscules les parties d'une adresse email séparées par le caractère '@'.

Diagramme :


MyDate
Sexe
Individu
Departement
Personnel
Produit
Lire
WordUtils
Emprunt
ControleSaisie
InputData
AffichagePers
GestionEmprunt
Magasin
Sauvegarde
GestionPersonnel
GestionJFrame
PersonnelXml
FixedLenght
ClassMain