Programmation Objet Avancé

- Le Zoo
- Lanterna
- Pattern
- Annotation et Pattern
- JDBC et Cryptographie
- Batch
- Etudiant
- Salaire
- Ville
- Media
- Aeroport
- Aeroport2

Le Zoo

Correction sur Correction

Nous vous proposons de créer un petit zoo, puis de le gérer. Le zoo est constitué de plusieurs enclos et chaque enclos peut contenir plusieurs animaux. Dans un enclos tous les animaux doivent être du même type (ex : les baleines avec les baleines, les aigles avec les aigles, ...). Cependant, il doit être possible de mettre n'importe quelle espèce dans n'importe quel enclos.
Le zoo contient aussi un employé, qui sert d'interface entre le zoo et vous (l'utilisateur du programme). Au travers de cet employé vous pouvez donner à manger aux animaux, les transférer d'un enclos à un autre, nettoyer les enclos...
Ci-dessous nous vous donnons les spécications minimales du programme.
Elles sont parfois incomplètes. A vous de créer des classes, des variables d'instance, des méthodes ou des interfaces supplémentaires dès que cela semble nécessaire.

Les animaux

Au minimum, le programme doit pouvoir créer des Loups, des Tigres, des Ours, des Baleines, des Poissons, des Requins, des Aigles et des Pingouins.
Pour chaque espèce, nous devons pouvoir indiquer leur poids, taille, nom de l'espece et age. De plus des booléens seront utilisés pour déterminer si l'animal a faim, dort ou est malade.
Au niveau des méthodes, en plus des accesseurs et mutateurs habituels, chaque animal doit pouvoir manger, émettre un son, être soigné, dormir ou se réveiller. Il faut aussi une méthode pour faire une clef de hachage de toutes ses caractéristiques.
Les Tigres et les Loups doivent pouvoir vagabonder. Les animaux marins doivent pouvoir nager et les animaux volant doivent pouvoir voler.
Les mamifères doivent pouvoir mettre bas, alors que les autres animaux pondent des oeufs. La naissance du nouvel animal dépend de la durée de gestation ou d'incubation de l'espèce.

Question En utilisant une hiérarchie d'héritage et des interfaces, écrivez des classes pour chacun de ces animaux. Toutes les classes écrites doivent être testées.

Les enclos

Chaque enclos peut contenir plusieurs animaux (i.e. un tableau d'animaux).
Dans un enclos tous les animaux doivent être du même type (ex : les baleines avec les baleines, les aigles avec les aigles, ...). Cependant, il doit être possible de mettre n'importe quelle espèce dans n'importe quel enclos (sauf pour les volières

et les aquariums).
Tous les enclos possèdent les caractéristiques suivantes : un nom, une supercie, un degrès de propreté (pouvant prendre comme valeur : mauvaise, correcte, bonne), le nombre d'animaux présents et le nombre maximum d'animaux qu'il
peut contenir.
Outres les accesseurs et mutateurs, les méthodes d'un enclos doivent permettre : d'acher ses caractéristiques, d'acher les caractéristiques des animaux qu'il contient, d'ajouter et d'enlever des animaux dans l'enclos, d'entretenir l'enclos si celui-ci est vide.
On distinguera deux sous-classes d'enclos particulier : les volières et les aquariums. Une volière ne peut contenir que des animaux volants. En plus des caractéristiques communes à tous les enclos, elle possède aussi une hauteur. L'entretien de ce type d'enclos nécessite aussi la vérication du sommet de la cage.
Similairement, un aquarium ne peut contenir que des animaux aquatiques.

Un aquarium possède deux variables supplémentaires, la profondeur du bassin et la salinité de l'eau. L'entretien d'un aquarium nécessite la vérication de la salinité de l'eau et le nettoyage du bassin.

Question Ecrivez les classes correspondant aux trois types d'enclos. Toutes
les classes écrites doivent être testées.

L'employé

L'employé possède les caractéristiques suivantes : le nom de l'employé, son age, son sexe. Outre les constructeurs, accesseurs et mutateurs nécessaires, les méthodes suivantes, entre autres, doivent permettre :

1. à l'employer d'examiner un enclos, Cette méthode ache les animaux
contenus dans l'enclos, ainsi que les caractéristiques de l'enclos,
2. de nettoyer un enclos si l'enclos est sale et vide,
3. de nourrir les animaux d'un enclos lorsqu'ils ne dorment pas,
4. d'ajouter à un enclos un nouvel animal lorsque c'est possible,
5. de lever un animal d'un enclos,
6. de transférer un animal d'un enclos à un autre.

En l'employé possède une dernière méthode qui constitue l'interface avec l'utilisateur. A l'aide d'un menu, l'utilisateur doit pouvoir diriger l'employé.

Question Ecrivez la classe correspondant à l'employé. N'oubliez pas de la
tester.

Le zoo

Il reste maintenant à concevoir le zoo. Un zoo possède un nom, un employé, un nombre maximal d'enclos nbMax et un tableau de nbMax enclos. Les méthodes d'un zoo permettent

1. d'acheter le contenu de tous les enclos,
2. et d'acher le nombre d'animaux présents dans le zoo.

En le zoo contient aussi la méthode main. Le comportement de la méthode main est le suivant. Dans une boucle while :
1. pour chaque animal du zoo, on va aléatoirement modier les valeurs des variables d'instance de cet animal (par exemple on le rend malade, on l'endort ou on l'aame).
2. pour chaque enclos, on modie aléatoirement son état de propreté, sa salinité, etc
3. enn on passe la main à l'employé (donc à vous, utilisateur) pour qu'il s'occupe du zoo.

Question Ecrivez la classe correspondant au zoo. N'oubliez pas de la tester.

EasyBatch

En cas d'utilisation de EasyBatch:

```
 public static void main(String[] args) throws Exception {
        List<AnimalCSV> productList=new ArrayList<AnimalCSV>();
    org.easybatch.core.job.Job job = new JobBuilder()
                .reader(new
FlatFileRecordReader("D:\\MyJavaProjects\\etnic2\\lanterna\\animaux.csv"))
                .mapper(new DelimitedRecordMapper(AnimalCSV.class, "id","type", "poid",
"age"))
                .filter(new MyProductFilter())
                .processor(new ProductProcessor(productList))
                .build();

        JobExecutor jobExecutor = new JobExecutor();
        JobReport report = jobExecutor.execute(job);
        jobExecutor.shutdown();


        System.out.println("job report = " + productList);
    }
```

```
import org.easybatch.core.processor.RecordProcessor;
import org.easybatch.core.record.Record;
```

```
public class ProductProcessor implements RecordProcessor<Record<AnimalCSV>, Record<AnimalCSV>>
{

    private List<AnimalCSV> productList;

    public ProductProcessor(List<AnimalCSV> productList2) {
        this.productList=productList2;
    }

    public Record<AnimalCSV> processRecord(Record<AnimalCSV> record) {
        productList.add(record.getPayload());
        return record;
        }


}
```

```
<dependency>
    <groupId>org.easybatch</groupId>
    <artifactId>easybatch-core</artifactId>
    <version>5.2.0</version>
</dependency>
    <!-- https://mvnrepository.com/artifact/org.easybatch/easybatch-flatfile -->
<dependency>
    <groupId>org.easybatch</groupId>
    <artifactId>easybatch-flatfile</artifactId>
    <version>5.2.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.easybatch/easybatch-xml -->
<dependency>
    <groupId>org.easybatch</groupId>
    <artifactId>easybatch-xml</artifactId>
    <version>5.2.0</version>
</dependency>
```

Lanterna

Lanterna est une librarie permettant de créer des interfaces graphiques en mode texte.

Le but de ce tp est de voir de la POO a travers cette librairie implémentant beaucoup de pattern.

Lanterna est une bibliothèque Java vous permettant d'écrire des interfaces utilisateur semi-graphiques simples dans un environnement texte uniquement, très similaire aux cursus de la bibliothèque C mais avec plus de fonctionnalités. Lanterna prend en charge les terminaux et les émulateurs de terminaux compatibles xterm tels que konsole, gnome-terminal, mastic, xterm et bien d'autres. Un des principaux avantages de Lanterna est qu'elle ne dépend d'aucune bibliothèque native, mais fonctionne à 100% en Java pur.

De plus, lors de l'exécution de Lanterna sur des ordinateurs dotés d'un environnement graphique (tel que Windows ou Xorg), un émulateur de terminal fourni écrit en Swing sera utilisé plutôt que la sortie standard. De cette façon, vous pouvez développer comme d'habitude à partir de votre IDE (la plupart d'entre eux ne prennent pas en charge les caractères de contrôle ANSI dans leur fenêtre de sortie), puis se déployer sur votre serveur sans interface utilisateur sans changer de code.

Installation

Dans le pom.xml rajouter :

```
    <dependency>
        <groupId>com.googlecode.lanterna</groupId>
        <artifactId>lanterna</artifactId>
        <version>3.0.1</version>
    </dependency>
```

Un terminal

Le noyau de la couche Terminal est l'interface de Terminal que vous devez récupérer. Cela peut être fait soit en instanciant directement l'une des classes concrètes implémentées (UnixTerminal, CygwinTerminal ou SwingTerminal) ou en utilisant DefaultTerminalFactory pour créer un terminal:

```
Terminal terminal = new DefaultTerminalFactory().createTerminal();
```

Un ecran

L'ecran est une surface sur laquel on va dessiner:

```
Terminal terminal = new DefaultTerminalFactory().createTerminal();
Screen screen = new TerminalScreen(terminal);
screen.startScreen();
```

Puis nous allons rajouter un panel:

```
 Panel panel = new Panel();
        panel.setLayoutManager(new GridLayout(2));

        final Label lblOutput = new Label("");

        panel.addComponent(new Label("Num 1"));
        final TextBox txtNum1 = new TextBox().setValidationPattern(Pattern.compile("[0-
9]*")).addTo(panel);

        panel.addComponent(new Label("Num 2"));
        final TextBox txtNum2 = new TextBox().setValidationPattern(Pattern.compile("[0-
9]*")).addTo(panel);

        panel.addComponent(new EmptySpace(new TerminalSize(0, 0)));
        new Button("Add!", new Runnable() {
            @Override
            public void run() {
                int num1 = Integer.parseInt(txtNum1.getText());
                int num2 = Integer.parseInt(txtNum2.getText());
                lblOutput.setText(Integer.toString(num1 + num2));
            }
        }).addTo(panel);

        panel.addComponent(new EmptySpace(new TerminalSize(0, 0)));
        panel.addComponent(lblOutput);
```

Et enfin on lit le panel avec l'ecran via un fenetre

```
  BasicWindow window = new BasicWindow();
        window.setComponent(panel);

        // Create gui and start gui
        MultiWindowTextGUI gui = new MultiWindowTextGUI(screen, new DefaultWindowManager(),
new EmptySpace(TextColor.ANSI.BLUE));
```

```
        gui.addWindowAndWait(window);
```

Un code plus interessant est de faire rajouter une liste d'objet à une table

```
package be.etnic.lanterna;

import java.awt.Menu;
import java.awt.MenuBar;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import com.googlecode.lanterna.TextColor;
import com.googlecode.lanterna.gui2.BasicWindow;
import com.googlecode.lanterna.gui2.Borders;
import com.googlecode.lanterna.gui2.Button;
import com.googlecode.lanterna.gui2.DefaultWindowManager;
import com.googlecode.lanterna.gui2.Direction;
import com.googlecode.lanterna.gui2.EmptySpace;
import com.googlecode.lanterna.gui2.GridLayout;
import com.googlecode.lanterna.gui2.Label;
import com.googlecode.lanterna.gui2.LinearLayout;
import com.googlecode.lanterna.gui2.MultiWindowTextGUI;
import com.googlecode.lanterna.gui2.Panel;
import com.googlecode.lanterna.gui2.TextBox;
import com.googlecode.lanterna.gui2.dialogs.FileDialogBuilder;
import com.googlecode.lanterna.gui2.table.Table;
import com.googlecode.lanterna.screen.Screen;
import com.googlecode.lanterna.screen.TerminalScreen;
import com.googlecode.lanterna.terminal.DefaultTerminalFactory;
import com.googlecode.lanterna.terminal.Terminal;

public class MyLanterna extends BasicWindow{

    Table<String> table ;
    private void clearAllRow()
    {
        while (table.getTableModel().getRowCount()>0)
```

```
    }
    table.getTableModel().removeRow(0);
  }
}
public MyLanterna()
{
  super();
  Panel mainPanel = new Panel();
  mainPanel.setLayoutManager(new LinearLayout(Direction.VERTICAL));
   table = new Table<String>("Nom");
  mainPanel.addComponent(table);
  mainPanel.addComponent(getFormPanel().withBorder(Borders.singleLine("Ajouter un
utilisateur")));

  this.setComponent(mainPanel);
  
 
}
public Panel getFormPanel() {
 Panel panel = new Panel();
      panel.setLayoutManager(new GridLayout(2));

      final Label lblOutput = new Label("");

      panel.addComponent(new Label("Nom"));
      final TextBox txtNum1 = new TextBox().addTo(panel);
      new Button("Add!", new Runnable() {
          public void run() {
         List<List<String>> value = table.getTableModel().getRows();
         
         clearAllRow();
         List<String> newV=new ArrayList<String>();
         newV.add(txtNum1.getText());
         
         value.add(newV);
         for (List<String> strs:value)
         {
          table.getTableModel().addRow(strs);
         }
```

```
            }
        }).addTo(panel);
        return panel;
    }
    public static void main(String[] args) throws IOException
    {
        Terminal terminal = new DefaultTerminalFactory().createTerminal();
        Screen screen = new TerminalScreen(terminal);
        screen.startScreen();
        MultiWindowTextGUI gui = new MultiWindowTextGUI(screen, new DefaultWindowManager(),
new EmptySpace(TextColor.ANSI.BLUE));
        gui.addWindowAndWait(new MyLanterna());
    }
}
```

Pattern

Soit une interface Job representant un job, un stage. Les jobs ont a minima un titre et un identifiant

Job

```
public interface Job {

	Long getId();
	String getTitle();
}
```

Les Stages

Les Stages sont des types de Job

```
public class Stage implements Job{

	private String title;
	private Long id;
	
	public Stage()
	{
		
	}
	public String getTitle() {
		return title;
	}

	public Long getId() {
		return id;
	}

	@Override
```

```
    public String toString() {
        return "Stage [title=" + title + ", id=" + id + "]";
    }
```

Pattern de builder

Le pattern de builder permet de construire des objets sans pouvoir les modifier (object immuable):

```
private Stage(Builder builder) {
    this.title = builder.title;
    this.id = builder.id;
}
/**
 * Creates builder to build {@link Stage}.
 * @return created builder
 */
public static Builder builder() {
    return new Builder();
}
/**
 * Builder to build {@link Stage}.
 */
public static final class Builder {
    private String title;
    private Long id;

    private Builder() {
    }

    public Builder withTitle(String title) {
        this.title = title;
        return this;
    }

    public Builder withId(Long id) {
        this.id = id;
        return this;
    }
```

```
    public Stage build() {
        return new Stage(this);
    }
}
```

Pattern de Singleton

La librairie Kryo permet de sauvegarder des grappes d'objet sur un fichier (et de les relire).

La dépendance est la suivante:

```
 <dependency>
   <groupId>com.esotericsoftware</groupId>
   <artifactId>kryo</artifactId>
   <version>5.0.0-RC4</version>
</dependency>
```

Le pattern de singleton est implémenté via un enum:

```
public enum KrioSingleton {

    INSTANCE;
    private Kryo kryo;

    private KrioSingleton()
    {
        this.kryo = new Kryo();
        kryo.register(java.util.HashMap.class);
        kryo.register(JobRepository.class);
        kryo.register(Stage.class);

    }
    public JobRepository read(String filename) throws FileNotFoundException
    {
        Input input = new Input(new FileInputStream(filename));
        JobRepository jobRepository = kryo.readObject(input, JobRepository.class);
        input.close();
        return jobRepository;
    }
```

```
public void save(String filename,JobRepository repository) throws FileNotFoundException
{
    Output output = new Output(new FileOutputStream(filename));
        kryo.writeObject(output, repository);
        output.flush();
        output.close();
}

}
```

Pattern de DAO.

Un pattern de DAO est capable de lire et d'ecrire un objet depuis une base de donnée, fichier ... Ici on fait une implémentation avec Kryo:

```
public class JobRepository {


Map<Long,Job> data=new HashMap<Long,Job>();
static String defaultFileName="jobstore.bin";
public JobRepository()
{

}
public static JobRepository getJobRepository()
{
    try {
        return KrioSingleton.INSTANCE.read(defaultFileName);
    } catch (FileNotFoundException e) {
        return new JobRepository();
    }
}
public Long nextId()
{
    return new Long(data.size()+1);

}
public Job readJob(Long id)
{
    return data.get(id);
}
```

```
    public void saveJob(Job job)
    {
        this.data.put(job.getId(), job);
        try {
            KrioSingleton.INSTANCE.save(defaultFileName, this);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    @Override
    public String toString() {
        return "JobRepository [data=" + data + "]";
    }


}
```

Annotation et Pattern

Pattern d'injection

Nous allons regarder le pattern d'injection.

Soit une interface Store permettant de sauvegarder des données dans un fichier

```
public interface Store {

	public void initStore(String str);
	public void storeData(String filename, byte [] data);
	
}
```

Premiere implémentation.

La première implémentation utilise la librairie Apache common pour sauvegarder des données dans un fichier

```
import org.apache.commons.io.IOUtils;

public class PlainStore implements Store {

	public void initStore(String str) {
		// TODO Auto-generated method stub
		
	}

	public void storeData(String filename, byte[] data) {
		try{
			FileOutputStream stream=new FileOutputStream(filename);
```

```
        IOUtils.write(data, stream);
        stream.close();
    }
    catch (Exception e)
    {
        throw new RuntimeException(e);
    }
}

}
```

Deuxième Implementation

La deuxième implémentation utilise de la cryptographie pour enregistrer les données

```
package be.etnic.guice;

import java.io.FileOutputStream;
import java.security.SecureRandom;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.SecretKeySpec;

import org.apache.commons.io.IOUtils;

public class AESStoreProcessor implements Store{

    private SecretKey key;

    public void initStore(String str) {
        try{
            SecureRandom sr = SecureRandom.getInstanceStrong();
            byte[] salt = new byte[16];
            sr.nextBytes(salt);
```

```
    PBEKeySpec spec = new PBEKeySpec(str.toCharArray(), salt, 65536, 128);
    SecretKey tmp = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1")
        .generateSecret(spec);
    this.key = new SecretKeySpec(tmp.getEncoded(), "AES");

  }
  catch (Exception e)
  {
    throw new RuntimeException(e);
  }
}

public void storeData(String filename, byte[] data) {
   try{
    Cipher aes = Cipher.getInstance("AES");
    aes.init(Cipher.ENCRYPT_MODE, key);
   FileOutputStream stream=new FileOutputStream(filename);

  IOUtils.write(aes.doFinal(data)
    , stream);
  stream.close();
  }
  catch (Exception e)
  {
    throw new RuntimeException(e);
  }
}

}
```

Module de haut niveau.

Nous mettons a disposition une classe permettant d'avoir des store

```
public class StoreProcessor {
```

```
Store store;

public StoreProcessor(Store store)
{
  this.store=store;
}
public void storeData(String filename,String initParameter,byte [] data)
{
  store.initStore(initParameter);
  store.storeData(filename, data);
}
}
```

Le hic est qu'il faut passer un store en parametre du store processor.

On peut utiliser ici un pattern d'injection

```
public class StoreModule extends AbstractModule {
  @Override
  protected void configure() {

    /*
     * This tells Guice that whenever it sees a dependency on a TransactionLog,
     * it should satisfy the dependency using a DatabaseTransactionLog.
     */
   bind(Store.class).to(PlainStore.class);
  }

}
```

Et placer l'injection sur le constructeur de StoreProcessor

```
public class StoreProcessor {
Store store;

@Inject
public StoreProcessor(Store store)
{
  this.store=store;
}
```

```
    public void storeData(String filename,String initParameter,byte [] data)
    {
        store.initStore(initParameter);
        store.storeData(filename, data);
    }
}
```

Dans la fonction main est ainsi faites

```
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
        StoreProcessor storeProcessor=new StoreProcessor(new AESStoreProcessor());
        storeProcessor.storeData("test.txt", "pilou", "hello world".getBytes());
        Injector injector = Guice.createInjector(new StoreModule());
        GuiceStoreProcessor storeProcessor2=injector.getInstance(GuiceStoreProcessor.class);
        storeProcessor2.storeData("test2.txt", "pilou", "hello world".getBytes());

    }
}
```

JDBC et Cryptographie

Le but de cette exercice est de stocké des donnée dans une base de donnée. Pour cela, on va faire une mini interface permettant de stocker des persoones dans une base SQLLite

```
<!-- https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc -->
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.28.0</version>
</dependency>
```

Le code suivant permet de créer une base de donnée d'ajouter des personnes et de les lire:

```
public class JDBC
{
  public static void main(String[] args) throws ClassNotFoundException
  {
    // load the sqlite-JDBC driver using the current class loader
    Class.forName("org.sqlite.JDBC");

    Connection connection = null;
    try
    {
      // create a database connection
      connection = DriverManager.getConnection("jdbc:sqlite:sample.db");
      Statement statement = connection.createStatement();
      statement.setQueryTimeout(30);  // set timeout to 30 sec.

      statement.executeUpdate("drop table if exists person");
      statement.executeUpdate("create table person (id integer, name string)");
      statement.executeUpdate("insert into person values(1, 'leo')");
      statement.executeUpdate("insert into person values(2, 'yui')");
      ResultSet rs = statement.executeQuery("select * from person");
      while(rs.next())
```

```
      {
        // read the result set
        System.out.println("name = " + rs.getString("name"));
        System.out.println("id = " + rs.getInt("id"));
      }
    }
    catch(SQLException e)
    {
      // if the error message is "out of memory",
      // it probably means no database file is found
      System.err.println(e.getMessage());
    }
    finally
    {
      try
      {
        if(connection != null)
          connection.close();
      }
      catch(SQLException e)
      {
        // connection close failed.
        System.err.println(e);
      }
    }
  }
}
```

Le code suivant permet de faire un peu de cryptographie:

```
import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Arrays;
import java.util.Base64;

import javax.crypto.Cipher;
```

```
import javax.crypto.spec.SecretKeySpec;

public class AES {

    private static SecretKeySpec secretKey;
    private static byte[] key;

    public static void setKey(String myKey)
    {
        MessageDigest sha = null;
        try {
            key = myKey.getBytes("UTF-8");
            sha = MessageDigest.getInstance("SHA-1");
            key = sha.digest(key);
            key = Arrays.copyOf(key, 16);
            secretKey = new SecretKeySpec(key, "AES");
        }
        catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
    }

    public static String encrypt(String strToEncrypt, String secret)
    {
        try
        {
            setKey(secret);
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            return
Base64.getEncoder().encodeToString(cipher.doFinal(strToEncrypt.getBytes("UTF-8")));
        }
        catch (Exception e)
        {
            System.out.println("Error while encrypting: " + e.toString());
        }
        return null;
```

```
    }

    public static String decrypt(String strToDecrypt, String secret)
    {
        try
        {
            setKey(secret);
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
            cipher.init(Cipher.DECRYPT_MODE, secretKey);
            return new String(cipher.doFinal(Base64.getDecoder().decode(strToDecrypt)));
        }
        catch (Exception e)
        {
            System.out.println("Error while decrypting: " + e.toString());
        }
        return null;
    }
    public static void main(String[] args)
    {
        final String secretKey = "Pilou";

        String originalString = "Hello World";
        String encryptedString = AES.encrypt(originalString, secretKey) ;
        String decryptedString = AES.decrypt(encryptedString, secretKey) ;

        System.out.println(originalString);
        System.out.println(encryptedString);
        System.out.println(decryptedString);
    }
}
```

L'idée est d'utiliser lanterna afin de faire un petit formulaire qui sauvegarderas en base des personnes de facon crypté.

Batch

Un batch peut se faire ne J2EE mais aussi peut se faire plus simplement:

Soit la classe produit:

```
import java.util.Date;


public class Product {
    private long id;
    private String name;
    private String description;
    private double price;
    private boolean published;
    private Date lastUpdate;
    // getters, setters and toString() omitted
}
```

Nous allons mettre en place le batch suivant

- Lire les données du fichier plat products.csv
- Filtrer les enregistrements commençant par #
- Mappe chaque enregistrement CSV à une instance du produit POJO
- Valider les données du produit
- Traiter chaque enregistrement à l'aide de l'implémentation ProductProcessor

Le fichier CSV a filtrer est le suivant:

```
#id,name,description,price,published,lastUpdate
0001,product1,description1,2500,true,2014-01-01
000x,product2,description2,2400,true,2014-01-01
0003,,description3,2300,true,2014-01-01
0004,product4,description4,-2200,true,2014-01-01
0005,product5,description5,2100,true,2024-01-01
0006,product6,description6,2000,true,2014-01-01
```

Le code suivant permet de filtrer les données du fichier CSV et de les mettre dans un autre fichier:

```
import org.easybatch.core.filter.StartWithStringRecordFilter;
import org.easybatch.core.job.Job;
import org.easybatch.core.job.JobBuilder;
import org.easybatch.flatfile.DelimitedRecordMapper;
import org.easybatch.flatfile.FlatFileRecordReader;
import org.easybatch.validation.BeanValidationRecordValidator;

public class WithEasyBatch {

    public static void main(String[] args) throws Exception {
        org.easybatch.core.job.Job job = new JobBuilder()
                .reader(new
FlatFileRecordReader("D:\\MyJavaProjects\\etnic2\\lanterna\\test.csv"))
                .mapper(new DelimitedRecordMapper(Product.class, "id","name", "description",
"price","published", "lastUpdate"))
                .filter(new MyProductFilter())
                .processor(new ProductProcessor())
                .writer(new StandardOutputRecordWriter())
                .build();

        JobExecutor jobExecutor = new JobExecutor();
        JobReport report = jobExecutor.execute(job);
        jobExecutor.shutdown();

        System.out.println("job report = " + report);
    }

}
```

Avec comme processur métier:

```
public class ProductProcessor implements RecordProcessor<Record, Record> {

    public Record processRecord(final Record record) throws Exception {
        System.out.println("product = " + record.getPayload());
        return record;
    }
```

```
}
```

Et comme filtre

```
public class MyProductFilter implements RecordFilter<Record<Product>> {

	public Record<Product> processRecord(Record<Product> record) {
		
		// TODO Auto-generated method stub
		return record.getPayload().getPrice()>0?record:null;
	}

}
```

Etudiant

Vous avez créer la classe Etudiant

```
package etudiant;


public class Etudiant {


}
```

Vous allez compléter le code de cette classe en lui ajoutant :

- un attribut privé de type String nommé nom ;
- un constructeur publique qui a un paramètre de type String servant à initiliser le nom de l'étudiant ;
- une méthode publique sans paramètre et qui ne renvoie rien, nommée travailler, qui écrit à l'écran, si le nom de l'étudiant a pour nom toto : toto se met au travail !
- une méthode publique sans paramètre et qui ne renvoie rien, nommée seReposer, qui écrit à l'écran, si le nom de l'étudiant a pour nom toto : toto se repose

```
public class Etudiant {
    public String nom;
    Etudiant(String nom) {
        this.nom = nom;
    }

    public String getNom() {
        return this.nom;
    }

    public void travailler() {
        System.out.println(this.nom + " se met au travail !");
    }

    public void seReposer() {
        System.out.println(this.nom + " se repose");
    }
```

```
}
```

Salaire

Le directeur d'une entreprise de produits chimiques souhaite gérer les salaires et primes de ses employés au moyen d'un programme Java. Un employé est caractérisé par son nom, son prénom, son âge et sa date d'entrée en service dans l'entreprise.

Codez une classe abstraite Employe dotée des attributs nécessaires, d'une méthode abstraite calculerSalaire (ce calcul dépendra en effet du type de l'employé) et d'une méthode getNom retournant une chaine de caractère obtenue en concaténant la chaine de caractères "L'employé " avec le prénom et le nom.

Dotez également votre classe d'un constructeur prenant en paramètre l'ensemble des attributs nécessaires.

Calcul du salaire Le calcul du salaire mensuel dépend du type de l'employé. On distingue les types d'employés suivants :

- Ceux affectés à la Vente. Leur salaire mensuel est le 20 % du chiffre d'affaire qu'ils réalisent mensuellement, plus 400 €.
- Ceux affectés à la Représentation. Leur salaire mensuel est également le 20 % du chiffre d'affaire qu'ils réalisent mensuellement, plus 800 €.
- Ceux affectés à la Production. Leur salaire vaut le nombre d'unités produites mensuellement multipliées par 5.
- Ceux affectés à la Manutention. Leur salaire vaut leur nombre d'heures de travail mensuel multipliées par 65 €.

Faire une classe Entreprise permettant:

- void ajouterEmploye(Employe) qui ajoute un employé à la collection.
- void calculerSalaires() qui affiche le salaire de chacun des employés de la collection.
- double salaireMoyen() qui affiche le salaire moyen des employés de la collection.

Correction

Une parmis tant d'autre

```
/* ********************************************************************
 *  La classe Employe
 * ********************************************************************/
abstract class Employe {
```

```
    private String nom;
    private String prenom;
    private int age;

    private String date;


    public Employe(String prenom, String nom, int age, String date) {

        this.nom = nom;
        this.prenom = prenom;

        this.age = age;
        this.date = date;

    }

    public abstract double calculerSalaire();


    public String getTitre()
        {
            return "L'employé " ;

        }

    public String getNom() {
        return getTitre() + prenom + " " + nom;

    }
}

/* *********************************************************************
 *  La classe Commercial (regroupe Vendeur et Représentant)
 * *********************************************************************/
abstract class Commercial extends Employe {

    private double chiffreAffaire;
```

```
    public Commercial(String prenom, String nom, int age, String date,

            double chiffreAffaire) {
        super(prenom, nom, age, date);
        this.chiffreAffaire = chiffreAffaire;

    }

    public double getChiffreAffaire()
        {
            return chiffreAffaire;

        }

}

/* *********************************************************************
 *  La classe Vendeur
 * *********************************************************************/
class Vendeur extends Commercial {

    private final static double POURCENT_VENDEUR = 0.2;
    private final static int BONUS_VENDEUR = 400;


    public Vendeur(String prenom, String nom, int age, String date,

            double chiffreAffaire) {
        super(prenom, nom, age, date, chiffreAffaire);
    }


    public double calculerSalaire() {
        return (POURCENT_VENDEUR * getChiffreAffaire()) + BONUS_VENDEUR;

    }

    public String getTitre()
        {
            return "Le vendeur ";
```

```
        }

}

/* ********************************************************************
 *  La classe Représentant
 * ********************************************************************/
class Representant extends Commercial {

    private final static double POURCENT_REPRESENTANT = 0.2;
    private final static int BONUS_REPRESENTANT = 800;



    public Representant(String prenom, String nom, int age, String date, double
chiffreAffaire) {

        super(prenom, nom, age, date, chiffreAffaire);
    }

    public double calculerSalaire() {

        return (POURCENT_REPRESENTANT * getChiffreAffaire()) + BONUS_REPRESENTANT;

    }

    public String getTitre()
        {
            return "Le représentant ";

        }
}

/* ********************************************************************
 *  La classe Technicien (Production)
 * ********************************************************************/
class Technicien extends Employe {

    private final static double FACTEUR_UNITE = 5.0;
    private int unites;
```

```
    public Technicien(String prenom, String nom, int age, String date, int unites) {

        super(prenom, nom, age, date);
        this.unites = unites;

    }

    public double calculerSalaire() {
        return FACTEUR_UNITE * unites;

    }

    public String getTitre()
        {
            return "Le technicien ";

        }
}

/* *********************************************************************
 *  La classe Manutentionnaire
 * *********************************************************************/
class Manutentionnaire extends Employe {

    private final static double SALAIRE_HORAIRE = 65.0;
    private int heures;


    public Manutentionnaire(String prenom, String nom, int age, String date,

                    int heures) {
        super(prenom, nom, age, date);
        this.heures = heures;

    }

    public double calculerSalaire() {
        return SALAIRE_HORAIRE * heures;
```

```
    }

    public String getTitre()
        {
            return "Le manut. " ;

        }
}

/* ********************************************************************
 *  L'interface d'employés à risque
 * ********************************************************************/
interface ARisque {

    int PRIME = 200;
}

/* ********************************************************************
 *  Une première sous-classe d'employé à risque
 * ********************************************************************/

class TechnARisque extends Technicien implements ARisque {

    public TechnARisque(String prenom, String nom, int age, String date, int unites) {

        super(prenom, nom, age, date, unites);
    }

    public double calculerSalaire() {

        return super.calculerSalaire() + PRIME;
    }

}

/* ********************************************************************
 *  Une autre sous-classe d'employé à risque
 * ********************************************************************/
class ManutARisque extends Manutentionnaire implements ARisque {
```

```
    public ManutARisque(String prenom, String nom, int age, String date, int heures) {

        super(prenom, nom, age, date, heures);
    }

    public double calculerSalaire() {

        return super.calculerSalaire() + PRIME;
    }

}
/* *********************************************************************
 *  La classe Personnel
 * *********************************************************************/
class Personnel {
    private Employe[] staff;

    private int nbreEmploye;
    private final static int MAXEMPLOYE = 200;


    public Personnel() {
        staff = new Employe[MAXEMPLOYE];

        nbreEmploye = 0;
    }

    public void ajouterEmploye(Employe e) {

        ++nbreEmploye;
        if (nbreEmploye <= MAXEMPLOYE) {

            staff[nbreEmploye - 1] = e;
        } else {

            System.out.println("Pas plus de " + MAXEMPLOYE + " employés");

        }
```

```
    }

    public double salaireMoyen() {

        double somme = 0.0;
        for (int i = 0; i < nbreEmploye; i++) {

            somme += staff[i].calculerSalaire();
        }

        return somme / nbreEmploye;
    }

    public void afficherSalaires() {

        for (int i = 0; i < nbreEmploye; i++) {

            System.out.println(staff[i].getNom() + " gagne "

                    + staff[i].calculerSalaire() + " francs.");

        }
    }
}


class Salaires {

    public static void main(String[] args) {

        Personnel p = new Personnel();
        p.ajouterEmploye(new Vendeur("Pierre", "Business", 45, "1995", 30000));

        p.ajouterEmploye(new Representant("Léon", "Vendtout", 25, "2001", 20000));

        p.ajouterEmploye(new Technicien("Yves", "Bosseur", 28, "1998", 1000));

        p.ajouterEmploye(new Manutentionnaire("Jeanne", "Stocketout", 32, "1998", 45));

        p.ajouterEmploye(new TechnARisque("Jean", "Flippe", 28, "2000", 1000));
```

```
        p.ajouterEmploye(new ManutARisque("Al", "Abordage", 30, "2001", 45));


        p.afficherSalaires();
        System.out.println("Le salaire moyen dans l'entreprise est de "

                + p.salaireMoyen() + " francs.");

    }


}
```

Ville

1 Créer une classe Ville correspondant a un nom de ville et un nombre d'habitant.
Ces villes doivent pouvoir donner accès a ces informations (nom, nb d'habitant).
Ces villes ont un constructeur permettant de créer une ville avec un nom de ville.
2 Créer une classe Test qui utilise la classe Ville et utilisera les ArrayList. La fonction main comprendra les lignes suivantes :
public static void main(String [] args){
ArrayList<Ville> mesVilles = new ArrayList<Ville>();
mesVilles.add(new Ville("Lille"));
mesVilles.add(new Ville("Calais"));

3. Rentrez 5 villes différentes. Créez une boucle for pour appeler la méthode toString() de toutes les villes

4. Surchargez le constructeur de la classe Ville. Définissez un constructeur, à deux arguments le nom de la
ville et le nombre d'habitant
Créez une classe Capitale qui hérite de la classe Ville. Celle-ci comprendra une variable d'instance supplémentaire : nomPays. .
Testez différents cas : appel explicite ou non au constructeur de la classe mère ; existence ou non d'un constructeur sans arguments.

6. Redéfinissez la méthode toString(), en faisant appel à la méthode de la classe mère. (toString()) qui affichera à l'écran Capitale de nomPays: nomVille ; nbHabitants). Testez.
7. Changez les modificateurs d'accès des données membres de la classe mère, en remplaçant private par protected. Peut-on accéder à ces variables depuis l'extérieur de la classe Ville ?

```
public class Ville {

 // les attributs de la ville
 private int nbHabitants;
 private String nomVille;

 // constructeurs de la classe Ville
 public Ville( String v){
 nomVille = v;
 nbHabitants = 0;
```

```
 }
 public Ville(String v, int k){
 nomVille = v;
 nbHabitants = k;
 }
 // les méthodes de la classe Ville
 public String getNomVille(){
 return nomVille;
 }
 public int getNbHabitants(){
 return nbHabitants;
 }
 public void setNbHabitants(int k){
 nbHabitants = k;
 }
 public String toString(){
 String resultat;
 resultat = "ville : " + nomVille + "\thabitants : " + nbHabitants;
 return resultat;
 }
}
```

Capitale:

```
public class Capitale extends Ville {
 // la classe Capitale hérite de la calsse Ville
 private String nomPays; // attribut supplémentaire de la classe Capitale
 // constructeur de la classe Capitale à 3 paramètre, pays, ville et nombre d'habitants
 public Capitale(String np, String nv, int nh){
 /*--- ne fonctionne pas car nomVille et nbHabitants sont privés
 nomVille = nv;
 nbHabitants = nh;
 */
 super(nv, nh); // appel au constructeur de la classe mère
 nomPays = np;
 }
 // les méthodes de Capitale
 public String toString(){
 String resultat;
 resultat = "Capitale de " + nomPays + " " + super.toString();
```

```
 return resultat;
 }
}
```

Media

Soit un media, un livre et un DVD. Un livre et DVD sont des cas particuliers de media.

- la classe Media définit un attribut ’titre’ et une méthode ’toString()’
- la classe Livre hérite de la classe Media (c’est un média particulier) et définit l’attribut nombre de page et redéfinit la méthode toString()
- la classe DVD hérite de la classe Media (c’est un média particulier) et définit l’attribut ’durée’ et
  redéfinit la méthode ’toString()’

Que doit donner le code suivant:

```
public class MediaTest {
 public static void main(String [] args){
 Media o1 = new Media("Le Figaro");
 Media o2 = new Livre("java head first",450);
 Media o3 = new DVD("home",120);
 System.out.println(o1);
 System.out.println(o2);
 System.out.println(o3);



 }
}
```

Correction

```
public class Media {
 private String titre;
 public Media(String unTitre){
 titre = unTitre;
 }
 public String getTitre(){
 return titre;
```

```
 }
 public String toString(){
 return "Media " + titre;
 }
}
```

```
public class Livre extends Media {
 private int nombreDePages;
 public Livre(String unTitre, int n){
 super(unTitre);
 nombreDePages= n;
 }
 public String toString(){
 return "Livre " + getTitre() + " " + nombreDePages + " pages";
 }
}
```

```
public class DVD extends Media {
 private int duree;
 public DVD(String unTitre, int n){
 super(unTitre);
 duree = n;
 }
 public String toString(){
 return "DVD " + getTitre() + " " + duree + " minutes";
 }
}
```

Aeroport

Cet exercice calcule le seuil de rentabilité des vols d’une compagnie aérienne.

Un aéroport est détermine par :
• Un nom de code (chaîne de caractères).
• Une longitude x (entier).
• Une latitude y (entier).
Ecrivez une classe Aeroport munie des attributs necessaires.
Ecrivez un constructeur prenant en param`etres les donnees necessaires a son initialisation.
Ecrivez un accesseur getNom du nom de code de l’aeroport.

Ecrivez une méthode distance(aero2) qui calcule et renvoie la distance d qui separe deux aeroports :

sqrt((x2-x1)^2 +(y2-y1)^2)

```
import java.lang.Math;
public class Aeroport {
private String m_nom; // nom de code
private int m_x; // longitude
private int m_y; // latitude
public Aeroport(String n, int x, int y){
m_nom = n;
m_x = x;
m_y = y;
}
public String getNom(){
return m_nom;
}
public double distance(Aeroport aero2){
double dx = (m_x - aero2.m_x);
double dy = (m_y - aero2.m_y);
return Math.sqrt(dx*dx + dy*dy);
}
}
```

Pour simplifier, on suppose que le coˆut d’un vol est independant du nombre de passagers transportes : il est uniquement fonction de la capacite k de l’avion, du nombre de membres d’equipage m et de la distance d entre les aeroports selon la formule :
100*d*sqrt(k+m)+50*sqrt(d)
Un vol sera donc constitue de :
• Un nom (chaine de caract`eres).
• Le nombre maximum de passagers k (entier positif) que l’avion peut transporter.
• Le nombre de membres d’equipage m (entier positif).
• La distance d (reel) entre les aeroports de depart et de destination

```
import java.lang.Math;
public class Vol {
private String m_nom; // nom du vol
private int m_k; // capacite (# de passagers)
private int m_m; // # de membres d’equipage
private double m_d; // distance
public Vol(String n, int k, int m, double d){
m_nom = n;
m_k = k;
m_m = m;
m_d = d;
}
public double eval(){return 100 * m_d * Math.sqrt((double)(m_k + m_m)) + 50 * Math.sqrt(m_d);
}
}
```

La compagnie a´erienne se pose la question suivante :
« Les billets ´etant vendus PRIX_BILLET euros, combien de passagers faut-il au minimum pour que le vol soit rentable ? ».
Votre programme doit :
• Instancier deux Aeroports et les initialiser avec les a´eroports de Mulhouse et de New-York.
• Calculer et afficher la distance entre les a´eroports.
• Instancier le Vol correspondant.
• Calculer et afficher le coˆut du vol.
• Demander le prix du billet.
• Calculer et afficher le nombre de passagers minimum pour que le vol soit rentabilis´e.

```
Distance entre MLH et JFK = 46.6154
Cout du vol = 82151.2
Prix du billet? 400
Nombre de billets a vendre = 206
```

```
import java.util.Scanner;
import java.util.Locale;
import java.lang.Math;
public class PGCompagnie{
public static void main(String[] args) {
Scanner cin = new Scanner(System.in);
cin.useLocale(Locale.US);
Aeroport a1 = new Aeroport("MLH", 2, 46);
Aeroport a2 = new Aeroport("JFK", 40, 73);
double d = a1.distance(a2);
System.out.println("Distance entre "+a1.getNom()
+" et "+a2.getNom()
+" = "+d);
Vol vol = new Vol("EPF 123", 300, 8, d);
double c = vol.eval();
System.out.println("Cout du vol = "+c);
System.out.print("Prix du billet? ");
double prixbillet = cin.nextDouble();
int np = (int)(Math.ceil(c / prixbillet));
System.out.println("Nombre de billets a vendre = "+np);
}
}
```

Aeroport2

Exercice 1 : Horaires.

On appelle horaire une donnée caractérisant un moment dans une journée, indépendamment de la date de
cette journée. Un horaire permet par exemple de préciser le moment du départ d'un train : "le train de
17h42 " où 17h42 représente l'horaire du train évoqué. Nous allons représenté un horaire par une instance
de la classe Horaire.
Une instance de Horaire est donc représentée par la donnée de deux nombres : le premier représente l'heure
(entre 0 et 23) et le second les minutes (entre 0 et 59).

La classe Horaire implémente l'interface Comparable en réalisant la relation d'ordre intuitive sur ces horaires. Elle propose en plus les méthodes suivantes :
• les accesseurs pour les heures et les minutes ;
• la méthode equals, deux horaires étant égaux s'ils ont mêmes heures et mêmes minutes ;
• une méthode

public int ecart(Horaire h)
dont le résultat est l'écart en minutes entre l'objet horaire invoquant la méthode et l'horaire h passé
en paramètre. Cette méthode déclenche une exception IllegalArgumentException si l'horaire h est plus tôt ("dans la journée") que l'horaire invoquant.
Q 1 . Donnez le code java de la méthode equals.
Q 2 . Le code java de la méthode ecart.
Q 3 . Pour toutes les méthodes de votre classe Horaire autres que les accesseurs et le constructeur donnez le code des méthodes de test que vous proposez.

Exercice 2 : Aéroport et vols.

Dans cet exercice nous utiliserons la classe util.Horaire définie à l'exercice précédent

On s'intéresse à la représentation de vols entre aéroports.

Les vols sont supposés être tous journaliers et il
s'agit de "vols courts" donc les horaires de départ et d'arrivée sont toujours dans la même journée. On ne
s'occupe donc pas des jours de départ, seul l'horaire compte.

Vols.

Un vol est caractérisé par un numéro unique (une chaîne de caractères), un horaire de départ et un horaire d'arrivée (de type Horaire), les aéroports de départ et de destination. Voici le diagramme de
la classe avion.Vol qui permet de représenter ces données :

```
Vol
- numero : String
- depart : Aeroport
- dest : Aeroport
- heureDepart : Horaire
- heureArrivee : Horaire
+ Vol(...)
+ getNumero() : String
+ getDepart() : Aeroport
+ getDestination() : Aeroport
+ getHeureDepart() : Horaire
+ getHeureArrivee() : Horaire
+ equals(o : Object) : boolean
+ hashCode() : int
```

Aéroports.

Un aéroport est caractérisé par un identifiant unique (une chaîne de caractères) et une table de hachage qui associe pour chacun des vols qui partent de cet aéroport, le numéro de vol à l'objet vol
correspondant.

Deux objets Aeroport sont donc égaux s'ils ont le même identifiant (on supposera que dans ce cas la liste des vols est nécessairement la même pour les deux objets).

Q 1 . Donnez le code java de l'entête de la classe avion.Aeroport ainsi que la déclaration de ses attributs
et de son constructeur sachant qu'initialement la table des vols est vide.

Q 2 . Donnez le code d’une méthode ajouteVol qui prend en paramètre un vol et l’ajoute à cet aéroport.
Cette méthode déclenche une IllegalArgumentException si cet aéroport n’est pas l’aéroport de départ de ce vol.

Q 3 . Donnez le code java d’une méthode volsDirects qui prend en paramètre un aéroport destination
dest. Cette méthode retourne la liste des vols qui partent de cette instance d’aéroport et dont la destination est dest.

Q 4 . Donnez le code java d’une méthode prochainVolDirect qui prend en paramètre un aéroport destination dest et un horaire h et dont le résultat est le numéro du premier vol de cette instance d’aéroport
qui part dans la journée pour dest après l’horaire h fourni. Une exception NoSuchElementException est déclenchée si aucun vol ne convient.

Donnez le code java du corps de la méthode suivante :

```
/** renvoie true ssi le prochain vol direct à destination de dest
* part dans moins de delai minutes de cet aéroport
* @param dest la destination
* @param delai le délai maximal avant le départ du prochain vol pour dest
* @return true si le prochain vol direct à destination de dest
* part dans moins de delai minutes de cet aéroport et
* false si ce n’est pas le cas ou s’il n’y pas de prochain vol direct
*/
public boolean volEnPartance(Aeroport dest, int delai) {
...
}
```

Q 7 . Donnez le code java d’une méthode volsParDestination dont le résultat est la table de hachage
qui pour cet (this) aéroport associe à chaque aéroport a la liste des vols directs partant de this dont a est la destination.