JAVA: projet final THREADS

Anaëlle & Anna

Exemple simple de code avec plusieurs threads

Qu’est ce qu’un thread ?

En Java, un thread est un objet de la classe *Thread* dans le *package java.lang*.

Un “fil d’exécution” ...

Un *thread* désigne un « fil d'exécution » dans le programme. C'est-à-dire une suite linéaire et continue d'instructions qui sont exécutées les unes après les autres.

Le langage Java est *multi-thread*, c'est-à-dire qu'il peut faire cohabiter plusieurs fils d'exécution de façon indépendante.

Ils permettent de réaliser plusieurs tâches en même temps sans interrompre le *main* et de rendre un programme plus efficace.

Partie 1 - Exemple de l'utilisation de THREAD

Deux threads devinent un nombre écrit dans le *main* du programme

```java
package projetEncheresPartie1;

public class GuessANumber extends Thread {
	private int number;
	public GuessANumber(int number) {
		this.number = number;
	}

	public void run() {
		int counter = 0;
		int guess = 0;
		do {
			try
			{
				guess = (int) (Math.random() * 10 + 1);
				System.out.println(this.getName() + " dit : " + guess);
				counter++;
				Thread.sleep(1000);
			}
			catch (InterruptedException e)
			{
				// Activation du flag d'interruption
				Thread.currentThread().interrupt();
			}
		} while(guess != number && ! isInterrupted());

		if (guess == number )
			System.out.println("** Correct! " + this.getName() + " deviné en " + counter + " essais.**");
	}
}
```

```java
package projetEncheresPartie1;

public class ThreadClassDemo {

    public static void main(String [] args) throws InterruptedException {

        Thread thread = new GuessANumber(2);
        Thread thread2 = new GuessANumber(2);
        thread.setName("Le premier utilisateur");
        thread2.setName("Le deuxième utilisateur");
        System.out.println(thread.getName() + " commence...");
        thread.start();
        System.out.println(thread2.getName() + " commence...");
        thread2.start();

        boolean thread1Alive = true;
        boolean thread2Alive = true;
        while(thread1Alive && thread2Alive)
        {
            Thread.sleep(100);
            thread1Alive = thread.isAlive();
            thread2Alive = thread2.isAlive();
        }

        if(thread1Alive)
        {
            thread.interrupt();
            thread.join();
            System.out.println(thread2.getName() + " a gagné! ");
        }
        else
        {
            thread2.interrupt();
            thread2.join();
            System.out.println(thread.getName() + " a gagné! ");
        }
    }
}
```

Deviner le nombre 2

```
Le premier utilisateur commence...
Le deuxième utilisateur commence...
Le premier utilisateur dit : 4
Le deuxième utilisateur dit : 4
Le premier utilisateur dit : 9
Le deuxième utilisateur dit : 1
Le premier utilisateur dit : 5
Le deuxième utilisateur dit : 3
Le premier utilisateur dit : 4
Le deuxième utilisateur dit : 2
Le premier utilisateur dit : 9
** Correct! Le deuxième utilisateur deviné en 4 essais.**
Le deuxième utilisateur a gagné!
```

Chaque utilisateur correspond à un thread.
Dès qu'un thread a deviné le nombre, les deux s'arrêtent.

Partie 2 - Les enchères

À partir de la classe bibliothécaire, utiliser des threads pour permettre à des emprunteurs d'enchérir sur un livre à emprunter.

- Personne
- Acheteur
- ComissairePriseur
- CommissairePriseurTest
- EnchereFailure
- EnchereTooLowException
- EncherisseurThread
- LivreEnchere
- VendeurLivresAuxEncheres

Hiérarchie complète du projet

Class Hierarchy

- java.lang.**Object**
  - traitementTextes.bibliotheque.**Amende**
  - traitementTextes.bibliotheque.**Archivage**
  - traitementTextes.bibliotheque.**Auteur** (implements java.io.Serializable)
  - traitementTextes.bibliotheque.**Bibliothecaire** (implements traitementTextes.bibliotheque.VendeurLivresAuxEncheres)
  - traitementTextes.bibliotheque.**BibliothecaireVerificateur**
  - traitementTextes.bibliotheque.**ComissairePriseur**
  - traitementTextes.bibliotheque.**Livre** (implements java.io.Serializable)
    - traitementTextes.bibliotheque.**LivreEnchere**
  - traitementTextes.bibliotheque.**LivreEmprunte**
  - traitementTextes.bibliotheque.**Personne**
    - traitementTextes.bibliotheque.**Acheteur**
    - traitementTextes.bibliotheque.**Lecteur**
      - traitementTextes.bibliotheque.**Etudiant**
      - traitementTextes.bibliotheque.**Travailleur**
  - java.lang.**Thread** (implements java.lang.Runnable)
    - traitementTextes.bibliotheque.**EncherisseurThread**
  - java.lang.**Throwable** (implements java.io.Serializable)
    - java.lang.**Exception**
      - traitementTextes.bibliotheque.**EnchereFailure**
      - traitementTextes.bibliotheque.**EnchereTooLowException**

Interface Hierarchy

- traitementTextes.bibliotheque.**VendeurLivresAuxEncheres**

Héritage

L'idée est qu'une nouvelle classe *extends* une autre déjà existante dans le programme

```java
package traitementTextes.bibliotheque;

/**
 * Classe qui présente les informations sur la personne
 * comprenant son nom, prenom et l'age.
 *
 * @author Anna Niskovskikh et Anaëlle Pierredon
 * @version 1.0
 */

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

    /**
     * Constructeur de la classe Personne
     * @param nom Nom de famille de la personne
     * @param prenom Prenom de la personne
     */
    public Personne(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }

    public String getNom() {
        return nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```java
public abstract class Lecteur extends Personne{

    private Amende amende;
    private boolean livreEnRetard;

    /**
     * Constructeur de la classe Lecteur
     * @param nom Nom de famille de lecteur
     * @param prenom prenom de lecteur
     */
    public Lecteur(String nom, String prenom) {
        super(nom, prenom);
    }
```

Polymorphisme

Une même méthode agit différemment selon la classe à laquelle elle appartient, dans le cadre de l'héritage.

Dans la classe Lecteur ...

```
public void afficherID(Lecteur lecteur) {
    System.out.println(lecteur.getNom()+lecteur.getPrenom());
}
```

Dans la classe Etudiant qui hérite de Lecteur ...

```
public void afficherID(Etudiant etudiant) {
    System.out.println(etudiant.getNumeroEtudiant());
}
```

Encapsulation

Permet de faire en sorte que les attributs d'une classe ne puissent pas être manipulés directement à l'extérieur de cette classe autrement qu'en utilisant des méthodes (*private*, *get() et set()*).

```
private String etablissement;
private int numeroEtudiant;
```

```
public String getEtablissement() {
    return etablissement;
}
public void setEtablissement(String etablissement) {
    this.etablissement = etablissement;
}

public int getNumeroEtudiant() {
    return numeroEtudiant;
}

public void setNumeroEtudiant(int numeroEtudiant) {
    this.numeroEtudiant = numeroEtudiant;
}
```

Interface

L'idée est qu'une classe *implements* une interface et qu'elle soit indépendante et ait son propre comportement

ce qui s'appelle *l'injection de dépendances*

```
public class Bibliothecaire implements VendeurLivresAuxEncheres {
```

```
package traitementTextes.bibliotheque;

/***
 * Cette interface permet à la classe Bibliothécaire
 * de se transformer au vendeur de livres et
 * de posséder une liste de livres pour mener des enchères.
 */

import java.util.ArrayList;

public interface VendeurLivresAuxEncheres {

    ArrayList<LivreEnchere> getLivreAVendreAuxEncheres();
}
```

Surcharge

L'idée est d'avoir deux (ou plus) méthodes avec le même nom dans une seule classe avec des paramètres différents (*startEnchere()* dans notre cas).

```java
public ArrayList<Acheteur> startEnchere(VendeurLivresAuxEncheres vendeurLivresAuxEncheres) throws EnchereFailure {
    ArrayList<Acheteur> acheteurList = new ArrayList<Acheteur>();
    for(var livreEnchere : vendeurLivresAuxEncheres.getLivreAVendreAuxEncheres()) {
        var winner = startEnchere(livreEnchere);
        if(winner != null)
            acheteurList.add(winner);
    }
    return acheteurList;
}

 * Cette méthode permet de créer des Threads, les lancer,
public Acheteur startEnchere(LivreEnchere livreEnchere) throws EnchereFailure {
        System.out.println( "La mise a prix de ce superbe livre de " + livreEnchere.getAuteur().getNom()
                + " est de " + livreEnchere.getDebutEnchere());
        enchereCourante = 0;
        // Creation des threads
       var encherisseurThreadList = creerThreads(encherisseurList,livreEnchere);

      //Start Thread
      encherisseurThreadList.forEach(thread -> thread.start());

      // attendre fin de thread
      var winnerThread = waitForAWinner(encherisseurThreadList);
      if (winnerThread == null)
          return null;

      // stopper tous les thread
      stopAllThread(encherisseurThreadList);
```

Gestion des exceptions

L'idée est de personnaliser des exceptions existant dans le programme (*extends Exception*)

Mots-clés :

- *throw* permet de créer une exception dans une méthode
- *throws* indique le type de l'exception

```
throws EnchereFailure
```

```
catch (InterruptedException e) {
  throw new EnchereFailure("Enchere interrompue.");
```

Les tests unitaires

Ils permettent de vérifier que notre code donne les résultats attendus.

```java
@Test
void testStartEnchere() {
    //GIVEN
    Auteur tolstoy = new Auteur("Tolstoy");
    LivreEnchere guerreEtPaix = new LivreEnchere(tolstoy, "Guerre et Paix", 15);

    ComissairePriseur comissairePriseur = new ComissairePriseur();

    Acheteur acheteur1 = new Acheteur("Pierredon", "Anaëlle", tolstoy);
    Acheteur acheteur2 = new Acheteur("Niskovskikh", "Anna", tolstoy);
    comissairePriseur.addEncherisseur(acheteur1);
    comissairePriseur.addEncherisseur(acheteur2);

    Acheteur winner;
    //WHEN
    try {
        winner = comissairePriseur.StartEnchere(guerreEtPaix);
    }
    catch(EnchereFailure e)
    {
        System.out.println( "Exception catché. Le test a échoué");
        return;
    }

    //THEN
    assertNotNull(winner);
    assertTrue(winner.getPrenom().equals(acheteur1.getPrenom()) || winner.getPrenom().equals(acheteur2.getPrenom()));

}
```

Démonstration du code sur Eclipse

```
La mise a prix de ce superbe livre de Tolstoy est de 15
Nouvelle enchere de Anaëlle : 22 !
Nouvelle enchere de Anna : 23 !
Nouvelle enchere de Anaëlle : 28 !
Nouvelle enchere de Anna : 32 !
Nouvelle enchere de Anaëlle : 34 !
Nouvelle enchere de Anna : 41 !
L enchere de Anaëlle de 41 a echoué.
L enchere de Anaëlle de 42 a echoué.
Nouvelle enchere de Anna : 42 !
Nouvelle enchere de Anaëlle : 49 !
L enchere de Anna de 46 a echoué.
Anaëlle ne veut plus encherir.
Nouvelle enchere de Anna : 54 !
1!
2!
3!
Anna remporte l'enchere! Le montant de l'enchère est de 54.
La mise a prix de ce superbe livre de Tolstoy est de 15
Anaëlle n est pas interesse par Tolstoy
Anna n est pas interesse par Tolstoy
La mise a prix de ce superbe livre de Hugo est de 10
Nouvelle enchere de Anaëlle : 19 !
L enchere de Anna de 12 a echoué.
Anna ne veut plus encherir.
1!
2!
3!
Anaëlle remporte l'enchere! Le montant de l'enchère est de 19.
```

- La valeur ajoutée à l'enchère à chaque tour est choisie au hasard entre 0 et 10.
- Si un des acheteurs essaye d'enchérir une somme inférieure à celle donnée par l'autre acheteur, l'enchère échoue.
- Selon l'intérêt des acheteurs ou non pour un livre, l'enchère peut avoir lieu ou non.