

# COMPT RENDU TP-4 POO -JAVA-

CLASSES ABSTRAITES

INTERFACES

Encadrer par

**Prof : M.MOUKHAFI**

Préparer par

**Zakaria El Omari**

Informatique S5

# SOMMAIRE

- 01**      Exercice 1 : Les Interfaces.
- 02**      Exercice 2 : Les Classes Abstraites.
- 03**      Exercice 3 : Les Threads.
- 04**      La Recherche De Quelques Notions En  
            JAVA.

**EXERCICE 1 :**

**LES INTERFACES.**

# EXERCICE 1 :

## LES INTERFACES.

```
package interfaces;
```

```
public interface A {  
    public int x = 7;  
    public void f();  
}
```

```
package interfaces;
```

```
public interface B {  
    public int x = 2;  
    public void f();  
}
```

```
package classes;
```

```
import interfaces.*;
```

```
public class C implements A,B{
```

```
    public void f() {  
        // Implementation de f()  
        System.out.println("Ok Interface ");  
    }  
}
```

# EXERCICE 1 :

## LES INTERFACES.

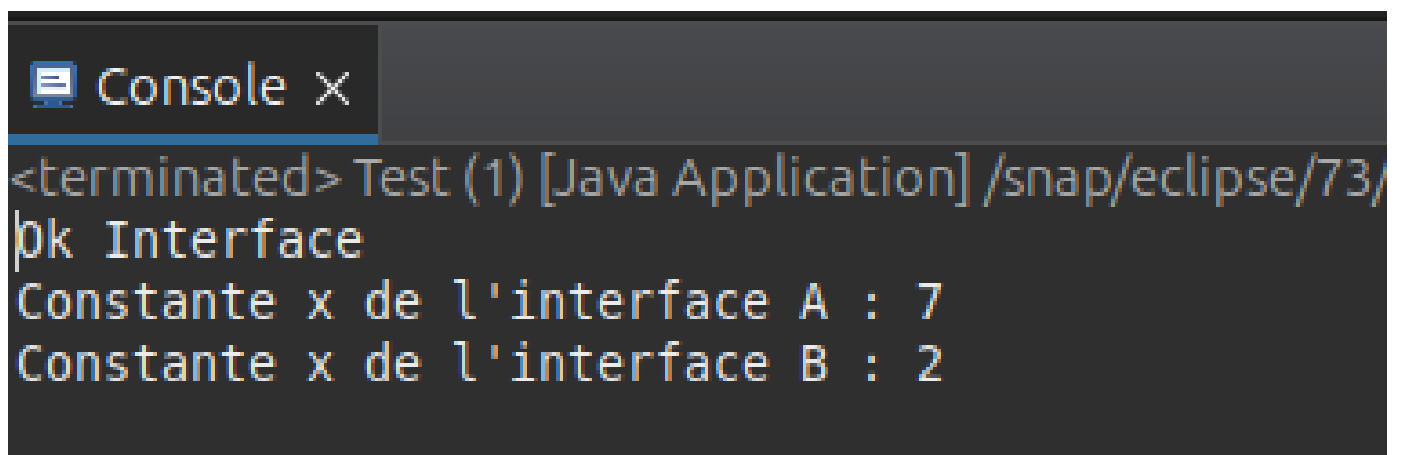
```
package Presentation;

import classes.*;
import interfaces.*;

public class Test {
    public static void main(String[] args) {
        C instanceC = new C();

        // Appel de la méthode f() de la classe C
        instanceC.f();

        // Accès aux constantes x des interfaces A et B
        System.out.println("Constante x de l'interface A : " + A.x);
        System.out.println("Constante x de l'interface B : " + B.x);
    }
}
```



The screenshot shows a console window titled "Console x" with the following output:

```
<terminated> Test (1) [Java Application] /snap/eclipse/73/
pk Interface
Constante x de l'interface A : 7
Constante x de l'interface B : 2
```

## **EXERCICE 2 : LES CLASSES ABSTRAITES.**

# EXERCICE 2 : LES CLASSES ABSTRAITES.

```
public abstract class A {  
    public abstract void methodeAbstraiteA(); // Méthode abstraite dans la classe A  
    public void methodeConcreteA() {  
        System.out.println("Méthode concrète de la classe A");  
    }  
}
```

```
public abstract class B extends A {  
    public abstract void methodeAbstraiteB(); // Méthode abstraite dans la classe B  
    public void methodeConcreteB() {  
        System.out.println("Méthode concrète de la classe B");  
    }  
}
```

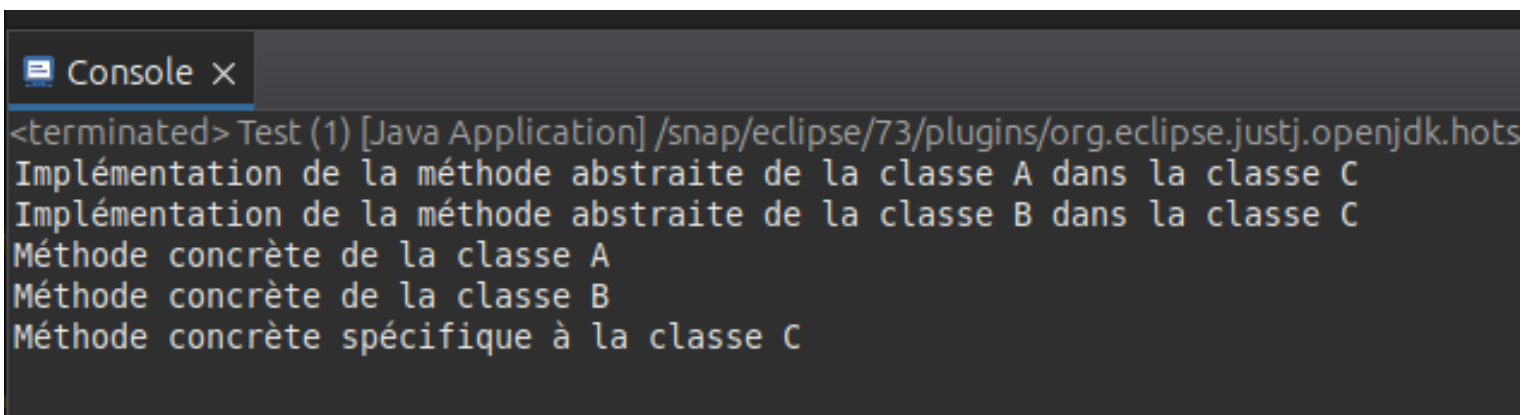
```
public class C extends B {  
    // Implémentation des méthodes abstraites de A et B  
    public void methodeAbstraiteA() {  
        System.out.println("Implémentation de la méthode abstraite de la classe A dans la classe C");  
    }  
}
```

```
    public void methodeAbstraiteB() {  
        System.out.println("Implémentation de la méthode abstraite de la classe B dans la classe C");  
    }  
}
```

```
    // Méthode concrète spécifique à la classe C  
    public void methodeConcreteC() {  
        System.out.println("Méthode concrète spécifique à la classe C");  
    }  
}
```

# EXERCICE 2 : LES CLASSES ABSTRAITES.

```
public class Test {  
    public static void main(String[] args) {  
        C instanceC = new C();  
  
        // Appels de méthodes de la hiérarchie  
        instanceC.methodeAbstraiteA();  
        instanceC.methodeAbstraiteB();  
        instanceC.methodeConcreteA();  
        instanceC.methodeConcreteB();  
        instanceC.methodeConcreteC();  
    }  
}
```



The screenshot shows a console window titled "Console x" with the following output:

```
<terminated> Test (1) [Java Application] /snap/eclipse/73/plugins/org.eclipse.justj.openjdk.hotspot  
Implémentation de la méthode abstraite de la classe A dans la classe C  
Implémentation de la méthode abstraite de la classe B dans la classe C  
Méthode concrète de la classe A  
Méthode concrète de la classe B  
Méthode concrète spécifique à la classe C
```

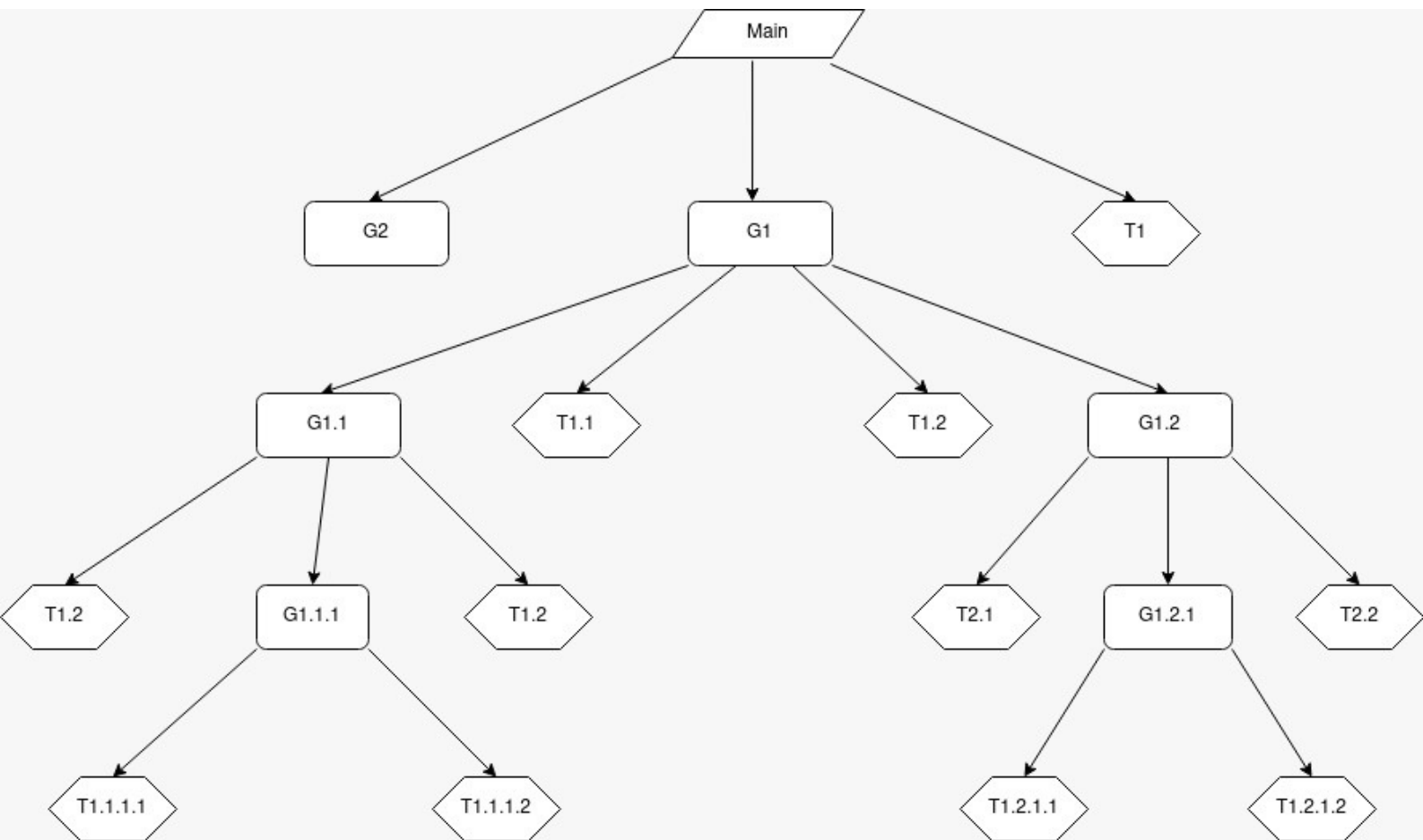


**EXERCICE 3 :**

**LES THREADS.**

# EXERCICE 3 :

# LES THREADS.



## EXERCICE 3 :

## LES THREADS.

```
package classes;
```

```
public class Proc extends Thread{  
    private int x;
```

```
    public Proc() {  
        super();  
    }
```

```
    public Proc(int x) {  
        this();  
        this.x = x;  
    }
```

```
    public Proc(int x, ThreadGroup group, String str) {  
        super(group,str);  
        this.x = x;  
    }
```

```
    public void run() {  
        for (int i = 0; i < x; i++) {  
            System.out.println("processus 1 : "+ i);
```

```
        }  
        try {  
            sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

## EXERCICE 3 :

## LES THREADS.

```
package Presentation;
```

```
import classes.*;
```

```
public class MaClasse {
```

```
    private static void displayThreadGroup(Thread thread) {  
        System.out.println(thread.getName() + " belongs to ThreadGroup: " +  
            thread.getThreadGroup().getName());  
    }
```

```
    public static void main(String[] args) {
```

```
        // Premier niveau
```

```
        Proc t1 = new Proc(5);
```

```
        displayThreadGroup(t1);
```

```
        ThreadGroup g1 = new ThreadGroup("g1");
```

```
        ThreadGroup g2 = new ThreadGroup("g2");
```

```
        // Deuxième niveau
```

```
        Proc t1_1 = new Proc(3, g1, "t1.1");
```

```
        Proc t1_2 = new Proc(4, g1, "t1.2");
```

```
        displayThreadGroup(t1_1);
```

```
        displayThreadGroup(t1_2);
```

```
        ThreadGroup g1_1 = new ThreadGroup(g1, "g1.1");
```

```
        ThreadGroup g1_2 = new ThreadGroup(g1, "g1.2");
```

## EXERCICE 3 :

## LES THREADS.

// Troisième niveau

```
Proc t1_1_1 = new Proc(2, g1_1, "t1.1.1");  
Proc t1_1_2 = new Proc(2, g1_1, "t1.1.2");  
displayThreadGroup(t1_1_1);  
displayThreadGroup(t1_1_2);
```

```
ThreadGroup g1_1_1 = new ThreadGroup(g1_1, "g1.1.1");
```

```
Proc t1_2_1 = new Proc(2, g1_2, "t1.2.1");  
Proc t1_2_2 = new Proc(2, g1_2, "t1.2.2");  
displayThreadGroup(t1_2_1);  
displayThreadGroup(t1_2_2);
```

```
ThreadGroup g1_2_1 = new ThreadGroup(g1_2, "g1.2.1");
```

// Quatrième niveau

```
Proc t1_1_1_1 = new Proc(2, g1_1_1, "t1.1.1.1");  
Proc t1_1_1_2 = new Proc(2, g1_1_1, "t1.1.1.2");  
Proc t1_2_1_1 = new Proc(2, g1_2_1, "t1.2.1.1");  
Proc t1_2_1_2 = new Proc(2, g1_2_1, "t1.2.1.2");  
displayThreadGroup(t1_1_1_1);  
displayThreadGroup(t1_1_1_2);  
displayThreadGroup(t1_2_1_1);  
displayThreadGroup(t1_2_1_2);
```

```
}  
}
```

# EXERCICE 3 : LES THREADS EN UTILISANT RUNNABLE.

```
package business;
```

```
public class ProcRunnable implements Runnable {  
    private int x;
```

```
    public ProcRunnable() {  
        super();  
    }
```

```
    public ProcRunnable(int x) {  
        this();  
        this.x=x;  
    }
```

```
    public ProcRunnable(int x,ThreadGroup group, String str) {  
        super();  
        this.x=x;  
    }
```

```
    public void run() {  
        for (int i = 0; i < x; i++) {  
            System.out.println("processus 1 : " + i );  
            try {  
                Thread.sleep(4000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

# EXERCICE 3 : LES THREADS EN UTILISANT RUNNABLE.

```
public class MaClasse {  
  
    private static void displayThreadGroup(Thread thread) {  
        System.out.println(thread.getName() + " belongs to  
ThreadGroup: " + thread.getThreadGroup().getName());  
    }  
  
    public static void main(String[] args) {  
        // Premier niveauThread t1 = new Thread(new  
ProcRunnable(5));  
        displayThreadGroup(t1);  
  
        ThreadGroup g1 = new ThreadGroup("g1");  
        ThreadGroup g2 = new ThreadGroup("g2");  
  
        // Deuxième niveauThread t1_1 = new Thread(g1, new  
ProcRunnable(3), "t1.1");  
        Thread t1_2 = new Thread(g1, new ProcRunnable(4),  
"t1.2");  
        displayThreadGroup(t1_1);  
        displayThreadGroup(t1_2);  
  
        ThreadGroup g1_1 = new ThreadGroup(g1, "g1.1");  
        ThreadGroup g1_2 = new ThreadGroup(g1, "g1.2");
```

# EXERCICE 3 : LES THREADS EN UTILISANT RUNNABLE.

```
// Troisième niveau
Thread t1_1_1 = new Thread(g1_1, new ProcRunnable(2), "t1.1.1");
Thread t1_1_2 = new Thread(g1_1, new ProcRunnable(2), "t1.1.2");
displayThreadGroup(t1_1_1);
displayThreadGroup(t1_1_2);

ThreadGroup g1_1_1 = new ThreadGroup(g1_1, "g1.1.1");

Thread t1_2_1 = new Thread(g1_2, new ProcRunnable(2), "t1.2.1");
Thread t1_2_2 = new Thread(g1_2, new ProcRunnable(2), "t1.2.2");
displayThreadGroup(t1_2_1);
displayThreadGroup(t1_2_2);

ThreadGroup g1_2_1 = new ThreadGroup(g1_2, "g1.2.1");

// Quatrième niveau
Thread t1_1_1_1 = new Thread(g1_1_1, new ProcRunnable(2),
"t1.1.1.1");
Thread t1_1_1_2 = new Thread(g1_1_1, new ProcRunnable(2),
"t1.1.1.2");
Thread t1_2_1_1 = new Thread(g1_2_1, new ProcRunnable(2),
"t1.2.1.1");
Thread t1_2_1_2 = new Thread(g1_2_1, new ProcRunnable(2),
"t1.2.1.2");
displayThreadGroup(t1_1_1_1);
displayThreadGroup(t1_1_1_2);
displayThreadGroup(t1_2_1_1);
displayThreadGroup(t1_2_1_2);

}
}
```



## EXERCICE 3 :

## LES THREADS.

 Console x

```
<terminated> MaClasse (1) [Java Application] /snap/eclipse/73,  
|Thread-0 belongs to ThreadGroup: main  
t1.1 belongs to ThreadGroup: g1  
t1.2 belongs to ThreadGroup: g1  
t1.1.1 belongs to ThreadGroup: g1.1  
t1.1.2 belongs to ThreadGroup: g1.1  
t1.2.1 belongs to ThreadGroup: g1.2  
t1.2.2 belongs to ThreadGroup: g1.2  
t1.1.1.1 belongs to ThreadGroup: g1.1.1  
t1.1.1.2 belongs to ThreadGroup: g1.1.1  
t1.2.1.1 belongs to ThreadGroup: g1.2.1  
t1.2.1.2 belongs to ThreadGroup: g1.2.1
```

# **LA RECHERCHE DE QUELQUES NOTIONS EN JAVA.**

# LA RECHERCHE DE QUELQUES NOTIONS EN JAVA.

- Modèle MVC-2 (Modèle-Vue-Contrôleur-2) :
  - Le modèle MVC-2 est une architecture logicielle qui sépare les composants d'une application en trois parties :
    - Modèle (Model) : Représente la logique métier et les données de l'application.
    - Vue (View) : Gère l'interface utilisateur et l'affichage des données provenant du modèle.
    - Contrôleur (Controller) : Gère les interactions de l'utilisateur, met à jour le modèle et actualise la vue.
- DAO (Data Access Object) :
  - Le modèle DAO est un motif de conception qui sépare la logique d'accès aux données de la logique métier dans une application.
    - Interface DAO : Définit les opérations CRUD (Create, Read, Update, Delete) pour une entité métier spécifique.
    - Implémentation DAO : Une ou plusieurs classes implémentent l'interface DAO pour interagir avec une source de données (base de données, fichier, etc.).
    - Isolation des couches : Permet de changer la source de données sans affecter la logique métier.
    - Facilite les tests : Permet de substituer une implémentation DAO réelle par une implémentation mock pour les tests unitaires.

# LA RECHERCHE DE QUELQUES NOTIONS EN JAVA.

- Applets :
  - Les applets Java sont des programmes autonomes s'exécutant dans une machine virtuelle Java (JVM).
    - Interface graphique : Peut créer une interface utilisateur graphique à l'aide de bibliothèques Java telles que Swing ou AWT.
    - Intégration avec d'autres applications : Peut être intégré en tant que composant réutilisable dans des applications Java.
    - Exécution en tant qu'application indépendante : Peut être exécuté en tant qu'application indépendante avec une configuration appropriée.
    - Accès aux ressources locales : A un accès limité aux ressources locales en raison des restrictions de sécurité, mais peut effectuer certaines opérations en fonction de la politique de sécurité de l'environnement d'exécution.

# LA RECHERCHE DE QUELQUES NOTIONS EN JAVA.

- Cycle des threads (Slide 7) :
  - Le cycle de vie des threads en Java passe par plusieurs étapes. Dans le cycle des threads en Java, plusieurs étapes marquent la vie d'un thread depuis sa création jusqu'à sa terminaison :
    - Nouveau (New) : Le thread est créé mais n'a pas encore été démarré avec la méthode `start()`. À ce stade, il est considéré comme non démarré.
    - Prêt (Runnable) : Après l'appel de `start()`, le thread est prêt à être exécuté. Il attend son tour pour être sélectionné par le planificateur de tâches.
    - En cours d'exécution (Running) : Le thread est actuellement en cours d'exécution, exécutant son code.
    - Bloqué (Blocked) : Le thread peut passer à cet état lorsqu'il est mis en pause, en attendant l'acquisition d'un verrou ou en attente d'une entrée/sortie.
    - En attente (Waiting) : Le thread est en attente, généralement après avoir appelé la méthode `wait()`, et attend qu'un autre thread le réveille à l'aide de `notify()` ou `notifyAll()`.
    - En attente de délai (Timed Waiting) : Le thread peut être en attente de délai après l'utilisation de méthodes telles que `sleep()` ou `join()` avec une durée spécifiée.
    - Terminé (Terminated) : Une fois que le thread a terminé son exécution, soit naturellement soit en raison d'une exception non capturée, il entre dans cet état.

# LA RECHERCHE DE QUELQUES NOTIONS EN JAVA.

- Mémoire d'un JAR après exécution (maximum 512 Mo) :
  - La mémoire maximale qu'un JAR peut utiliser après son exécution dépend de divers facteurs tels que les paramètres de la machine virtuelle Java (JVM) lors de son lancement. Cette limite est généralement spécifiée à l'aide de l'option de ligne de commande -Xmx suivie de la taille maximale, comme -Xmx512m, qui signifie 512 mégaoctets.
- Priorité par défaut d'un thread (5) et priorité maximale (10) :
  - En Java, les threads ont des priorités allant de 1 à 10. La priorité par défaut est normalement définie à 5, et la priorité maximale est 10. Les threads de priorité plus élevée ont une chance plus élevée d'être choisis pour l'exécution, mais cela dépend également de la stratégie d'ordonnancement de la JVM.
- isDaemon() :
  - La méthode isDaemon() est utilisée pour déterminer si un thread est un thread daemon (démon). Un thread daemon est un thread qui s'exécute en arrière-plan et ne bloque pas l'arrêt de l'application. Vous pouvez appeler cette méthode sur une instance de la classe Thread pour savoir si le thread est un daemon ou non.
- API ThreadGroup :
  - La classe ThreadGroup en Java est utilisée pour créer et manipuler des groupes de threads. Elle fournit un moyen de gérer plusieurs threads en tant qu'unité unique. Vous pouvez utiliser les méthodes de cette classe pour manipuler et contrôler les threads appartenant à un même groupe.