GEN
Complément pour le mini-projet

Objets Actifs en Java
Threads

Aperçu

1. En préliminaire
2. Diagramme d'états et de transitions
3. Réaliser des objets actifs en Java
4. Politique d'allocation du processeur
5. Exclusion mutuelle
6. Rendez-vous
7. Divers

1. En préliminaire

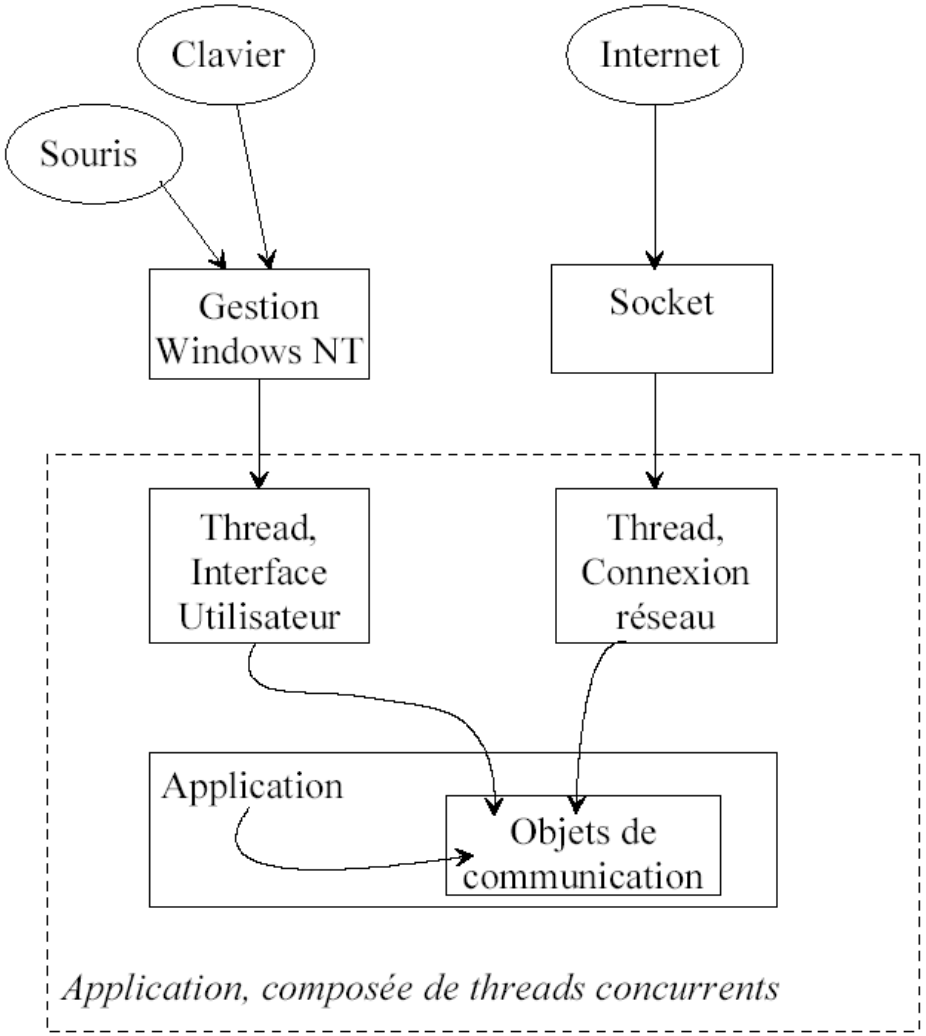
Les threads sont des processus légers

	Threads	⇨ *Concrètement en Java*
Mémoire	Mémoire partagée. Même espace d'adressage.	⇨ Deux threads peuvent référencer la même variable
Code	Partage des mêmes instructions.	⇨ Deux threads peuvent invoquer la même méthode
Ressources	Partage des mêmes ressources.	⇨ Deux threads peuvent travailler sur le même fichier
Fiabilité	Les threads ne sont pas protégés entre eux (corruption de l'application possible).	⇨ Ne pas faire n'importe quoi!

Pourquoi utiliser plusieurs threads dans la même application?

➔ Permet de gérer des événements provenant de sources différentes,

- de manière **concurrente,**
- et **indépendante**

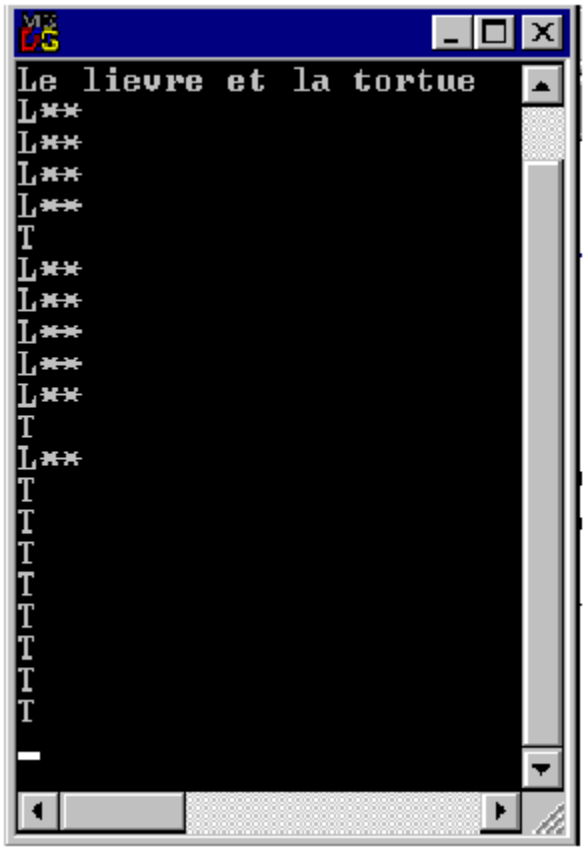


Prenons un premier exemple: le lièvre et la tortue (1)...

```
class Animal extends Thread {

   private String nom;      Nom de l'animal
   private int période;     Période entre 2 affichages

   public Animal(String nom, int période) {
      this.période = période;
      this.nom = nom;
   }

   public void run(){       Code de l'activité

      int distance = 10;  // Distance à parcourir
      while(distance > 0){
         try {Thread.sleep(période);}        Pause: l'animal court...

         catch(InterruptedException e) { }
         System.out.println(nom);
         distance--;
      }
   }

}
```

Le lièvre et la tortue (2)...

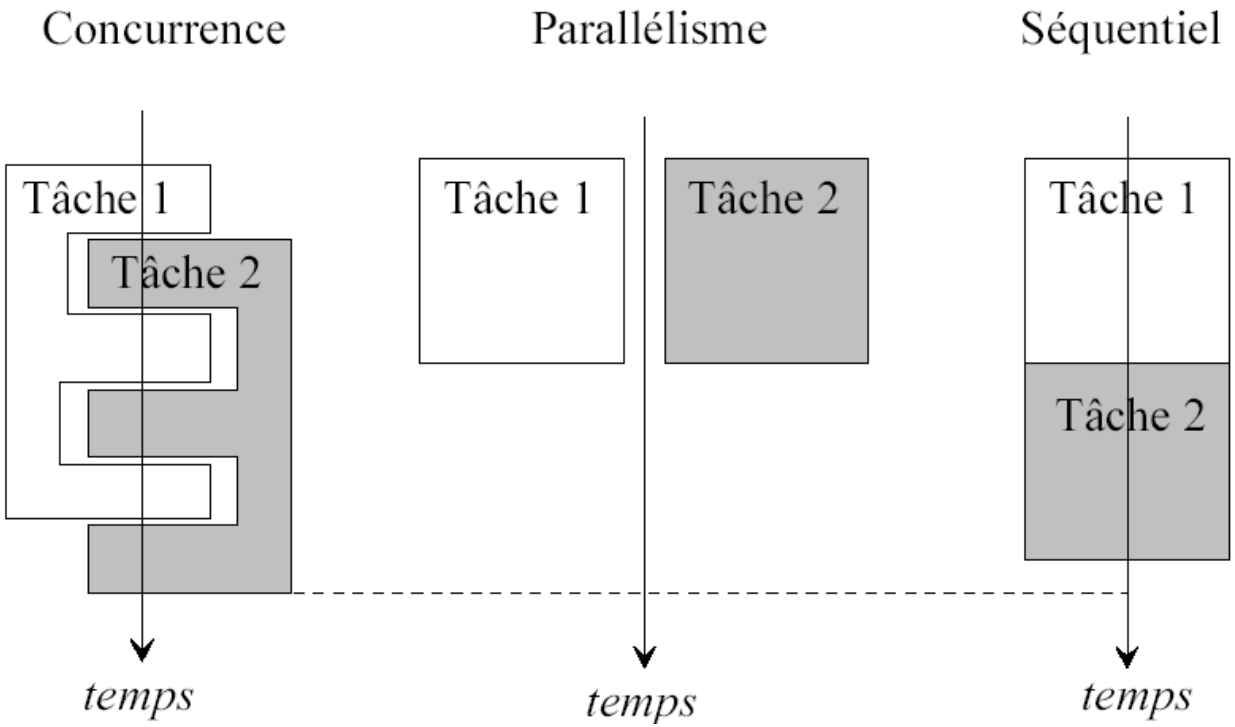
```
public class LievreEtTortue {
   public static void main(String arg[]) {
      System.out.println("Le lièvre et la tortue");
      Animal lièvre = new Animal("L**", 20);
      Animal tortue = new Animal("T", 100);
      tortue.start();      // Démarrer les activités
      lièvre.start();
      try {int j = 0; j = System.in.read();}
      catch(IOException e) { }; Attendre RETURN
   }
}
```



Le lièvre affiche des "L**" toutes les 20 msec et la tortue des "T" toutes les 100 msec.

Lequel arrive en premier?

Concurrence ou parallélisme?



Comment travaille la JVM?

- La JVM propose un système de threads concurrents
- En s'appuyant sur la mécanique sous-jacente du OS

 Le parallélisme offert par certaines plateformes peut être exploité.

Exemple: Windows NT/2000, threads exécutés en parallèle

Mais..

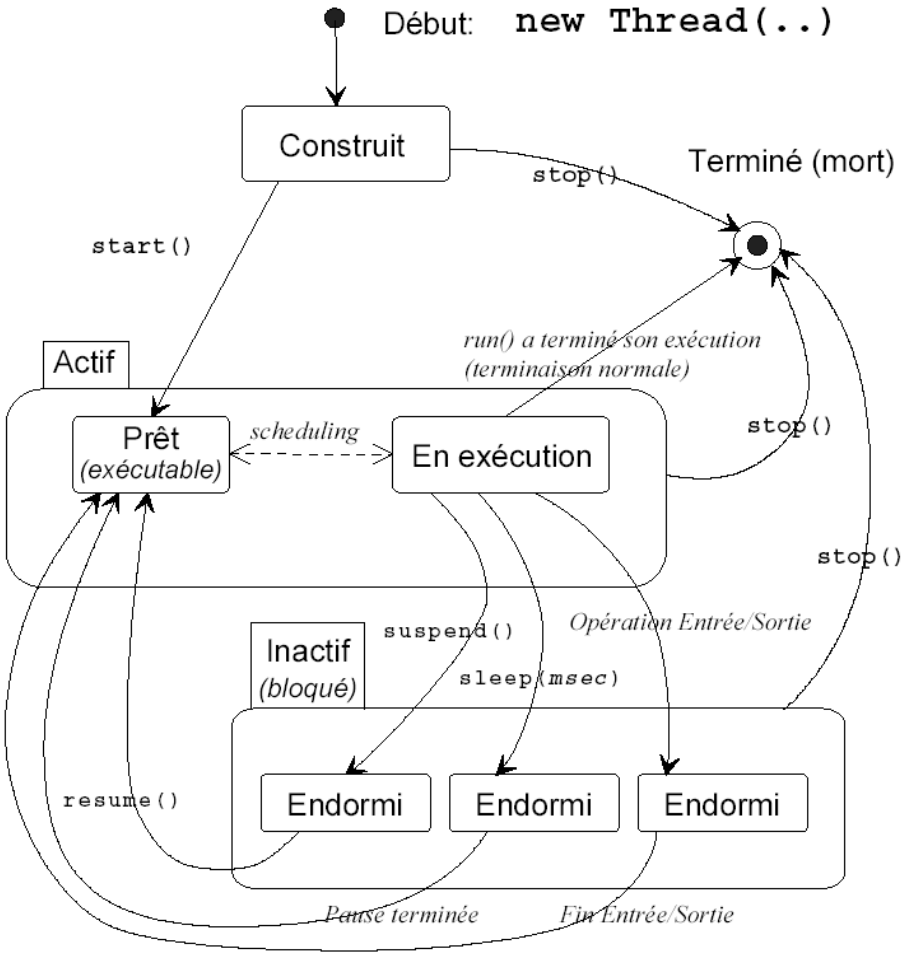
 Les applications Java ne sont plus portables d'un système à l'autre!

Exemple:

- Le « time-slicing » n'est mis en œuvre que sur certains systèmes

⇨Tester les résultats sur tout les systèmes

2. Diagramme d'états et de transitions



- Les threads vivants mais déférencés ne sont pas détruits!!

  La JVM conserve en effet une référence sur tous les threads actifs.

  Un thread déférencé continue donc son exécution!

- Contrôler l'état d'un thread: `isAlive()`

  Retourne `true` si le thread est dans l'état **Actif** ou **Inactif**.

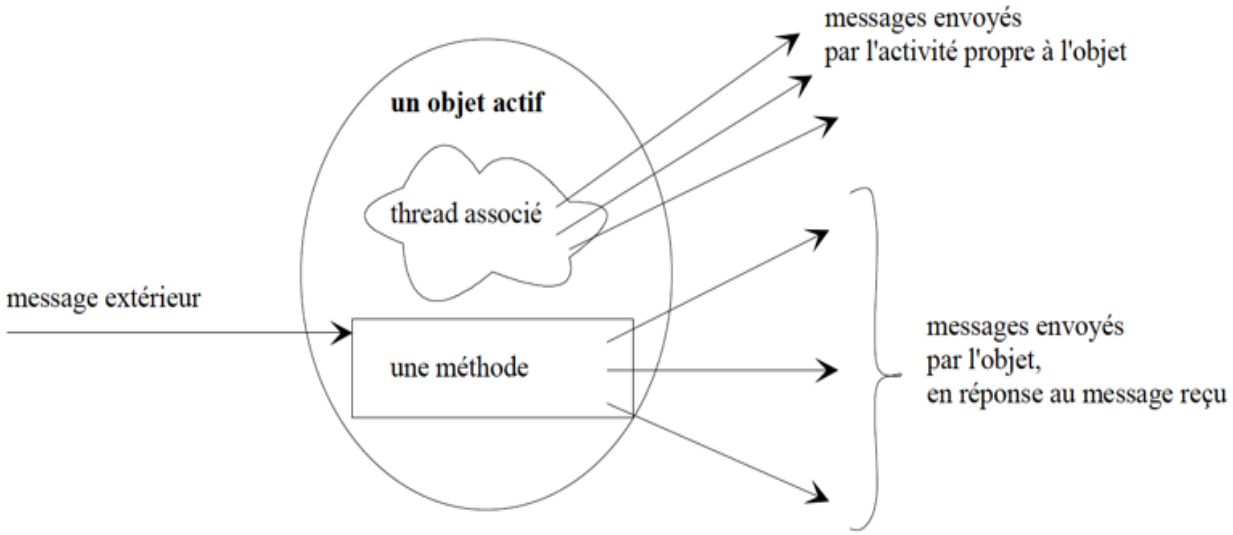
- Attendre la terminaison d'un thread: `join()`

  Ce message, envoyé à un thread `t`, bloque le thread courant en attendant la terminaison du thread `t`

  Le thread courant est réveillé aussitôt que le thread `t` passe à l'état **Terminé**

3. Réaliser des objets actifs en Java

Dans la méthodologie Objet, Booch définit le concept des **objets actifs:**



Un objet actif :

- Peut changer d'état par lui-même,
- Peut générer par lui-même des messages vers d'autres objets.

Deux méthodes pour implémenter un objet actif en Java..

Méthode 1

L'objet actif **est-un** thread, c'est-à-dire une instance d'une sous-classe de `Thread`.

Deux méthodes pour implémenter un objet actif en Java

Méthode 1

L'objet actif **est-un** thread, c'est-à-dire une instance d'une sous-classe de `Thread`.

Méthode 2

- L'objet actif est associé à un thread,
- généralement déclaré sous la forme d'une variable d'instance de l'objet actif.

Deux méthodes pour implémenter un objet actif en Java

Méthode 1

L'objet actif **est-un** thread, c'est-à-dire une instance d'une sous-classe de `Thread`.

Méthode 2

- L'objet actif est associé à un thread,
- généralement déclaré sous la forme d'une variable d'instance de l'objet actif.

La 2ème méthode est meilleure ☺

- elle est plus naturelle du point de vue de la méthodologie objet,
- l'objet actif peut hériter d'autre chose que de la classe `Thread` (pas d'héritage multiple).

Méthode 1:
par dérivation
de `Thread`

```
class Bavard extends Thread{
     private String texte;
     private int pause;


     public Bavard (String txt, int pause) {
          texte = txt; this.pause = pause;
     }
     public void run () {
          while (true) {
               try {Thread.sleep(pause); }
               catch (InterruptedException e) {}
               System.out.println(texte);
          }
     }
}


public class Bavards {
     public static void main (String arg[]) {
          new Bavard ("ttt", 30).start();
          new Bavard ("xxxxx", 100).start();
     }
}
```

Méthode 2: implémenter *Runnable*

```
class Bavard implements Runnable{
   private String texte;
   private int pause;

   private Thread activité;

   public Bavard (String txt, int pause) {
          texte = txt; this.pause = pause;

          activité = new Thread (this);

          activité.start();
   }

   public void run () {
      while (true) {
            try {Thread.sleep(pause); }
            catch (InterruptedException e) {}
            System.out.println(texte);
      }
   }

   public void démarrer() {activité.start();}
   public void arrêter() {activité.stop();}
}
```

*La classe "Bavard" s'engage à implémenter la méthode "**run()**"*

Déclaration d'une activité associée

*Construction du thread. "**this**" indique que le thread doit exécuter la méthode "run()" de la classe "Bavard". Le type de ce paramètre doit être de type "Runnable"*

Pour démarrer le thread (ici ou ailleurs)

Méthodes éventuelles, pour contrôler l'activité depuis l'extérieur

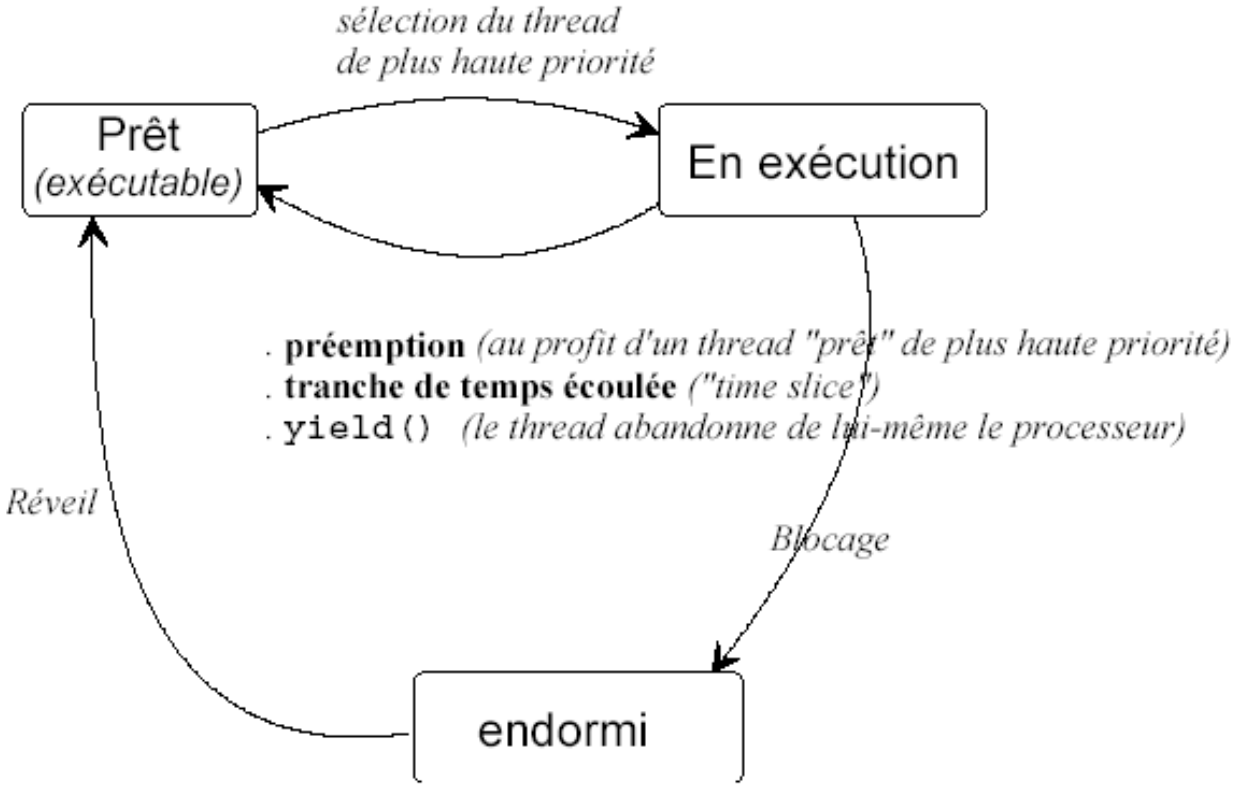
Méthode 3: Avec une **classe interne anonyme**

```
class Bavard {
    private String texte;
    private int pause;
    private boolean arreter = false;

    public Bavard (String txt, int pause) {
        texte = txt; this.pause = pause;
    }

    public void demarrer() {
        new Thread(){
            public void run () {
                while (!arreter) {
                    try {Thread.sleep(pause); } catch (InterruptedException e) {}
                    System.out.println(texte);
                }
            }
        }.start();
    }

    public void arreter() {arreter = true;}
}
```

Suicide forcé du thread

4. Politique d'allocation du processeur

Scheduling



Priorité des threads

```
unThread.setPriority (priority)
```

- Priorité par défaut ⇨ `Thread.NORM_PRIORITY` **5**
- Priorité minimale ⇨ `Thread.MIN_PRIORITY` **1**
- Priorité maximale ⇨ `Thread.MAX_PRIORITY` **10**

Implémentation des priorités sur les divers OS

La plupart du temps ..

→ Les systèmes ignorent purement et simplement les priorités !

Sur certains..

→ Ne sont distingués que les 3 niveaux

`NORM_PRIORITY`, `MIN_PRIORITY` et `MAX_PRIORITY`

Ne pas se baser sur les priorités!

Le time slicing

LaJVM s'appuie sur l'OS sous-jacent...

- **Windows >** time-slicing
- **SUN-Solaris**: Pas de time-slicing implémenté, pas prévu.
- **Linux**: possible, mais pas par défaut

☺ Solution pour garder une portabilité: utilisation de **`yield()`,**

⇨ méthode qui autorise un thread à abandonner le processeur au profit d'un thread de même priorité.

```java
class Bavard extends Thread {
    private String monSigle;

    public Bavard (int priorité, String monSigle) {
        setPriority (priorité) ;
        this.monSigle = monSigle;
    }

    public void run () {
        for (int cpt = 1 ; cpt <= 3 ; cpt++) {
            System.out.println (monSigle + "  " ) ;
            yield() ;
        }
    }
}

public class LesBavards {
    public static void main (String arg[]) {
        new Bavard (5, "55555").start();
        new Bavard (1, "11111").start();
        for (int cpt = 1 ; cpt <= 3 ; cpt++) {
            System.out.println ("Thread par défaut") ;
        }
    }
}
```

Windows

```
Thread par defaut
55555
Thread par defaut
55555
Thread par defaut
55555
11111
11111
11111
```

Ou

```
Thread par defaut
55555
11111
Thread par defaut
Thread par defaut
11111
55555
11111
55555
```

Ou . .

```
class Bavard extends Thread {
    private String monSigle;

    public Bavard (int priorité, String monSigle) {
        setPriority (priorité) ;
        this.monSigle = monSigle;
    }

    public void run () {
        for (int cpt = 1 ; cpt <= 3 ; cpt++) {
            System.out.println (monSigle + "   " ) ;
            yield() ;
        }
    }
}

public class LesBavards {
    public static void main (String arg[]) {
        new Bavard (5, "55555").start();
        new Bavard (1, "11111").start();
        for (int cpt = 1 ; cpt <= 3 ; cpt++) {
            System.out.println ("Thread par défaut") ;
        }
    }
}
```

MacOs

```
Thread par défaut
Thread par défaut
Thread par défaut
11111
11111
11111
55555
55555
55555
```

Ou

```
Thread par défaut
Thread par défaut
Thread par défaut
55555
55555
55555
11111
11111
11111
```

```
class Bavard extends Thread {
    private String monSigle;

    public Bavard (int priorité, String monSigle) {
        setPriority (priorité) ;
        this.monSigle = monSigle;
    }

    public void run () {
        for (int cpt = 1 ; cpt <= 3 ; cpt++) {
            System.out.println (monSigle + "   " ) ;
            yield() ;
        }
    }
}

public class LesBavards {
    public static void main (String arg[]) {
        new Bavard (5, "55555").start();
        new Bavard (1, "11111").start();
        for (int cpt = 1 ; cpt <= 3 ; cpt++) {
            System.out.println ("Thread par défaut") ;
        }
    }
}
```

MacOs

```
Thread par défaut
Thread par défaut
Thread par défaut
11111
55555
11111
55555
11111
55555
```

Ou

```
Thread par défaut
Thread par défaut
Thread par défaut
55555
11111
55555
11111
55555
11111
```

Note supplémentaire d'implémentation avec Windows

Le risque de famine

Pour compliquer le tout…

La JVM garantit qu'un jour ou l'autre un thread de basse priorité finira par s'exécuter, pour éviter le risque de famine.

Cela signifie que dans la queue des processus **Prêt**, ce n'est pas forcément le thread de plus haute priorité qui prend la main...

En conclusion...

Pour écrire une application Java **portable**:

> Votre programme doit être conçu pour fonctionner dans le pire des cas….

En conclusion...

Pour écrire une application Java **portable**:

Votre programme doit être conçu pour fonctionner dans le pire des cas….

- **Les threads peuvent être interrompus à tout moment !**

  → Utiliser donc le verrou de synchronisation mutuelle (`synchronized`, voir paragraphe suivant).

En conclusion...

Pour écrire une application Java **portable**:

Votre programme doit être conçu pour fonctionner dans le pire des cas….

- **Les threads peuvent être interrompus à tout moment !**

  → Utiliser donc le verrou de synchronisation mutuelle (`synchronized`, voir paragraphe suivant).

- **Les threads peuvent ne jamais être interrompus.**

  → Utiliser des méthodes comme `sleep()` ou `yield()` pour abandonner le processeur quand il le faut.

En conclusion...

Pour écrire une application Java **portable**:

Votre programme doit être conçu pour fonctionner dans le pire des cas….

- **Les threads peuvent être interrompus à tout moment!**

  → Utiliser donc le verrou de synchronisation mutuelle (`synchronized`, voir paragraphe suivant).

- **Les threads peuvent ne jamais être interrompus**

  → Utiliser des méthodes comme `sleep()` ou `yield()` pour abandonner le processeur quand il le faut.

- **Ne pas compter sur les 10 niveaux de priorité à disposition**

En conclusion...

Pour écrire une application Java **portable**:

Votre programme doit être conçu pour fonctionner dans le pire des cas….

•Il faut s'attendre à ce que vos threads soient interrompus à tout moment. Utiliser donc le verrou de synchronisation mutuelle (`synchronized`, voir paragraphe suivant).

•Il faut aussi s'attendre à ce que vos threads ne soient jamais interrompus. Utiliser donc des méthodes comme `sleep()` ou `yield()` pour abandonner le processeur quand il le faut.

•On ne peut pas non plus compter sur les 10 niveaux de priorité à disposition (cf Win32), mais seulement sur 3 niveaux (ou plus, mais à vous de tester...).