Programmation pour le multimédia : une introduction

Programmation graphique avancée et animations

But du chapitre

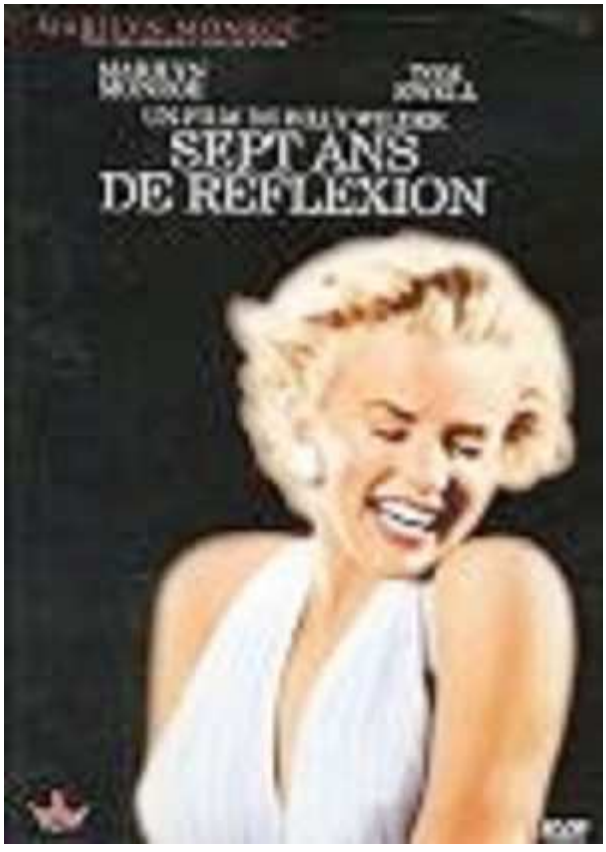
Multimédia = texte, graphique, son (musique, parole), audiovisuel (vidéo).
En informatique car numérisation de données multimédia.
"Donc" traitement de données multimédia avec des outils.

Un exemple (Stéphane Natkin)

Une banque de données multimédia qui gère un catalogue de films. Un utilisateur cherche une œuvre en :

- sifflant au microphone le thème de la musique du film
- scanne ou dessine un croquis représentant une actrice blonde et souriante
- indique sous forme parlée "dans une scène du film, la jupe de l'actrice se soulève lorsqu'elle passe sur une grille d'aération du métro"

Le logiciel lui visionne alors Marylin Monroe dans "Sept ans de réflexion".



Un exemple (Stéphane Natkin) (suite)

Pour faire un tel logiciel, il faut :

- avoir des outils d'analyse de mélodies (traitement du son), de dessin ou d'images, de compréhension de texte parlé
- des systèmes d'indexation et de structuration de données (documents vidéo, sonores, croquis, etc.)
- des interfaces homme machine présentant des systèmes d'acquisition sonores (périphériques), de saisie de dessin, de visualisation vidéo

Remarques

1°) Certains points sont du domaine de la recherche

2°) Pour écrire un tel logiciel, il faut avoir un langage de programmation avec un environnement traitant, dans un cadre unifié, le multimédia.

But du chapitre (suite)

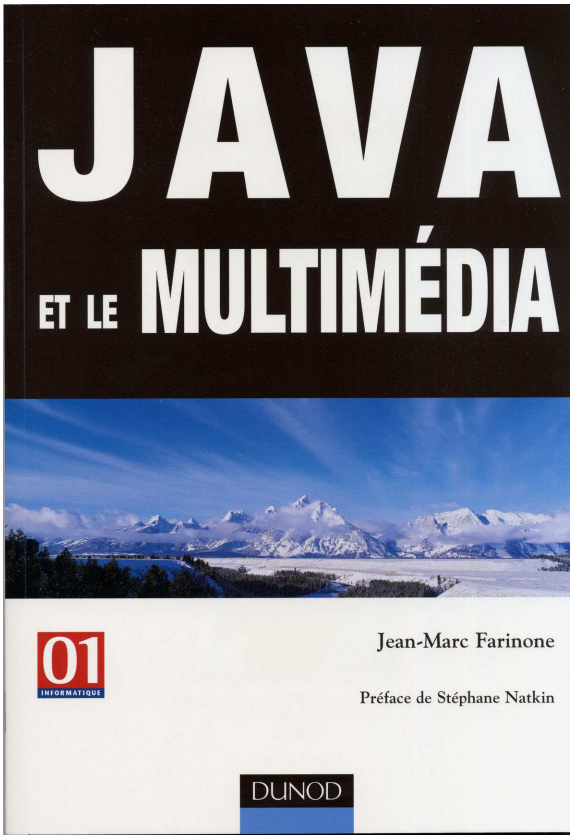
Présenter un environnement de programmation pour le multimédia. Un seul environnement pour traiter :

- les animations,
- la parole
- la 3D
- la vidéo
- la musique et les sons

Une solution : Java

Ce chapitre a pour but d'apprendre mais plutôt revoir Java

Une première bibliographie



Pourquoi programmer ?

Pour mieux maîtriser ce qu'on fait

- Pour faire des choses "plus finement"
- Pour être "libre" ... de récupérer, d'enrichir, de modifier le logiciel
- Pour ne pas être dépendant ... du fournisseur de logiciel, de ses diktats (changements de version, de matériel)
- D'autres personnes peuvent vous aider à reprendre votre programme et vous pouvez reprendre les programmes d'autres personnes et les améliorer.
- phénomène du logiciel libre : Linux, ...

Que faut il pour programmer ?

- Un langage de programmation
- ... Assez riche pour ne pas à avoir à réinventer la roue
- Qui donne les structures de données, les algorithmes, les "couches basses" de nombreux domaines (programmation réseau, IHM, accès au BD, ...)
- et au multimédia

Un langage pour le multimédia : Java

En standard c'est à dire le noyau = `Core` ou la Java Standard Edition (Java SE), propose les APIs de base (`http://java.sun.com/javase/6/docs/api/`)
Sous forme d'extension (qui deviennent souvent intégrées au noyau), des domaines pointus (entre autre le multimédia).

Un environnement gratuit à charger à partir de :
`http://java.sun.com/javase/downloads/index.jsp`
Prendre le **JDK** (pas le JRE)

Premier exercice

Télécharger et installer le JDK Java 6

Une introduction au multimédia : les animations

Les animations sont elles-mêmes un sujet en soi (cf. les grands films d'animation tel Volt, les films des productions Pixar (les indestructibles, Toy story, etc.)
Les programmer en Java nécessite de présenter les notions suivantes :

- comment dessiner
- pouvoir écrire un programme qui puisse faire plusieurs fils d'exécution simultanément (une animation qui s'exécute ne doit pas bloquer d'autres travaux et réciproquement)

Les méthodes graphiques

Rappels

En Java, la classe permettant de faire de dessin est la classe `java.awt.Graphics`. Cette classe possède les méthodes de dessins de formes géométriques, d'affichage d'images, etc.

Qu'est ce qu'un `Graphics` ?

Un objet `Graphics` est créé par la machine virtuelle Java (JVM) pendant l'exécution du programme. Cet objet contient et décrit tout ce qu'il faut avoir pour pouvoir dessiner ("boites de crayons de couleurs", les divers "pots de peinture", les valises de polices de caractères, les règles, compas pour dessiner des droites et cercles, ...) ainsi que la "toile" de dessin sur laquelle on va dessiner. Cette toile correspond à la partie qui était masquée et qui doit être redessinée.

On peut parfois récupérer le `Graphics` associé à un `Component` par la méthode `getGraphics()` de la classe `Component`.

En général cet objet est l'argument de la méthode `paint()`. Cet argument a été construit par la JVM.

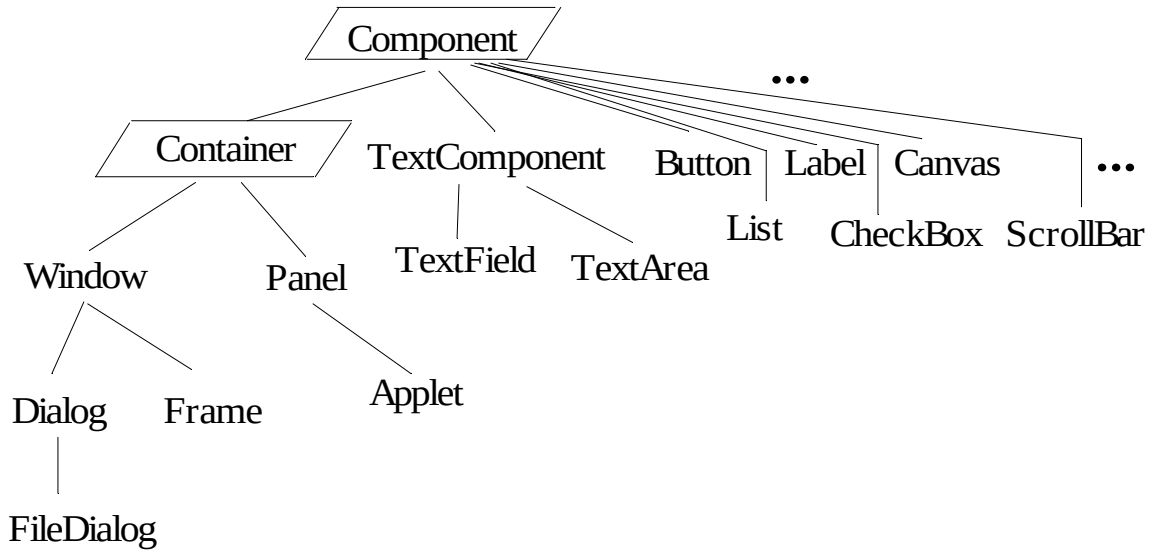
exemple :

```
public void paint(Graphics g) {
  g.drawArc(20, 20, 60, 60, 90, 180);
  g.fillArc(120, 20, 60, 60, 90, 180);
}
```

Les méthodes graphiques

`repaint()`, `update(Graphics g)`, `paint(Graphics g)`

Ces méthodes sont définies dans la classe `Component` (donc existe dans tout composant graphique : héritage).
rappel :



`repaint()` est une méthode de la classe `Component` gérée par "la thread AWT". Le code de cette méthode appelle, dès que cela est possible, la méthode `update(Graphics g)`.
La méthode `update(Graphics g)` de la classe `Component` appelle la méthode `paint(Graphics g)`.

Les méthodes graphiques (suite)

`repaint()`, `update(Graphics g)`, `paint(Graphics g)` de `Component`

`update(Graphics g)` de la classe `Component` efface le composant (le redessine avec sa couleur de fond), puis appelle `paint(Graphics g)`.

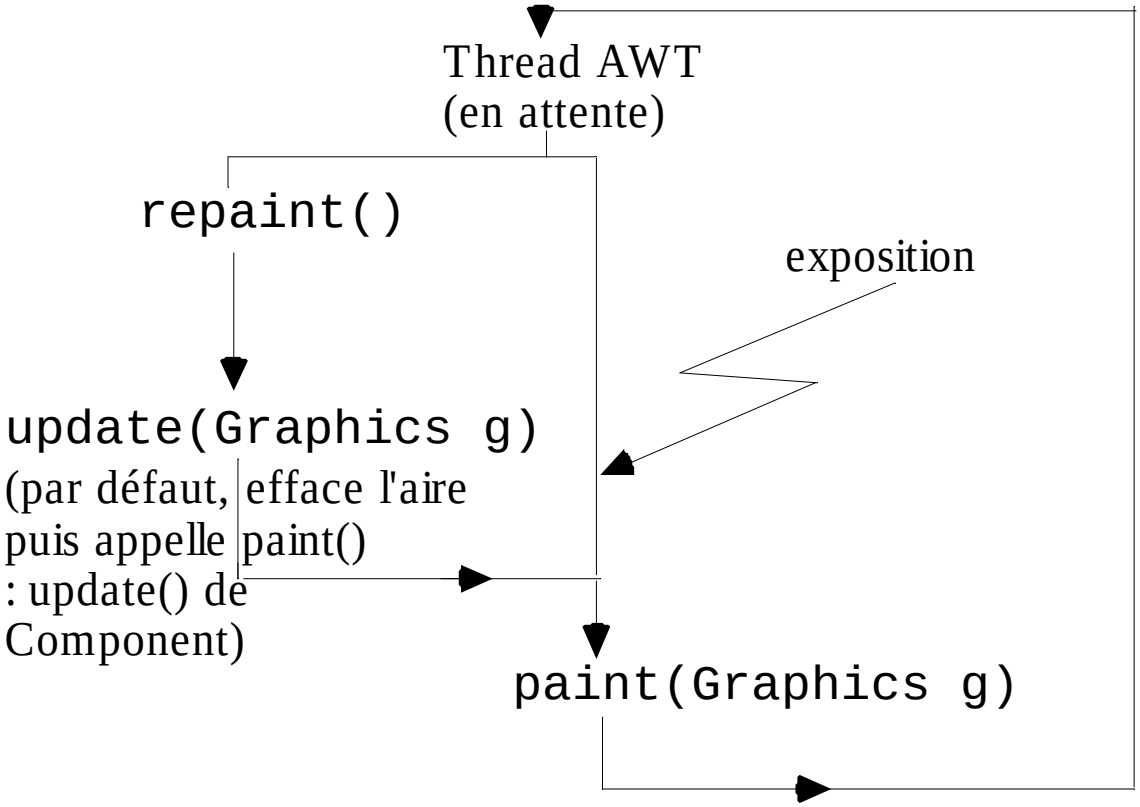
D'ailleurs `update(Graphics g)` de la classe `Component` est :

```
public void update(Graphics g) {
  g.setColor(getBackground());
  g.fillRect(0, 0, width, height);
  g.setColor(getForeground());
  paint(g);
}
```

Le `Graphics` repéré par la référence `g` des méthodes `update()` et `paint()` a été construit par la thread AWT.

Les méthodes graphiques (suite)

`repaint()`, `update()`, `paint()`



Lors d'un événement d'exposition, `paint()` est lancé sur la partie du composant graphique qui doit être redessiné : zone de clipping.

Coder les animations

Nécessite une boucle d'animations.
Une solution :

⇨ pouvoir lancer plusieurs taches (unité d'exécution) ou threads

Rappels sur les threads
Définition

Une thread (appelée aussi processus léger, tâche, fils d'exécution ou activité) est une suite d'instructions à l'intérieur d'un processus.
Les programmes qui utilisent plusieurs threads sont dits multithreadés.

Syntaxe

Les threads peuvent être créés comme instance d'une classe dérivée de la classe `Thread`. Elles sont lancées par la méthode `start()`, qui demande à l'ordonanceur de thread de lancer la méthode `run()` de la thread. Cette méthode `run()` doit être implantée dans le programme.

Premier exemple

```
class DeuxThreadAsynchrones {
   public static void main(String args[ ]) {
      new UneThread("la thread 1").start();
      new UneThread("la seconde thread").start();
   }
}

class UneThread extends Thread {
   public UneThread(String str) {
       super(str);
   }
public void run() {
   for (int i=0; i<10; i++) {
       System.out.println(i+" "+getName());
       try {sleep((int)(Math.random()*10));}
       catch (InterruptedException e){}
   }
   System.out.println(getName()+" est finie");
 }
}
```

une exécution

```
% java DeuxThreadAsynchrones
0 la thread 1
0 la seconde thread
1 la thread 1
2 la thread 1
1 la seconde thread
3 la thread 1
4 la thread 1
5 la thread 1
6 la thread 1
7 la thread 1
2 la seconde thread
3 la seconde thread
4 la seconde thread
8 la thread 1
5 la seconde thread
9 la thread 1
6 la seconde thread
la thread 1 est finie
7 la seconde thread
8 la seconde thread
9 la seconde thread
la seconde thread est finie
```

Une autre exécution donne des sorties différentes.

Une classe "threadée"

C'est une classe qui implémente une thread.
Syntaxiquement on la construit :
- ou bien comme classe dérivée de la classe `Thread`. Par exemple

```
class MaClasseThread extends Thread {
  •••
}
```

- ou bien comme implémentation de l'interface `Runnable`.

```
class MaClasseThread implements Runnable {
  •••
}
```

Cette dernière solution est la seule possible pour une applet "threadée" puisque Java ne supporte pas l'héritage multiple :

```
public class MonAppletThread extends Applet implements
Runnable {
  •••
}
```

Constructeurs de thread

Il y en a plusieurs. Quand on écrit

```
t1 = new MaClasseThread();
```

le code lancé par `t1.start();` est le code `run()` de la classe threadée `MaClasseThread`.
Si on écrit :

```
t1 = new Thread(monRunnable);
```

le code lancé par `t1.start();` est le code `run()` de l'objet référencé par `monRunnable`. Exemple :

```
class MaClasseThread extends Thread {
  Thread t1;
  public static void main(String args[]) {
    t1 = new MaClasseThread ();
    t1.start();
  }
  public void run() { •••
    // ce code est lancé
  }
}
```

et

```
class MonApplet extends Applet implements Runnable {
  public void init() {
    Thread t1 = new Thread(this);
    t1.start();
  }
  public void run() { •••
    // ce code est lancé
  }
}
```

le multithreading dans les applets

Il est bien de prévoir dans une applet, l'arrêt de traitement "désagréable" (animation ou musique intempestive) par clics souris.

Méthodes `stop()` et `start()` de la classe `java.applet.Applet`

Quand un navigateur est iconifié ou qu'il change de page, la méthode `stop()` de l'applet contenue dans cette page est lancée. Celle ci doit libérer les ressources entre autre le processeur.
Si aucune thread n'a été implantée dans l'applet, le processeur est libéré.
Sinon il faut (sauf bonne raison) arrêter les threads en écrivant :

```
public void stop() {
  if (lathread != null) {
     lathread.interrupt();
  }
}
```

C'est la thread d'animation elle-même qui se charge de s'arrêter (`stop()` des threads est dépréciée). Son code est :

```
public void run() {
    boolean threadAnimationInterrompue = false;
    while ((!threadAnimationInterrompue)) {
    // le code de l'animation
    if (lathread.isInterrupted()) {
      threadAnimationInterrompue = true;
      lathread = null;
    }
}
```

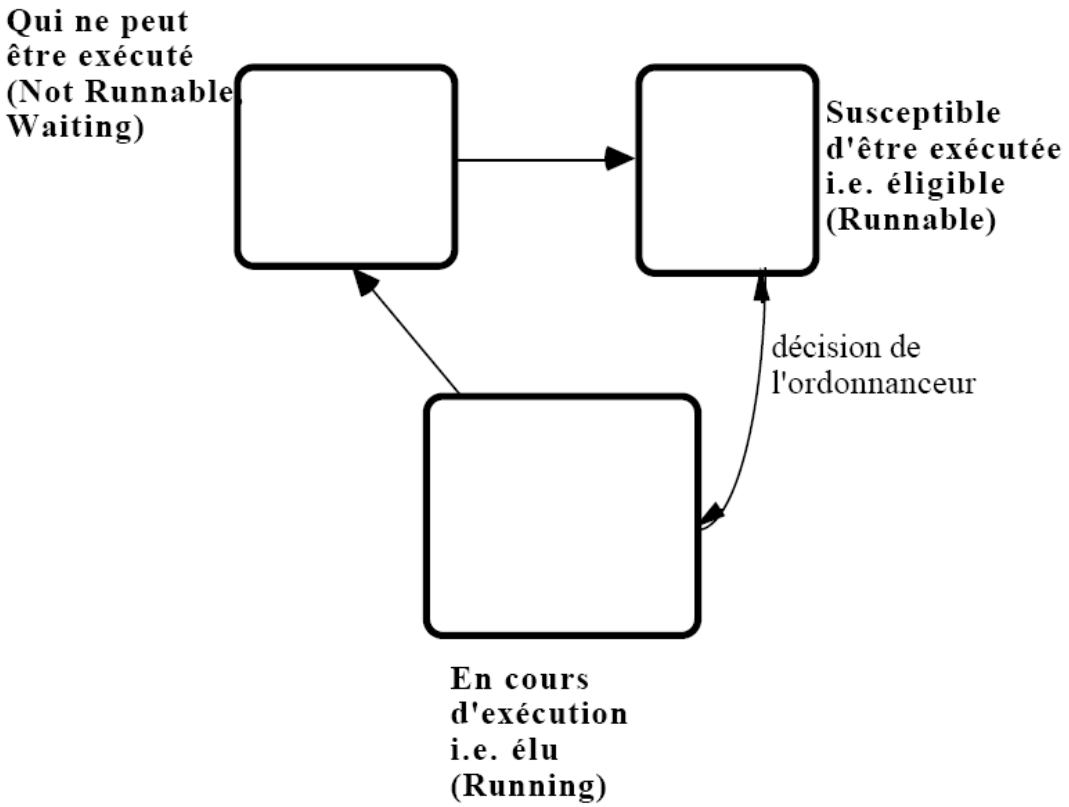
le multithreading dans les applets (suite)

Pour relancer une thread, il faut implanter la méthode `start()` d'une applet par :

```
public void start() {
  if (lathread == null) {
    lathread = new Thread (•••);
    lathread.start(); // ce N'est Pas recursif. OK ?
  }
}
```

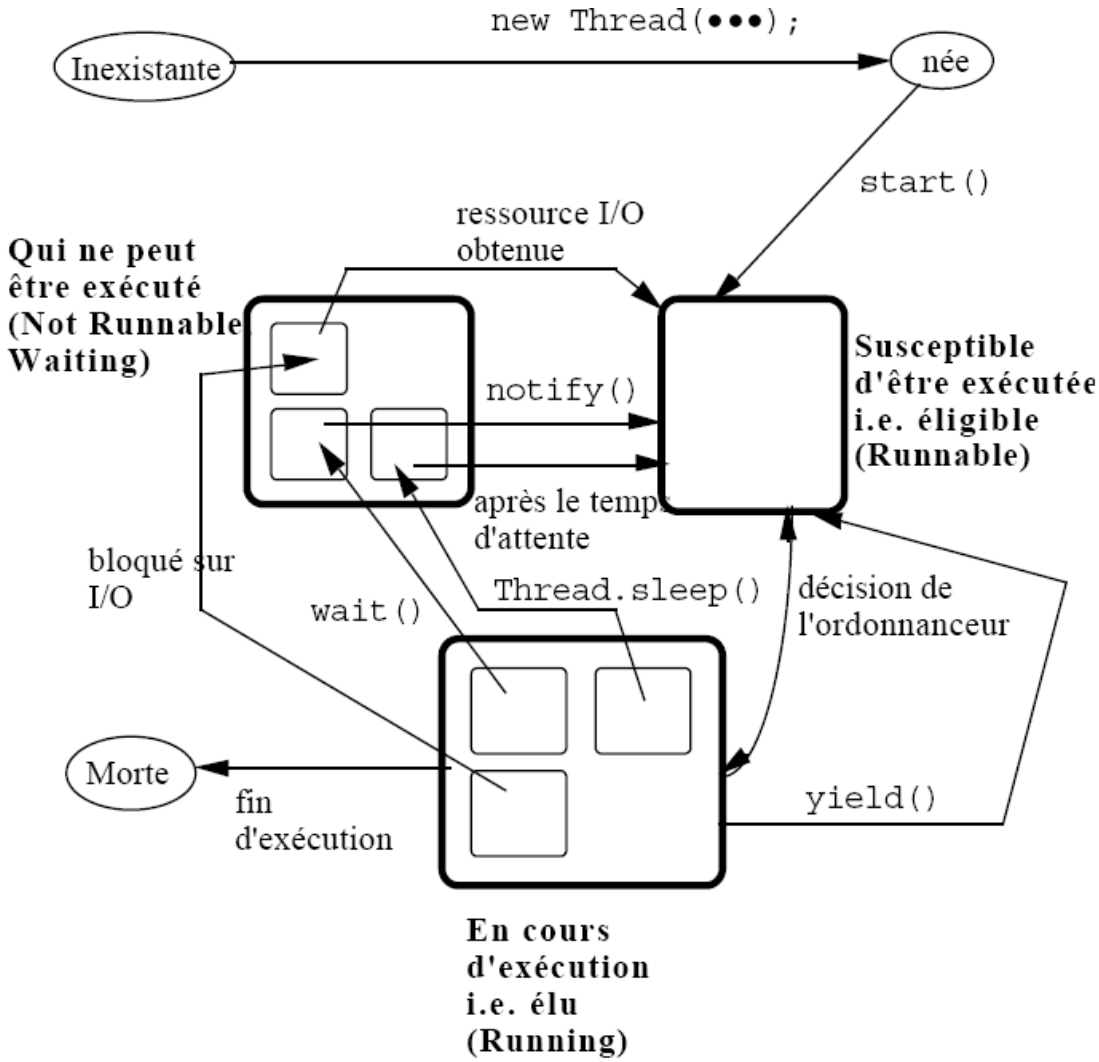
Les états d'une thread

Durant la vie d'une thread, celle-ci passe par trois états fondamentaux classiques (cf. un cours sur la multiprogrammation des processus, des threads, etc.). Ce sont :



Les états d'une thread

Plus précisément en Java 1.2 et suivant on a :



Les états d'une thread (suite)

À l'état "née", une thread n'est qu'un objet vide et aucune ressource système ne lui a été allouée.

On passe de l'état "née" à l'état "Runnable" en lançant la méthode `start()`. Le système lui alloue des ressources, l'indique à l'ordonanceur qui lancera l'exécution de la méthode `run()`.

A tout instant c'est une thread éligible de plus haute priorité qui est en cours d'exécution.

Les états d'une thread (suite)

On entre dans l'état "Not Runnable" suivant 4 cas :
- quelqu'un a lancé la méthode `sleep()`
- la thread a lancé la méthode `wait()` en attente qu'une condition se réalise.
- la thread est en attente d'entrées/sorties

Pour chacune de ces conditions on revient dans l'état "Runnable" par une action spécifique :
- on revient de `sleep()` lorsque le temps d'attente est écoulé
- on revient de `wait()` par `notify()` ou `notifyAll()`.
- bloqué sur une E/S on revient a "Runnable" lorsque la ressource d'E/S est disponible.

On arrive dans l'état "Morte" lorsque l'exécution de `run()` est terminé.

Les Animations

Les programmes (applets) qui suivent ne présentent que des animations. Aussi on pourrait se passer des threads. Mais ce n'est pas le cas général où une animation est (et doit être) réellement un fil d'exécution à part. Aussi on a garder le codage des animations dans une thread.
Exemple 1 voir à
`http://cedric.cnam.fr/~farinone/AFCET/10.html`
On considère l'applet :

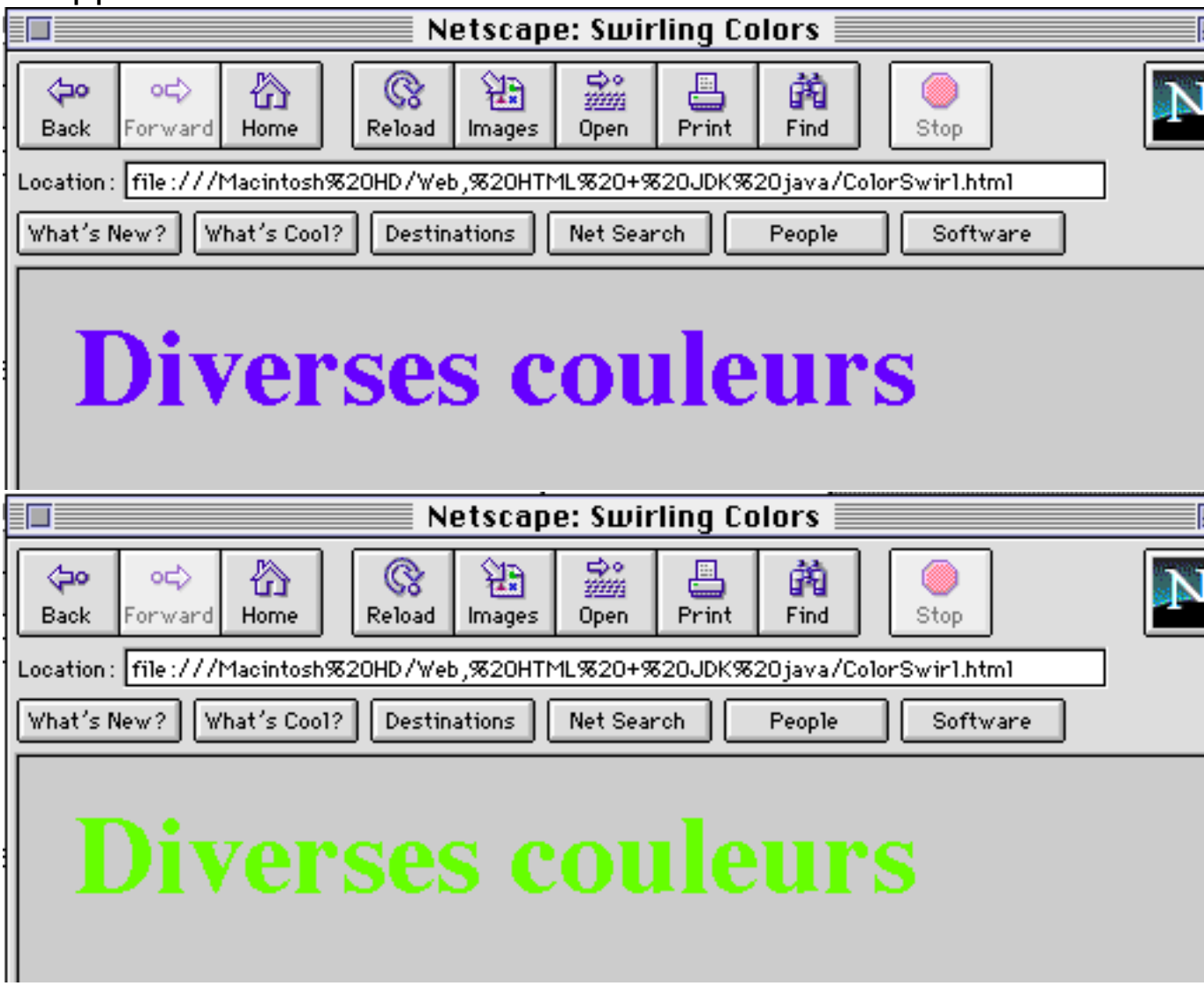
```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;

public class ColorSwirl extends java.applet.Applet
implements Runnable {

  Font f = new Font("TimesRoman",Font.BOLD,48);
  Color colors[] = new Color[50];
 Thread threadAnimation;

  public void start() {
    if (threadAnimation == null) {
      threadAnimation= new Thread(this);
      threadAnimation.start();
    }
  }

  public void stop() {
    threadAnimation = null;
    /* devrait être remplacée car l'appel a stop() des threads
est déprécié */
    /*if (threadAnimation != null) {
      threadAnimation.stop();
    */
  }
```

```
  public void run() {
    // initialise le tableau de couleurs
    float c = 0;
    for (int i = 0; i < colors.length; i++) {
      colors[i] = Color.getHSBColor(c, (float)1.0,
(float)1.0);
      c += .02;
    }
    // la boucle d'animation des couleurs

    int i = 0;
    Thread threadCourante = Thread.currentThread();
    while (threadCourante == threadAnimation) {
      setForeground(colors[i]);
      repaint();
      i++;
      try { Thread.currentThread().sleep(40); }
      catch (InterruptedException e) {
        System.out.println("thread interrompue");
      }
      if (i == (colors.length)) i = 0;
      try {
        Thread.sleep(10);
      } catch (InterruptedException e){
        System.out.println("thread interrompue");
      }
    }
  }
  public void paint(Graphics g) {
    g.setFont(f);
    g.drawString("Diverses couleurs", 15, 50);
  }
}
```

Résultat du programme

applet à charger dans un navigateur contenant :

```
<applet code="ColorSwirl.class" width=400
height=100>
</applet>
```



Une telle animation même si elle fonctionne fait apparaître des tremblements. Ceci est du à la gestion du rafraîchissement en Java et plus précisement au code de la méthode `update(Graphics g)` de la classe `Component`.

C'est la méthode `update(Graphics g)` de la classe `Component` qui est ici utilisé.

Les animations

Très souvent la méthode `update(Graphics g)` donné par Java dans la classe `Component`, pose problème dans les animations (tremblements), car il est intercalé entre 2 dessins une image de couleur unie.

Première solution : redéfinir `update()`

on écrit dans `update()` le seul appel à `paint()`.
Dans le code ci dessus on ajoute simplement :

```
public void update(Graphics g) {
        paint(g);
}
```

et il n'y a plus de tremblements.

Cette solution ne fonctionne pas s'il ne faut redessiner qu'une partie du Composant (optimisation) ou si ce dessin prend beaucoup de temps d'exécution (visualisation de la construction du dessin)

Autre solution le double buffering.

Les tremblements dans les animations (suite)

Seconde solution : le double-buffering

On prépare tout le dessin à afficher à l'extérieur de l'écran (dans un tampon annexe). Lorsque ce second tampon est prêt, on le fait afficher à l'écran. L'écran a ainsi deux buffers (=> double buffering).
Pour cela on utilise :
1°) deux objets un de la classe `Image`, l'autre de la classe `Graphics`. Les deux objets sont associés. Plus précisément :

- on créée d'abord une image hors écran en précisant sa taille à l'aide de la méthode `public Image createImage(int width, int height)`. En général c'est la taille du composant à dessiner
- puis on lui associe un objet `Graphics` par `public abstract Graphics getGraphics()` de `Image`.

2°) les dessins se font dans l'instance `Graphics`.
3°) quand tout est dessiné, on associe l'image au contexte graphique du composant.

Par exemple
Le corps de `update()` doit être :

```
public void update(Graphics g) {
        paint(g);
}
```

Les tremblements dans les animations (suite)

Seconde solution : le double-buffering

Syntaxe

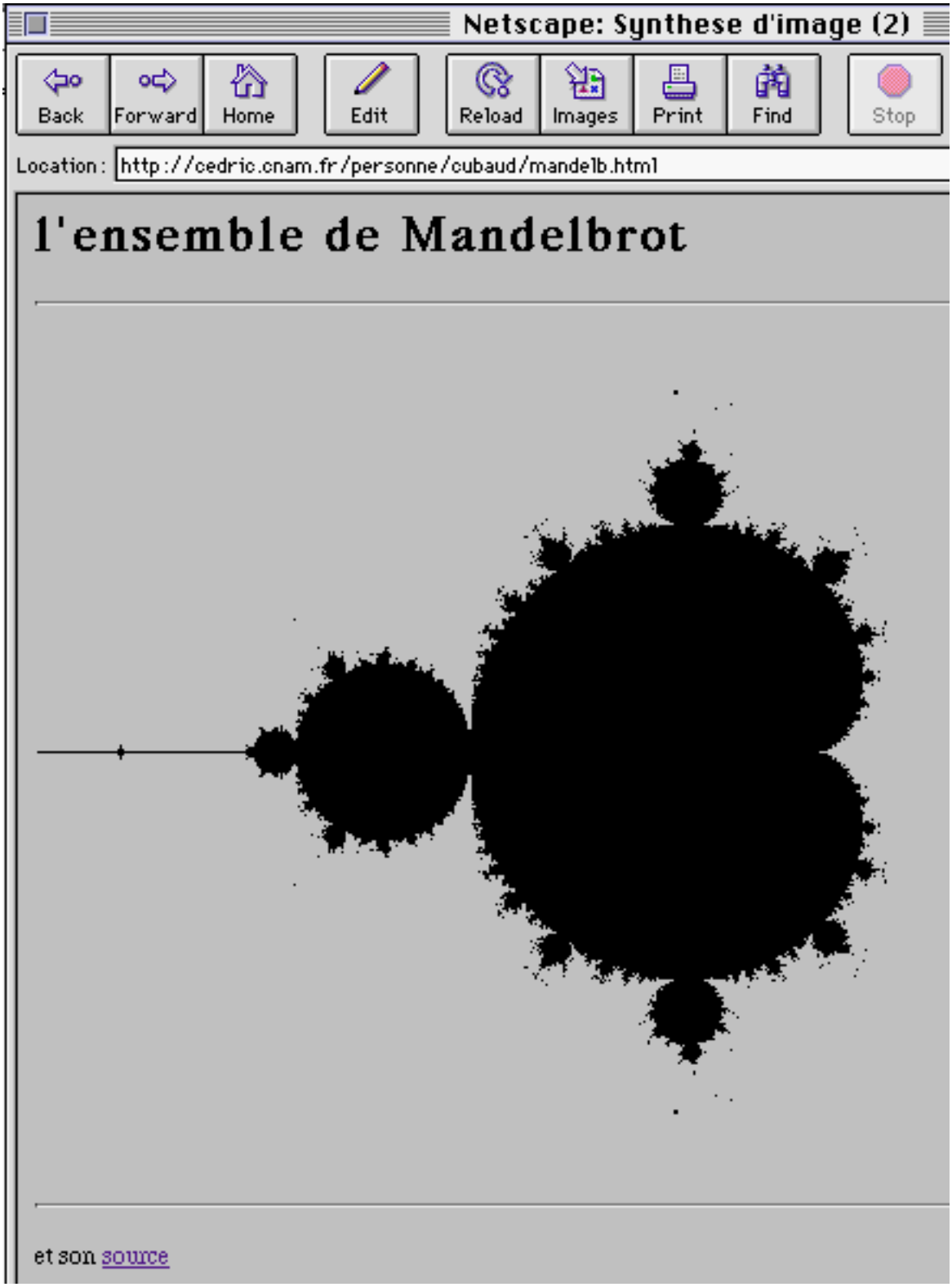
1°) les initialisations sont :

```
Image bufImg;
Graphics bufgc;

public void init() {
        bufImg = createImage(this.getSize().width,
        this.getSize().height);
        bufgc = bufImg.getGraphics();
}
```

2°) et 3°) les dessins sont faits dans le buffer `Graphics` qu'on finit par associer à l'écran. Par exemple:

```
public void paint(Graphics g) {
        bufgc.setColor(Color.Black);
        bufgc.fillOval(20, 60, 100, 100);
        ...
        // paint() doit obligatoirement se terminer par :
        g.drawImage(bufImg, 0, 0, this);
}
```

Double buffering, exemple : Pierre Cubaud



```
import java.awt.*;
import java.applet.Applet;

public class mandelb extends Applet
{
 int haut=400;
 int larg=400;
 int incligne=1;
 int inccolonne=1;

 double x1= -2; //-0.67166;
 double x2= 0.5; //-0.44953;
 double y1= -1.25; //0.49216;
 double y2= 1.25; //0.71429;
 double limite= 50;
 double incx= (x2-x1)/larg;
 double incy= (y2-y1)/haut;

 Image ofbuff;
 Graphics ofg,ong;
 boolean premiere_fois=true;

public mandelb()
{
  resize(haut,larg);
  repaint();
}
```

```
public void paint(Graphics g)
{
  int ligne,colonne,compt;
  double p0,q0,module,x,y,aux;

 if (premiere_fois)
 {
  ofbuff=createImage(larg,haut);
  ofg=ofbuff.getGraphics();
  ofg.setColor(Color.black);
  colonne=0;
  while (colonne<=larg)
  {
   p0=x1+colonne*incx;
   ligne=0;
   while (ligne <= (haut/2))
   {
    q0=y1+ligne*incy;
    x=0;y=0;compt=1;module=0;
    while ((compt<=limite)&&(module<4.0))
    {
     aux=x;
     x=x*x-y*y+p0;
     y=2*y*aux+q0;
     module=x*x+y*y;
     compt++;
    }
    if (module<4.0)
    {
     ofg.drawLine(colonne,ligne,colonne,ligne);
     ofg.drawLine(colonne,haut-ligne,colonne,haut-
ligne);
    }
    ligne+=incligne;
   }
   colonne+=inccolonne;
  }
```

```
 premiere_fois=false;
}
g.drawImage(ofbuff,0,0,null);
}
}
```

Les tremblements dans les animations (suite)

Optimisation : préciser dans `update()` la zone de cliping

zone de cliping = zone sensible de dessin (i.e. à l'extérieur rien n'est redessiné). On écrit alors :

```
public void update(Graphics g) {
  g.clipRect(x1, y1, x2, y2);
  paint(g);
}
```

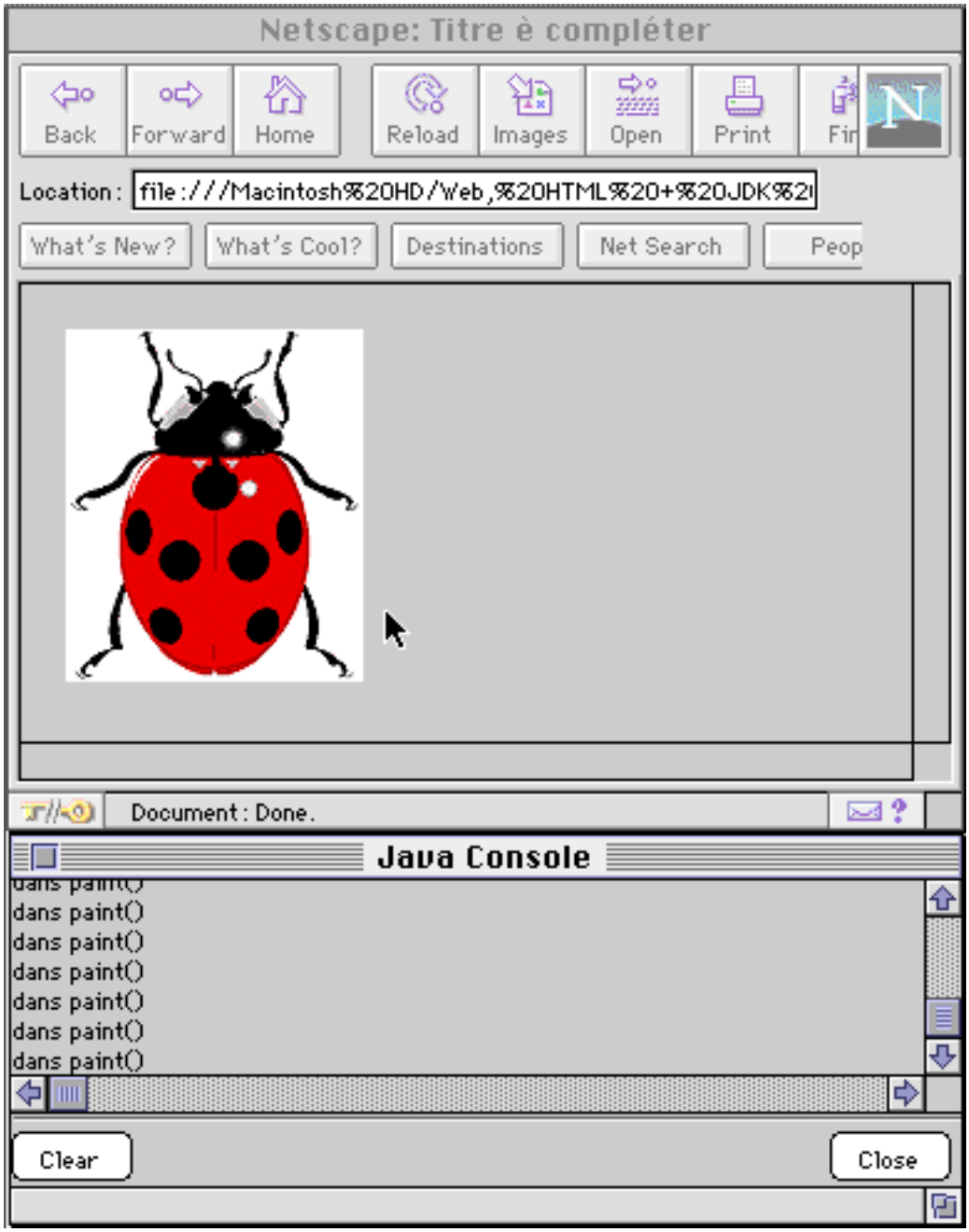
Asynchronisme de `drawImage()`

On considère l'applet Java :

```
import java.awt.Graphics;
import java.awt.Image;

public class LadyBug extends java.applet.Applet {
Image bugimg;
  public void init() {
    bugimg = getImage(getCodeBase(),
       "images/ladybug.gif");
}

public void paint(Graphics g) {
    g.drawImage(bugimg,10,10,this);
    System.out.println("dans paint()");
}
}
```

Au moment du `getImage()` l'image est "repérée" mais pas chargée. Elle l'est réellement lors du `drawImage()`. Ce chargement est effectué dans une thread et `drawImage()` est une méthode asynchrone (i.e. non bloquante).
Cette méthode rend la main à son appelant ce qui explique qu'on ait besoin de passer (par `this`) le composant dans lequel `drawImage()` doit dessiné. Cette thread sera appelé régulièrement par lancement de `repaint()` tant que le dessin n'a pas été entièrement fait. Ceci explique les nombreux passages dans `paint()`.

Asynchronisme de `drawImage()`(suite)



La classe `MediaTracker`

Elle permet de gérer l'asynchronisme de `drawImage()`. Elle offre des méthodes indiquant si un ensemble d'images a été entièrement chargé.

```
import java.awt.*;
import java.applet.*;
public class testMediaTracker extends Applet {
private Image bugimg;
static final private int numero = 0;
private MediaTracker tracker;

  public void init() {
    tracker = new MediaTracker(this);
    bugimg = getImage(getCodeBase(),
"images/ladybug.gif");
    tracker.addImage(bugimg, numero);
    try {
        tracker.waitForAll();
    } catch (InterruptedException e) {
    System.out.println("Pb dans le MediaTracker");
    }
}

public void paint(Graphics g) {
    int resulMT;
    resulMT = tracker.statusID(numero, false);
    if ((resulMT & MediaTracker.COMPLETE) != 0) {
        g.drawImage(bugimg,10,10,this);
        System.out.println("dans paint()");
    }
}
}
```

La classe `MediaTracker` présentation

Cette classe permet de gérer des objets multimédia bien que seuls les fichiers images sont actuellement gérer. Après avoir créer une instance de cette classe par le seul constructeur `MediaTracker(Component)` qui crée un mediatracker pour ce composant, on ajoute les images par

`addImage(Image img, int numero)`

Un numéro peut contrôler un ensemble d'images et indique un ordre de chargement (les petits entiers d'abord) ainsi qu'un identificateur pour cet ensemble.

Le chargement est contrôlé par 4 variables `static` :

`ABORTED` : le chargement a été abandonné.

`COMPLETE` : le chargement s'est bien effectué.

`ERRORED` : erreur au cours du chargement

`LOADING` : chargement en cours.

La classe `MediaTracker` principales méthodes

`addImage(Image img, int num)`
ajoute l'image `img` avec le numéro `num` dans l'instance.

Il existe 2 familles de méthodes pour cette classe : les "check" et les "wait".
Les méthodes "wait" sont bloquantes alors que les "check" ne le sont pas.
Les "wait" attendent que la thread de chargement soit finie pour rendre la main : cela ne signifie pas que le chargement des images ait réellement été effectués (i.e. délai de garde dans les "wait" par exemple, images inexistantes, …).
Les "check" indiquent si les images ont bien été chargées.

La classe `MediaTracker` la famille "wait"

Les méthodes "wait" sont sous contrôle d'`InterruptedException` (donc non masquables) levée lorsqu'une thread a interrompue la thread courante.

`public void waitForAll()`
la thread de chargement de toutes les images mises dans l'instance est lancée.

`public boolean waitForAll(long delai)`
la thread de chargement de toutes les images mises dans l'instance est lancée et se terminera au plus tard après `delai` millisecondes.

`public void waitForID(int num)`
la thread de chargement de l'ensemble des images de numéro `num` est lancé.

`public synchronized boolean waitForID(int num, long delai)`
la thread de chargement de l'ensemble des images de numéro `num` est lancée et se terminera au plus tard après `delai` millisecondes.

Animation : Neko le chat

C'est une animation graphique écrite à l'origine pour macintosh par Kenjo Gotoh en 1989. Elle montre un petit chat (Neko en japonais), qui court de gauche à droite, s'arrête, baille, se gratte l'oreille, dort un moment puis sort en courant vers la droite.

On utilise pour cela les images suivantes :

On peut récupérer les images à :

`http://cedric.cnam.fr/~farinone/AFCET/11.html`

Le code de l'animation Neko

```
import java.awt.*;

public class Neko extends java.applet.Applet
implements Runnable {

  Image nekopics[ ] ;
String nekosrc[] = { "right1.gif", "right2.gif",
"stop.gif","yawn.gif", "scratch1.gif", "scratch2.gif",
"sleep1.gif", "sleep2.gif", "awake.gif" };
  Image currentimg;
  Thread runner;
  int xpos;
  int ypos = 50;
  MediaTracker mt;

  public void init() {
    mt = new MediaTracker(this);
    nekopics= new Image[nekosrc.length];
    for (int i = 0; i < nekosrc.length; i++) {
      nekopics[i] = getImage(getCodeBase(), nekosrc[i]);
      mt.addImage(nekopics[i], 0);
      try {
        mt.waitForAll();
      } catch (InterruptedException e) {
        System.out.println("Pb dans le MediaTracker");
      }
    }
    System.out.println("Fin de init()");
  }
```

```
 public void start() {
   if (runner == null) {
     runner = new Thread(this);
     runner.start();
   }
 }

 public void stop() {
   if (runner != null) {
     runner = null;
   }
 }

 public void run() {
   setBackground(Color.white);
    int milieu = (this.getSize().width) / 2;
    System.out.println("milieu = " + milieu);

   // court de gauche a droite
   flipflop(0, milieu, 150, 1, nekopics[0], nekopics[1]);

   // s'arrete
   flipflop(milieu, milieu, 1000, 1, nekopics[2],
nekopics[2]);

   // baille
   flipflop(milieu, milieu, 1000, 1, nekopics[3],
nekopics[3]);

   // se gratte 4 fois
   flipflop(milieu, milieu, 150, 4, nekopics[4],
nekopics[5]);

   // ronfle 5 fois
   flipflop(milieu, milieu, 250, 5, nekopics[6],
nekopics[7]);
```

```
    // se reveille
    flipflop(milieu, milieu, 500, 1, nekopics[8],
nekopics[8]);

    // et part
    flipflop(milieu, this.getSize().width + 10, 150, 1,
nekopics[0], nekopics[1]);
  }

   void flipflop(int start, int end, int timer, int numtimes,
Image img1, Image img2) {
    System.out.println("entree dans flipflop, start = " +
start +" end = " + end);
    for (int j = 0; j < numtimes; j++) {
    for (int i = start; i <= end; i+=10) {
      this.xpos = i;
      // swap images
      if (currentimg == img1)
        currentimg = img2;
      else if (currentimg == img2)
        currentimg = img1;
      else currentimg = img1;

      repaint();
      pause(timer);
      }
    }
  }

  void pause(int time) {
    try { Thread.sleep(time); }
    catch (InterruptedException e) { }
  }
```

```
  public void paint(Graphics g) {
    // currentimg peut être null car la
    // thread AWT peut etre lancée
    // avant la thread d'animation
    if (currentimg != null)
      g.drawImage(currentimg, xpos, ypos,this);
  }
}
```

Les sons dans les applets

Java propose la classe `java.applet.AudioClip` permettant de manipuler les sons.

On peut charger facilement des sons dans une applet grâce aux méthodes

```
public AudioClip getAudioClip(URL url)
```

ou

```
public AudioClip getAudioClip(URL url,
String nom)
```

de la classe `Applet`.

Après avoir récupérer un objet de la classe `AudioClip` on peut utiliser les 3 méthodes de cette classe : `play()`, `loop()`, `stop()`.
Il est conseillé de lancer la méthode `stop()` de la classe `AudioClip` dans la méthode `stop()` de la classe `Applet`.

Programme de Sons

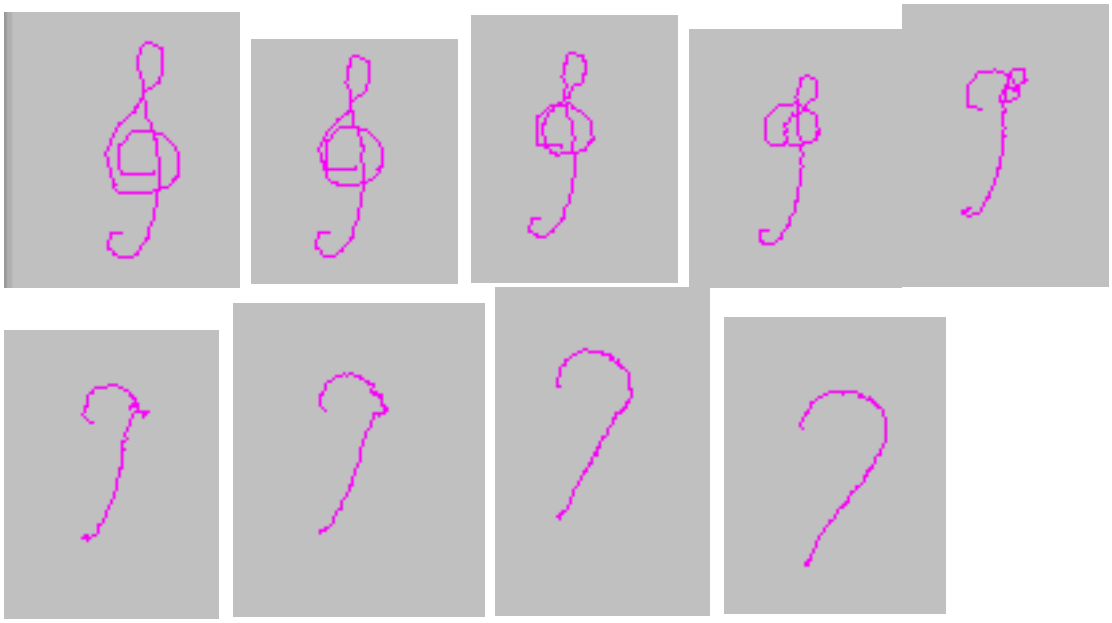
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Musique extends Applet {

  AudioClip bgsound;
  AudioClip beep, gong;
  Thread runner;
  Button btFin, btgong, btbeep;

  public void start() {
    if (bgsound != null) bgsound.loop();
  }

  public void stop() {
      if (bgsound != null) bgsound.stop();
      if (beep != null) beep.stop();
      if (gong != null) gong.stop();
  }
```

```
  public void init() {
    btFin = new Button("Cliquez ici pour arreter la
Musique");
    add(btFin);
    btgong = new Button("lancer gong");
    add(btgong);
    btbeep = new Button("beep ...");
    add(btbeep);
    bgsound = getAudioClip(getCodeBase(),
"audio/loop.au");
    beep = getAudioClip(getCodeBase(),
"audio/beep.au");
    gong = getAudioClip(getCodeBase(),
"audio/gong.au");
    btFin.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        if (bgsound != null) bgsound.stop();
        if (beep != null) beep.stop();
        if (gong != null) gong.stop();
      }
    });
    btgong.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        if (gong != null) gong.loop();
      }
    });
    btbeep.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        if (beep != null) beep.play();
      }
    });
  }
  public void paint(Graphics g) {
    g.drawString("Entendez vous ces musiques ?", 10,
90);
  }
}
```

Morphing 2D en Java

Une applet écrite par Pat Niemeyer (`pat@pat.net`).

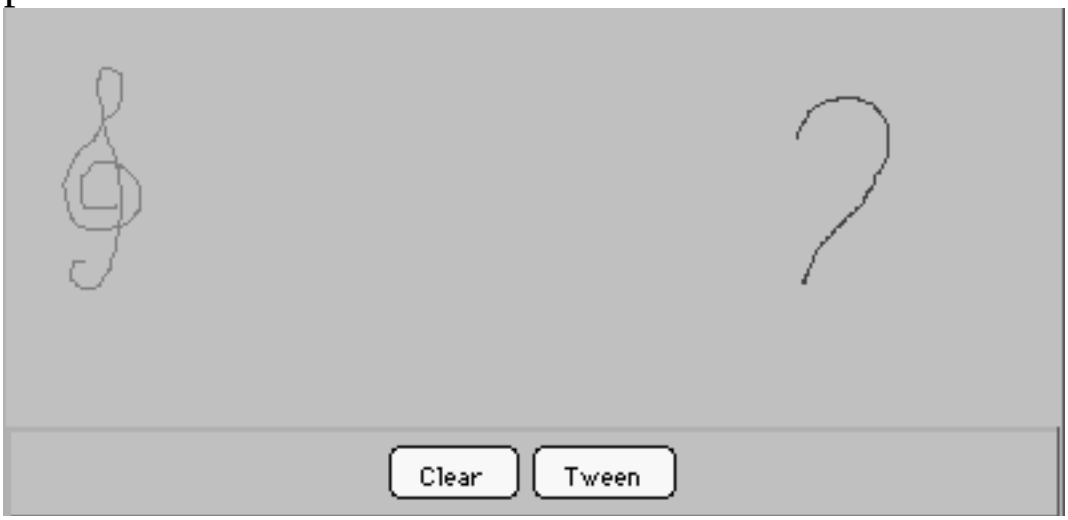
On fait 2 dessins et le programme passe de l'un à l'autre par "morphing".



"... feel free to use it for your class but please don't redistribute it"

code morphing P. Niemeyer

L'applet utilise essentiellement 2 classes dans le paquetage `tween` : la classe `Editor` et la classe `Canvas`. L'applet présente tout d'abord un éditeur de dessin :



dont une partie du code est :

```
package tween;
•••
class Editor extends Panel {
  tween.Canvas feuilleDessin;
  Graphics canvasGr;
  int xpos, ypos, oxpos, oypos;
  private int [][] track = new int [2048][2];
// une courbe a au plus 2048 points
  private int trackEnd = 0;
  private Vector tracks = new Vector();

  Editor() {
    •••
    add( "Center", feuilleDessin = new tween.Canvas() );
    Panel p = •••
    btClear = new Button("Clear");
    btTween = new Button("Tween");
    p.add(btClear);
    p.add(btTween);
    add( "South", p );
```

```
    feuilleDessin.addMouseMotionListener(new
TraiteDeplaceSouris());
    feuilleDessin.addMouseListener(new TraiteClickSouris());

    btClear.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        feuilleDessin.clear();
        tracks = new Vector();
      }
    });

    btTween.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        feuilleDessin.setTracks( tracks );
        feuilleDessin.startTweening();
      }
    });

  private void addPoint( int x, int y ) {
    try {
      track[trackEnd][0] = x;
      track[trackEnd][1] = y;
      trackEnd++;
    } catch ( ArrayIndexOutOfBoundsException e2 ) {
      System.out.println("too many points!!!");
      trackEnd = 0;
    }
  }
}

class TraiteDeplaceSouris extends MouseMotionAdapter {
  public void mouseDragged(MouseEvent e) {
    int x = e.getX(), y = e.getY();
    xpos = x; ypos = y;
    addPoint(x, y );
    if ( canvasGr == null )
      canvasGr = feuilleDessin.getGraphics();
    canvasGr.setColor( Color.red );
 canvasGr.drawLine( oxpos, oypos, xpos, ypos );
 oxpos=xpos; oypos=ypos;
 }}
```

```
class TraiteClickSouris extends MouseAdapter {
  public void mousePressed(MouseEvent e) {
    int x = e.getX(), y = e.getY();
    oxpos = x; oypos = y;
    trackEnd = 0;
    addPoint( x, y );
  }

  public void mouseReleased(MouseEvent e) {
    int [][] tr = new int [ trackEnd ][2];
    for ( int i=0; i< tr.length; i++) {
      tr[i][0] = track[i][0];
      tr[i][1] = track[i][1];
    }
    tracks.addElement( tr );
  }
}
```

morphing P. Niemeyer (suite)

L'éditeur est un code classique de « l'élastique » (rubber band) avec les 3 cas à traiter de bouton souris : appui, déplacer bouton enfoncé, relachement. À chaque déplacement de la souris, le code remplit un tableau `track` (indice `trackEnd` incrémenté) qui va "contenir la courbe tracée par l'utilisateur".
Lorsque le tracé est fini ce tableau est alors mis dans le vecteur `tracks` qui constitue alors le premier élément du vecteur. Le tracé suivant sera le second élément du vecteur, (etc. car on peut mettre plusieurs tracés).
Un tracé est constitué d'un ensemble de points qui ont eux même 2 coordonnées d'où la déclaration de tableau à deux indices de la forme : `track[numeroDuPoint][0 pour x, 1 pour y]`

morphing P. Niemeyer (suite)

La seconde classe importante est `tween.Canvas` :

```
package tween;

import java.awt.*;
import java.util.Vector;
import java.io.*;

class Canvas extends java.awt.Canvas implements Runnable
{
  private Vector tracks = new Vector();
  private Thread runner;
  private Image drawImg;
  private Graphics drawGr;
  private boolean loop = false;

  void setTracks( Vector tracks ) {
    this.tracks = tracks;
  }
  synchronized public void startTweening( ) {
    if ( tracks == null || tracks.size() == 0 )
      return;
    runner = new Thread( this );
    runner.start();
  }

  public void run() {
    do {
      tweenTracks();
    } while ( loop );
  }
```

```
private void tweenTracks() {
  // on récupère le nombre de dessins fait par l'utilisateur
  // (a priori 2)
  int n = tracks.size();

  for (int i=0; i< n-1; i++)  {
    tweenTrack( (int [][])tracks.elementAt(i),
      (int [][])tracks.elementAt(i+1) );
  }
  // on dessine correctement le dessin final
  clearGr();
  drawTrack( (int [][])tracks.elementAt(n-1), Color.magenta );
}
```

```
  /* La méthode qui fait le morphing entre deux courbes
dessinées par l'utilisateur c'est à dire construit les pistes
intermédiaires
  */
  private void tweenTrack(
    int [ ][ ] fromTrack, int [][] toTrack ) {
    int tweens = averageDistance( fromTrack, toTrack )/2;

    /* on dessine " tweens" courbes
    for ( int tweenNo=0; tweenNo < tweens; tweenNo++) {
      /* la tweenNo ieme courbe a len points */
      int len = (int)( fromTrack.length
        + 1.0*(toTrack.length - fromTrack.length)/tweens *
tweenNo );
      /* la tweenNo ieme courbe est dessinée dans le tableau
tweenTrack */
      int [][] tweenTrack = new int [len][2];

      for ( int i = 0; i < tweenTrack.length; i++ ) {
        /* le i ieme point est obtenu comme interpolation des
points correspondant dans les deux courbes dessinés par
l'utilisateur */
        int from = (int)(1.0*fromTrack.length/tweenTrack.length *
i);
        int to = (int)(1.0*toTrack.length/tweenTrack.length * i);
        int x1 = fromTrack[from][0];
        int x2 = toTrack[to][0];
        int y1 = fromTrack[from][1];
        int y2 = toTrack[to][1];

        tweenTrack[i][0] = (int)(x1 + 1.0*(x2-x1)/tweens *
tweenNo);
        tweenTrack[i][1] = (int)(y1 + 1.0*(y2-y1)/tweens *
tweenNo);
      }
      /* on dessine la i ieme courbe */
      clearGr();
      drawTrack( tweenTrack, Color.magenta );
      try {
        Thread.sleep( 1000/24 );
      } catch ( Exception e ) { }
    }}
```

```
  /*
    Ne prend qu'1/10 de la moyenne du nombre de points des 2
courbes et calcule la distance moyenne de ces deux courbes.
  */
  private int averageDistance( int [][] track1, int [][] track2 ) {
    int averages = (track1.length + track2.length)/2/10;
    if ( averages < 3 )
      averages = 3;
    int t = 0;
    for ( int i=0; i< averages; i++ ) {
        // track1[k] représente les coordonnées du point k de track1
        // track1[ i * track1.length / averages ] marche aussi
      int [ ] p1 = track1[ (int)( 1.0 * track1.length / averages * i ) ];
      int [ ] p2 = track2[ (int)( 1.0 * track2.length / averages * i ) ];
      int dx = p2[0] - p1[0];
      int dy = p2[1] - p1[1];
      t += (int)Math.sqrt( dx*dx + dy*dy );
    }
    return t/averages;
  }

  public void update( Graphics g ) {
    paint(g);
  }

  public void paint( Graphics g ) {
    if ( drawImg == null )
      return;
    g.drawImage(drawImg, 0, 0, null);
  }

  public void drawTrack(int [][] track, Color color ) {
    Graphics gr = getOffScreenGraphics();
    gr.setColor( color );

    for ( int i=0; i < track.length-1; i++ )
      gr.drawLine( track[i][0], track[i][1],
        track[i+1][0], track[i+1][1] );

    repaint();
  }
```

```
  private void clearGr() {
    getOffScreenGraphics().clearRect(0, 0, getSize().width,
size().height);
  }

  public Graphics getOffScreenGraphics() {
    if ( drawGr == null ) {
      drawImg = createImage( size().width, getSize().height );
      drawGr = drawImg.getGraphics();
    }
    return drawGr;
  }
```

classe `tween.Canvas` P. Niemeyer

Dans l'éditeur, après appui sur le bouton tween, la méthode `setTracks()` est lancée puis "l'animation morphing" par `startTweening()`.
`setTracks()` positionne le champ `tracks` qui est le vecteur des formes à "joindre" par morphing.
`startTweening()` crée la thread de morphing dont le corps est `tweenTracks()`.

`tweenTracks()` va faire le morphing entre les 2 formes finales en parcourant le vecteur des formes finales (on peut supposer que ce vecteur n'a que 2 éléments !!).

Morphing (P. Niemeyer)

Finalement c'est donc la méthode `private void tweenTrack(int [][] fromTrack, int [][] toTrack)` qui est la clé de tout l'ensemble.
Cette méthode cherche d'abord la distance moyenne entre les 2 courbes. Cette distance donne alors le nombre de courbes intermédiaires à dessiner. Il faut alors construire et dessiner chaque courbe intermédiaire. Pour une courbe donnée (i.e. pour une valeur de `tweenNo`), on construit cette courbe à `len` points. `len` est calculé de sorte à être proche du nombre de point de la courbe finale pour les dernières courbes et vice-versa pour les premières courbes.
On repère les points correspondants aux 2 courbes initiales pour le point à construire, puis on construit ce point ce qui initialise les 2 composantes de `tweenTrack[i]`.

Les dessins sont fait en double buffering et on dessine 24 courbes par secondes.

Second exercice proposé

Voir énoncé à l'URL :
`http://cedric.cnam.fr/~farinone/CCAM/TPAnim/enonceTPanim.html`
On dispose des 2 images :

2 voitures avancent de droite à gauche sur l'image du monde.

Bibliographie

`http://www.javaworld.com/javaworld/jw-03-1996/jw-03-animation.html`

`http://java.sun.com:81/applets/Fractal/1.0.2/example1.html`

Teach yourself Java in 21 days : Laura Lemay, Charles L.Perkins ; ed Sams.net
traduit en français ed S&SM "Le programmeur Java"

Compléments et bibliographie

Java, les animations et le multimédia

On trouve sur Internet certains sites proposant des programmes Java (souvent des applets) qui intègrent, le mouvement, le changement d'images, parfois les sons, etc.

`http://www.boutell.com/baklava/scones.html`
Un site contenant entre autre des petits jeux en ligne

`http://junior.apk.net/~jbarta/idiot/idiot.html`
un bon gag : un bouton poussoir qui fuit quand on essaie de cliquer dessus (mais parfois on y arrive !).

Le site de Karl Hörnell à
`http://www.javaonthebrain.com/` est très beau avec de magnifique applets à
`http://www.javaonthebrain.com/brain.html`
comme le cube de Rubik en Java : magnifique !

Un magnifique site utilisant des systèmes physiques en Java Web Start : `http://sodaplay.com/`
Merci Stéphane P.

`http://javaboutique.internet.com/`
Une belle "boutique" de programmes Java dans beaucoup de domaines y compris le multimédia.

Mon premier exposé sur "Java et le multimédia" :
`http://cedric.cnam.fr/~farinone/AFCET/`