Programmation graphique avancée et animations

Les méthodes graphiques

Rappels

En Java 1.1, la classe permettant de faire de dessin est la classe `java.awt.Graphics`. Seule 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 crée par le "moteur Java". 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 Java machine.

exemple :

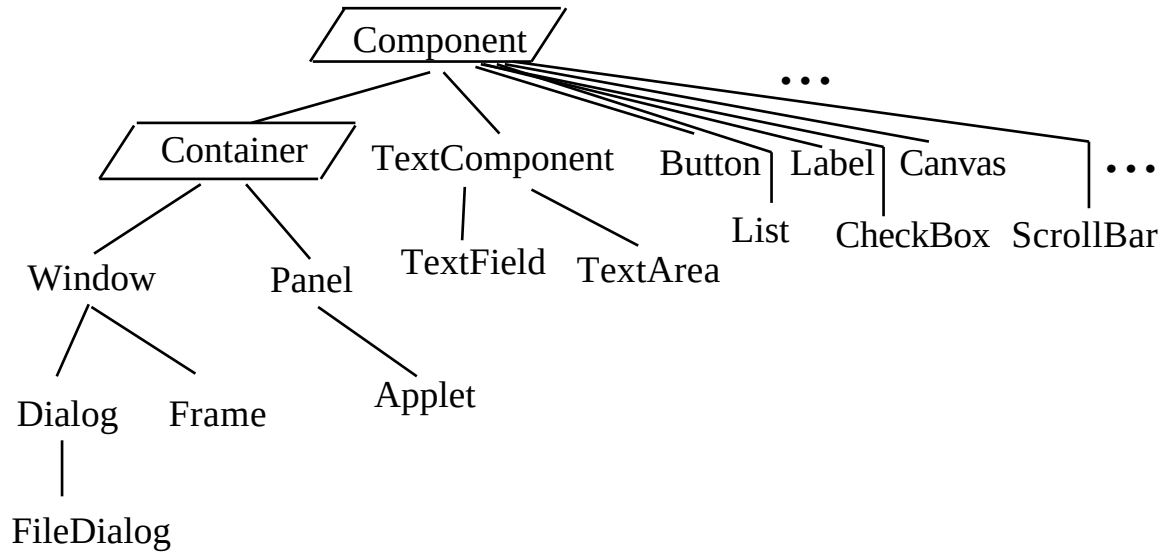
```
public void paint(Graphics g) {
        g.drawArc(20, 20, 60, 60, 90, 180);
        g.fillArc(120, 20, 60, 60, 90, 180);
}
```

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 (qu'il ne faut jamais redéfinir) "système Java" gérée par "la thread AWT" qui appelle, dès que cela est possible, la méthode `update(Graphics g)` qui appelle ensuite `paint(Graphics g)`.

Les méthodes graphiques (suite)

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

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

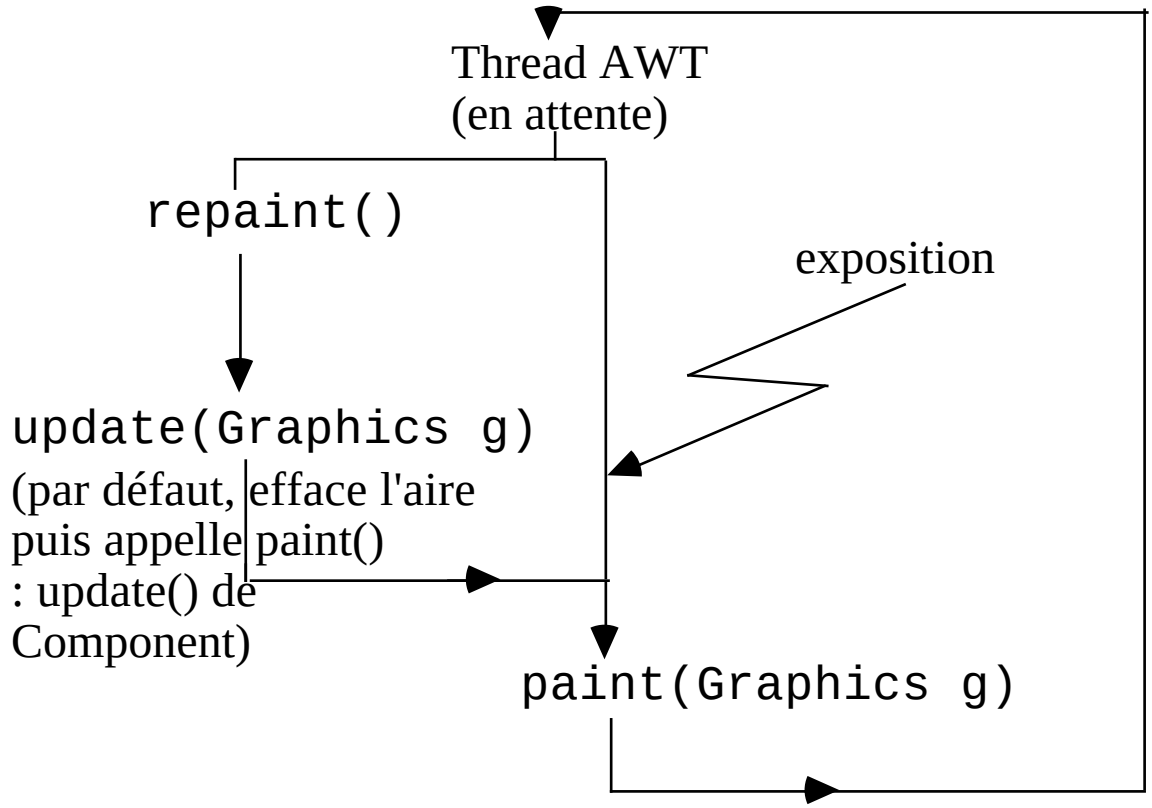
D'ailleurs update(Graphics g) de la classe Component est :

```
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.

Rappels sur les threads

Définition

Une thread (appelée aussi processus léger ou activité) est une suite d'instructions à l'intérieur d'un process.
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 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 Appli extends Thread {
        Thread t1;
        public static void main(String args[]) {
                t1 = new Appli();
                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'arret de traitement "désagréable" (animation ou musique intempestive) par clics souris.

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

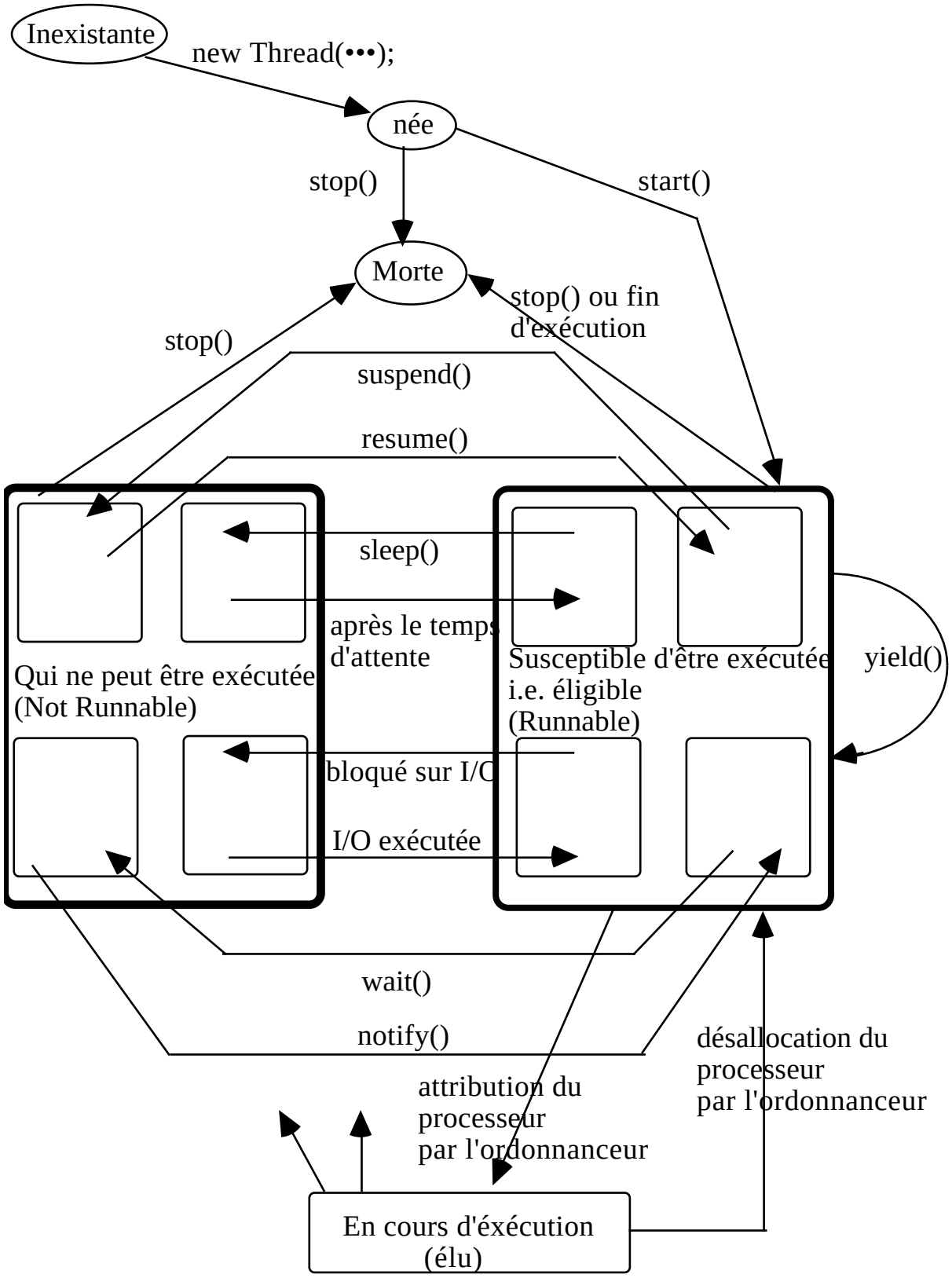
Quand un browser est iconifié ou qu'il change de page, la méthode stop() de l'applet est lancée. Celle ci doit libérer les ressources entre autre le processeur.
Si aucune thread n'a été implantée dans l'applet, le processeur est libéré.
Sinon il faut (sauf bonne raison) arrêter les threads en écrivant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

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

Pour relancer une thread arrétée par stop(), il faut implanter la méthode start() par :

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

Les états d'une thread



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 `suspend()`
- 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 `suspend()` par `resume()`
- on revient de `wait()` par `notify()` ou `notifyAll()`.
- bloqué sur une E/S on revient a "Runnable" lorsque l'opération d'E/S est réalisé.

On arrive dans l'état "Morte" lorsque l'exécution de `run()` est terminée ou bien après un appel à `stop()`.

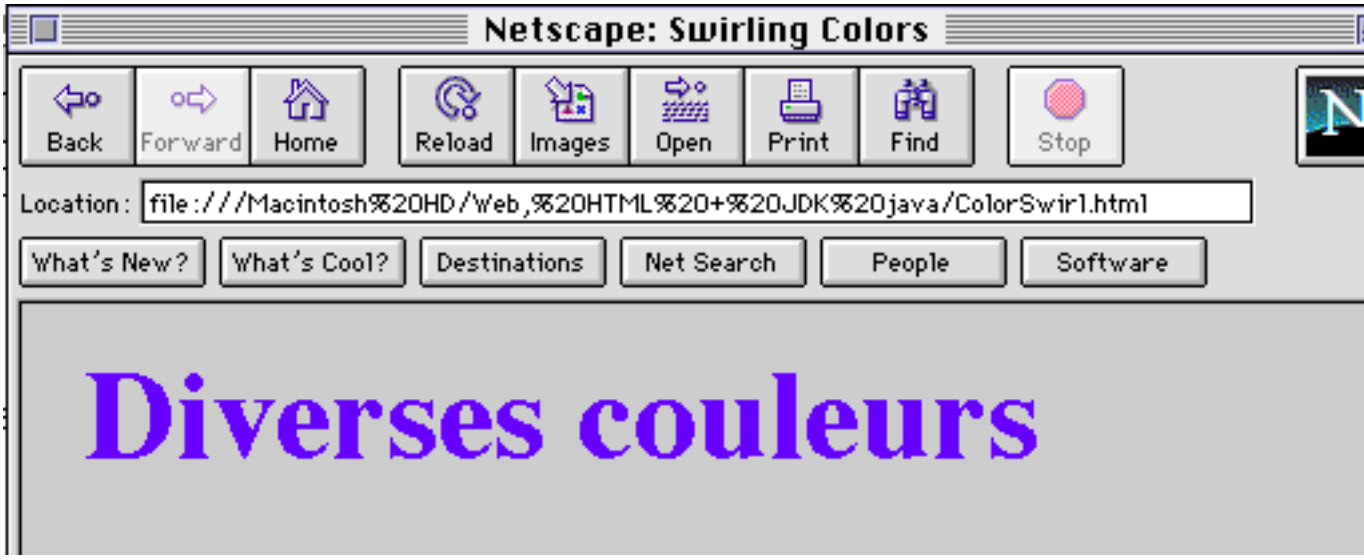
Les Animations

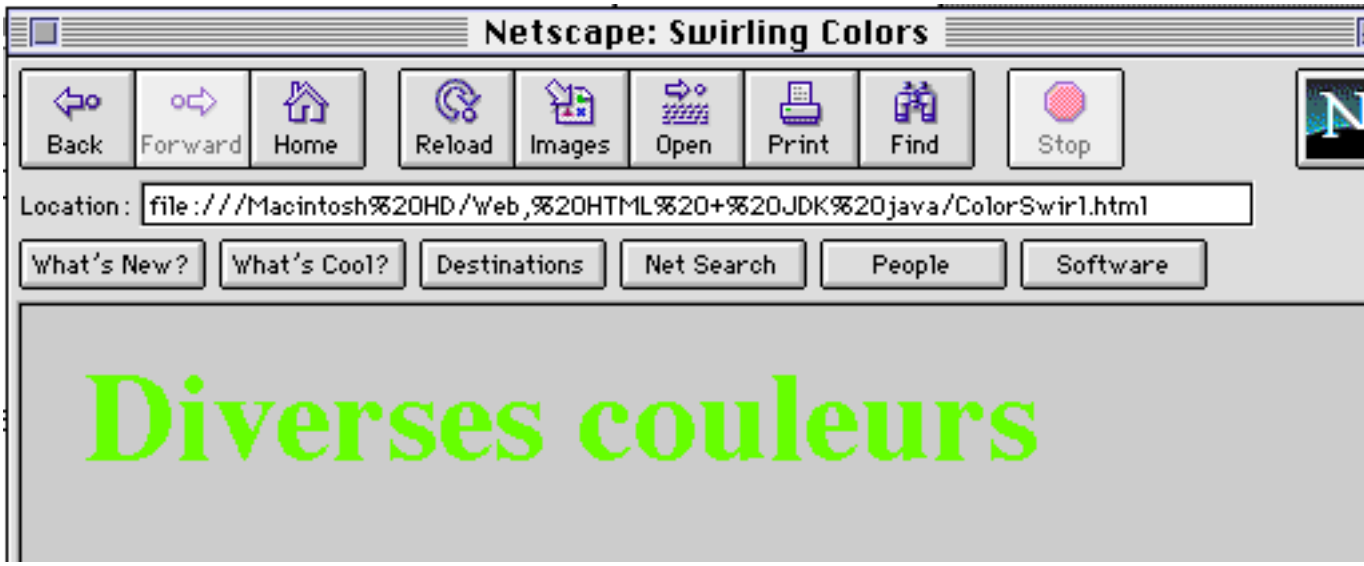
On considère l'applet :

```
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 runThread;

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

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

```
  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;
    }

    // parcourir toutes les couleurs
    int i = 0;
    while (true) {
      setForeground(colors[i]);
      repaint();
      i++;
      try { Thread.currentThread().sleep(50); }
      catch (InterruptedException e) { }
      if (i == (colors.length)) i = 0;
    }
  }

  public void paint(Graphics g) {
    g.setFont(f);
    g.drawString("Diverses couleurs", 15, 50);
  }
}
```

Résultat du programme





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)`.
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 buffer annexe). Lorsque ce second buffer 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`, qu'on initialise dans la méthode `init()`. Les deux objets sont associés.
2°) les dessins se font dans l'instance `Graphics`.
3°) quand tout est dessiné, on associe l'image au contexte graphique de l'applet.

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

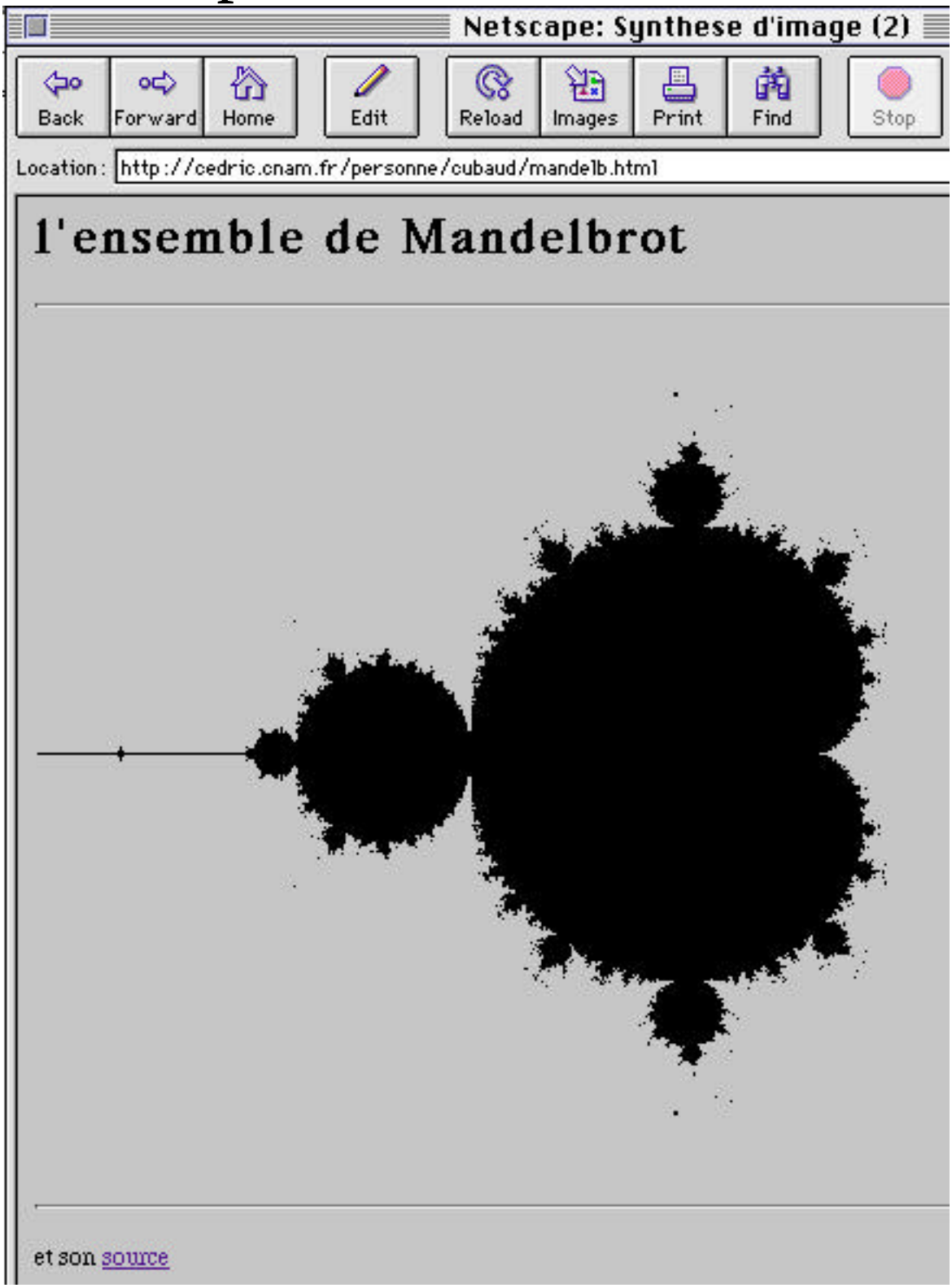
1°) les initialisations sont :

```
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 boolean action(Event e, Object o)
{
Graphics theg;

 if (e.id==Event.MOUSE_UP)
 {
  if (!premiere_fois)
  {
   ong.drawLine(e.x,e.y,e.x,e.y);
```

```
  }
  return true;
 }
 else return false;
}

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

 if (premiere_fois)
 {
  ong=g;
  ong.setColor(Color.black);
  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);
    }
    // pour patienter pdt le calcul
    g.drawLine(colonne,ligne,colonne,ligne);
    ligne+=incligne;
   }
   colonne+=inccolonne;
  }
  premiere_fois=false;
 }
 g.drawImage(ofbuff,0,0,null);
}

public static void main(String[] args)
{
  Applet m= new mandelb();
}

}
```

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);
}
```

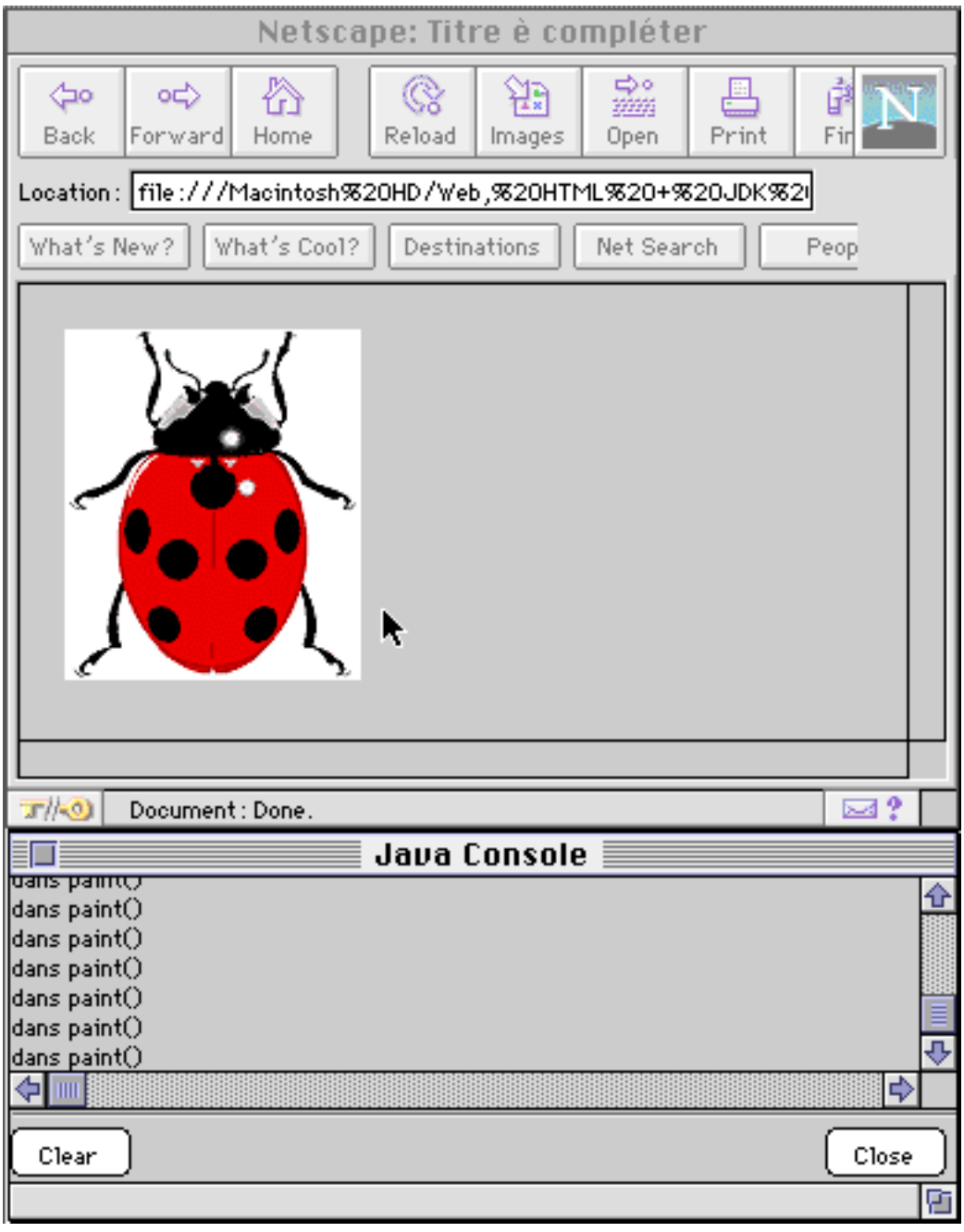
Asynchronisme de drawImage()

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 le passer (par `this`) et `repaint()` est lancé. 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)`
Le numéro peut contrôler un ensemble d'images et indique un ordre de chargement 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).
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 lancé.

`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 lancé et se terminera au plus tard après `delai` millisecondes.

La classe MediaTracker
la famille "check"

`public boolean checkAll()`
vérifie si toutes les images mises dans l'instance sont chargées.

`public boolean checkAll(boolean relance)`
vérifie si toutes les images mises dans l'instance sont chargées. Relance le chargement si `relance` vaut `true`.

`public boolean checkID(int num)`
vérifie si l'ensemble des images repéré par `num` mises dans l'instance sont chargées.

`public boolean checkID(int num, boolean relance)`
vérifie si l'ensemble des images repéré par `num` mises dans l'instance sont chargées. Relance le chargement si `relance` vaut `true`.

La classe MediaTracker le contrôle des erreurs

Il existe des méthodes de contrôle des erreurs de chargement.

`public synchronized Object[] getErrorsAny()`
retourne une liste de médias qui ont posé problème lors du chargement dans l'instance (ou `null` sinon)

`public synchronized Object[] getErrorsID(int id)`
retourne une liste de médias de numéro `id` qui ont posé problème lors du chargement dans l'instance (ou `null` sinon)

`public synchronized boolean isErrorAny()`
renvoie `true` si une des images à provoquer une erreur lors du chargement dans l'instance.

`public synchronized boolean isErrorID(int id)`
renvoie `true` si une des images repérée par `id` à provoquer une erreur lors du chargement dans l'instance.

La classe MediaTracker le contrôle des erreurs (fin)

`public int statusAll(boolean load)`
retourne un masque OR (à comparer avec `MediaTracker.COMPLETE`, ...) indiquant comment s'est passé le chargement. Si `load` vaut `true`, la chargement des images non déjà chargées est relancé.

`public int statusID(int id, boolean load)`
retourne un masque OR (à comparer avec `MediaTracker.COMPLETE`, ...) indiquant comment s'est passé le chargement des images numérotées `id`. Relance le chargement si `load` vaut `true`.

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 :

Le code de l'animation Neko

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

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

  Image nekopics[] = new Image[9];
  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;

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

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

```
  public void run() {
    // chargement des images
    for (int i=0; i < nekopics.length; i++) {
      nekopics[i] = getImage(getCodeBase(),
"images/" + nekosrc[i]);
    }
    setBackground(Color.white);
     int milieu = (this.size().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.size().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) {
    g.drawImage(currentimg, xpos, ypos,this);
  }
}
```

Les sons dans les applets

Java propose la classe `java.applet.AudioClip` permettant de manipuler les sons.
Pour l'instant les seuls fichiers sons disponibles sont les `.au`, un format développé par Sun et jouable sur diverses plateformes.
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.Graphics;
import java.applet.AudioClip;

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

  AudioClip bgsound;
  AudioClip beep;
  Thread runner;

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

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

```
  public void init() {
    bgsound = getAudioClip(getCodeBase(),
"audio/loop.au");
    beep = getAudioClip(getCodeBase(),
"audio/beep.au");
  }

  public void run() {
    if (bgsound != null) bgsound.loop();
    while (runner != null) {
      try { Thread.sleep(5000); }
      catch (InterruptedException e) { }
      if (bgsound != null) beep.play();
    }
  }

  public void paint(Graphics g) {
    g.drawString("Execution de musique....", 10,
10);
  }
}
```