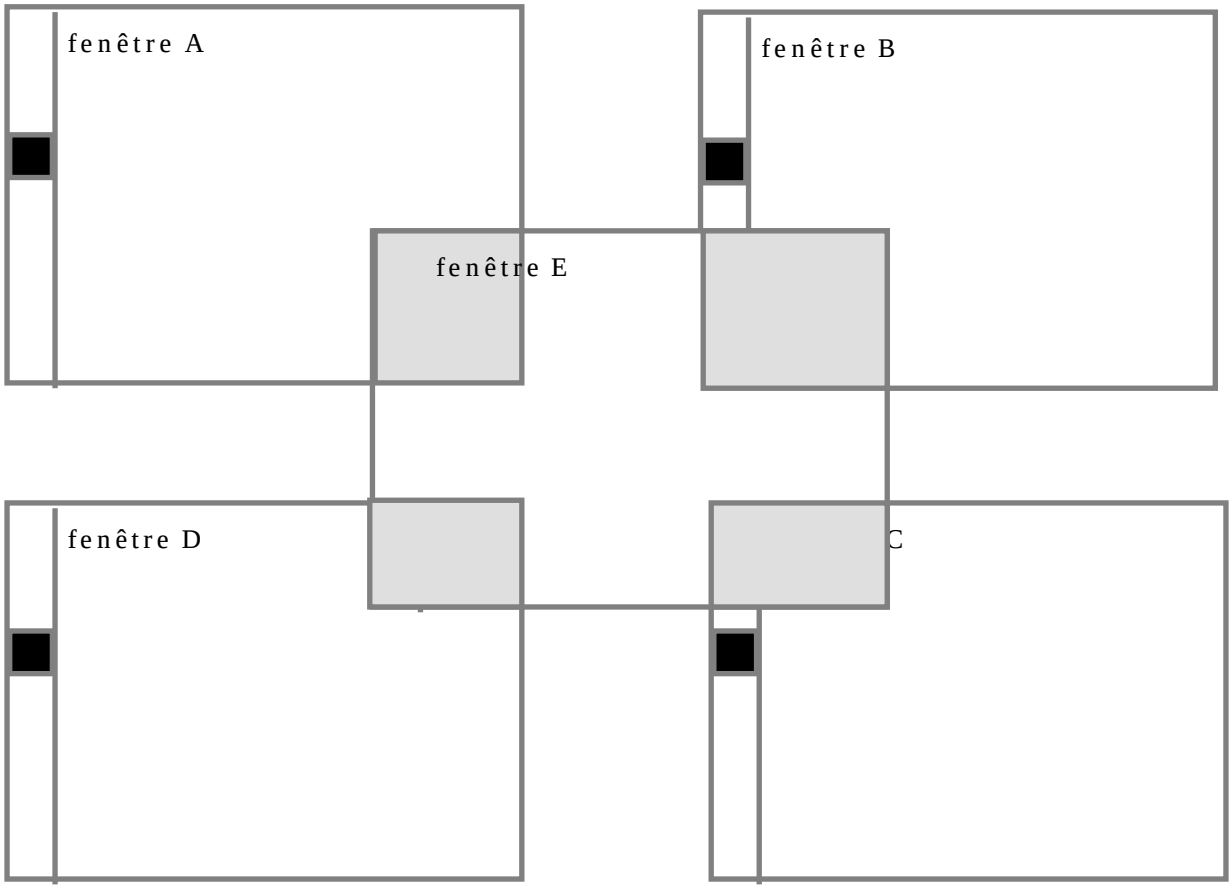
Programmation graphique avancée et animations

Les "expositions"



Lorsque la fenêtre E passe en premier plan, elle reçoît des Expose events afin de redessiner les zones

le Graphics

Lors de la réception d'un événement expose, 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.

Les méthodes graphiques

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

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

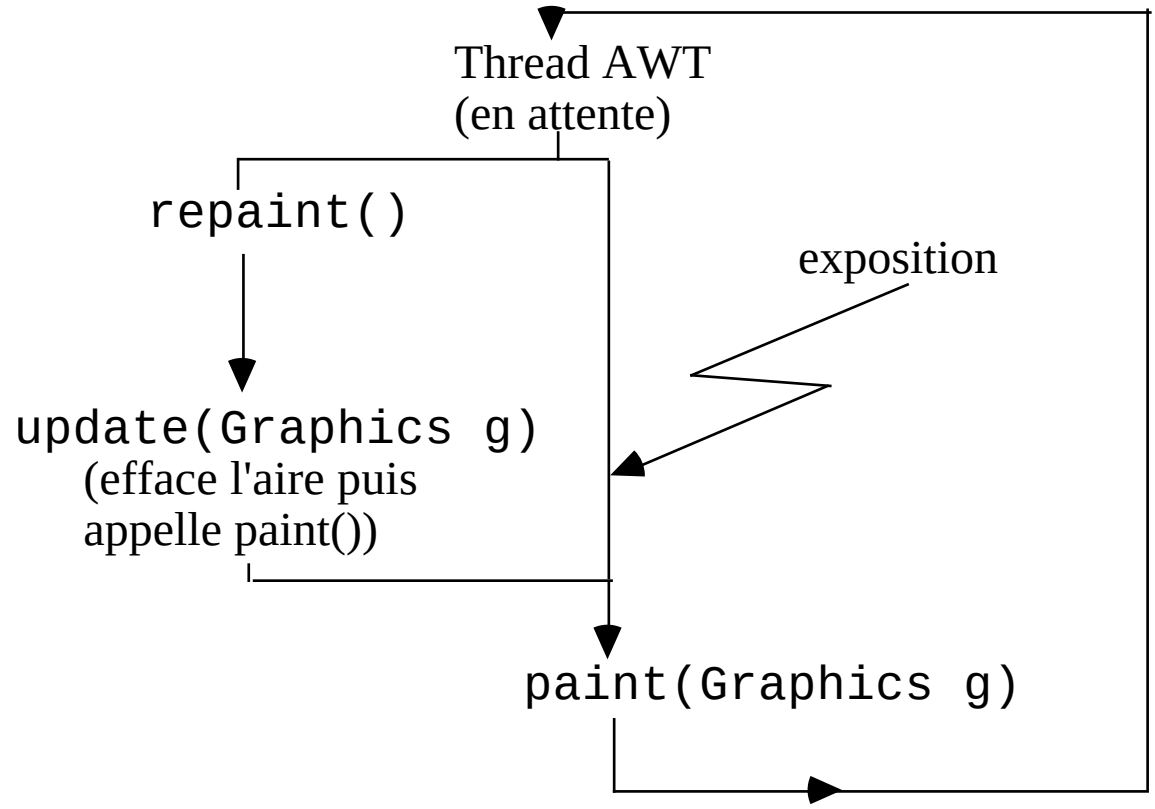
D'ailleurs update(Graphics g) de la classe Component est :

```
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.

Le moteur d'une animation

On ne relance pas dans `paint()` un appel à `repaint()` car cela aurait pour conséquence de saturer la thread AWT. D'ailleurs le code de `paint()` doit être un code qui est exécuté rapidement.

La technique consiste alors à créer une thread d'animation (donc distincte de "la thread système AWT") qui elle, lancera les `repaint()`. Par exemple pour une applet on a un squelette de code comme :

```
public
class trameAnimApplet extends java.applet.Applet implements
Runnable {
    int frame;
    int delay;
    Thread animator;

    /**
     * methode appelée lorsque l'applet devient visible à l'écran.
     * Crée une thread et la lance.
     */
    public void start() {
        animator = new Thread(this);
        animator.start();
    }
```

```
    /**
     * La methode moteur de l'animation.
     */
    public void run() {
        // Recupere l'heure de lancement de l'animation
        // (nombre de mililisecondes depuis le 1er Janvier 1970).
        long tm = System.currentTimeMillis();
        while (Thread.currentThread() == animator) {
            // affiche la prochaine "image" de l'animation.
            repaint();

            // Delai d'attente qui tient compte du temps de traitement
            //  pour l'affichage de la précédente image.
            try {
                tm += delay;
                Thread.sleep(Math.max(0, tm -
System.currentTimeMillis()));
            } catch (InterruptedException e) {
                break;
            }

            // passe a l'image suivante.
            frame++;
        }
    }

    /**
     * methode appelée lorsque l'applet n'est plus visible
     * a l'écran. Elle arrête la thread d'animation
     * (et donc les sons s'il y en a).
     */
    public void stop() {
        animator = null;
    }
}
```

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;
    }
  }
```

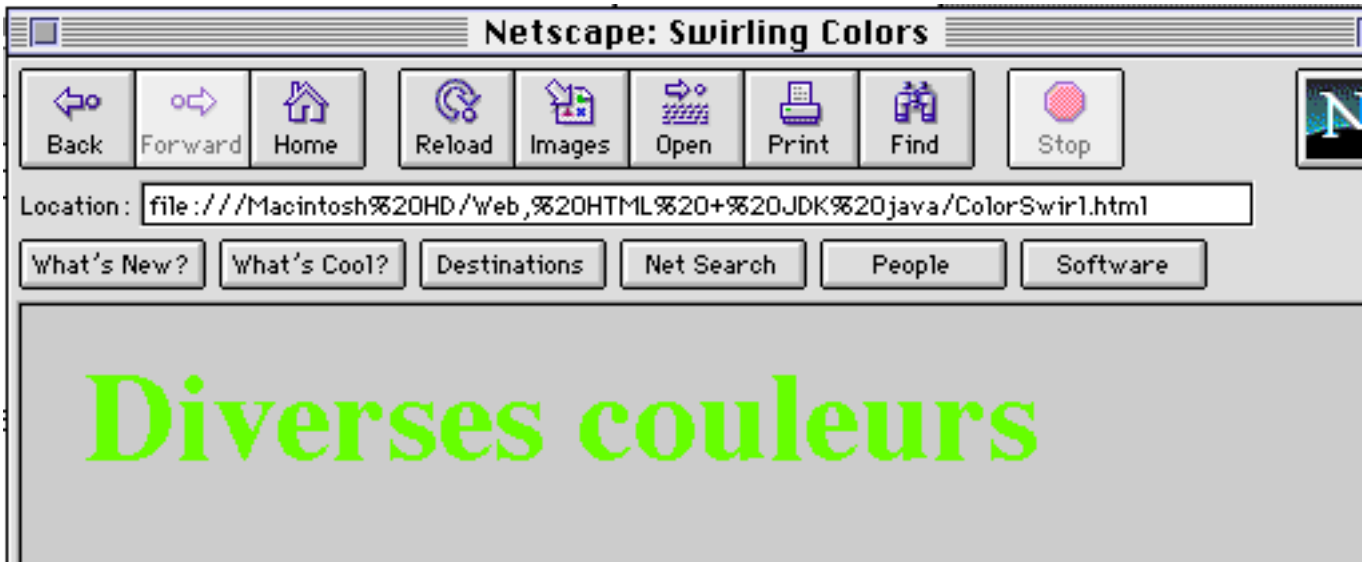
```
  public void run() {

    // 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;
    }

    // cycle through the colors
    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



remarques

- C'est un programme qui passe en revue les gammes de couleurs de manière continue
- Une telle animation même si elle fonctionne fait apparaître des tremblements. Ceci est du à la gestion du rafraîchissement en Java et plus précisément à l'effacement dans la méthode update().

Les tremblements dans les animations

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 pour le programme ci dessus. Il faut bien voir que le nouveau `paint()` avec la nouvelle couleur va dessiner par dessus l'ancien et que cela résout notre problème dans ce cas car les dessins sont opaques.

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 :

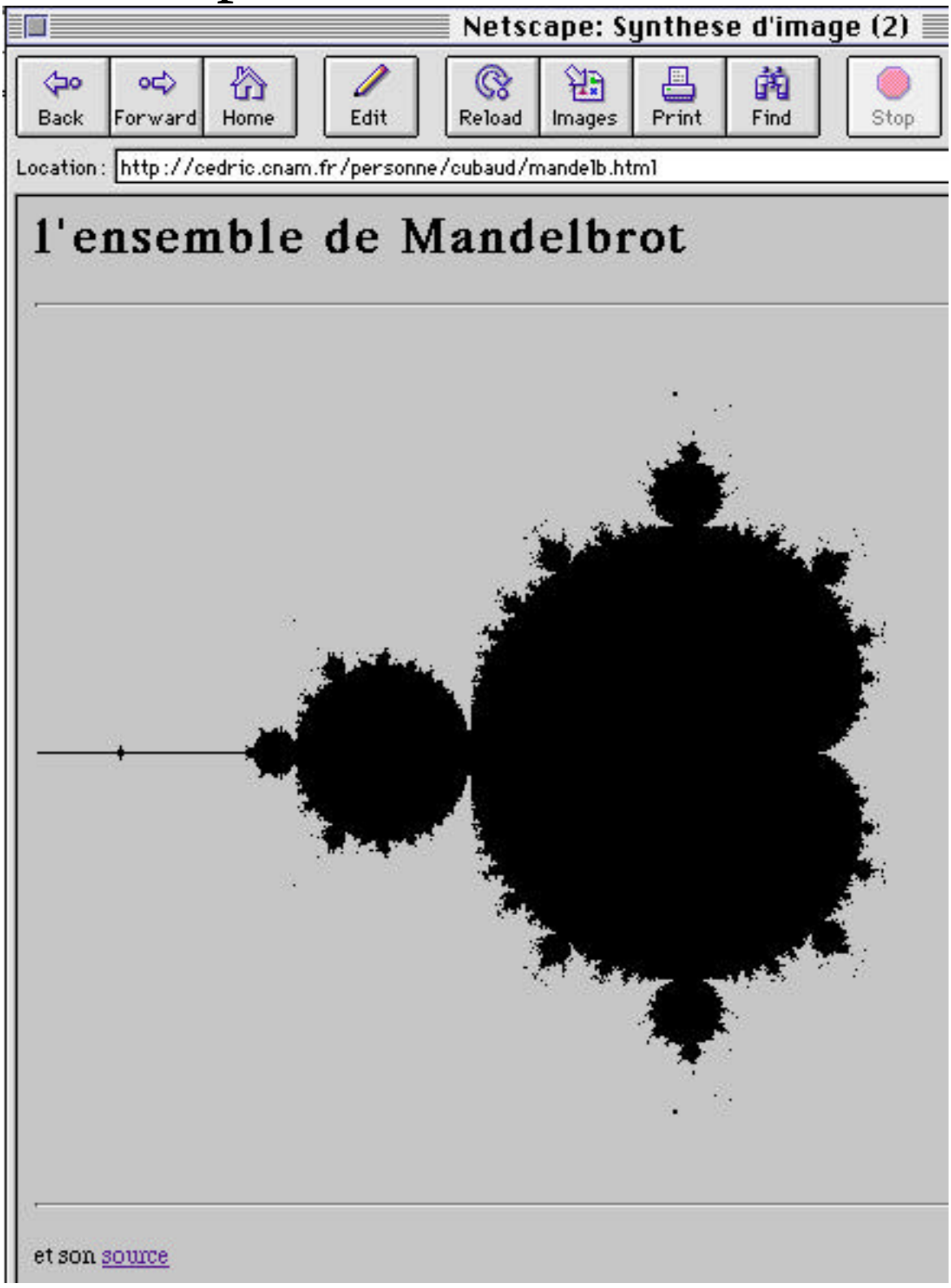
```
Image bufImg;
Graphics bufgc;

public void init() {
        bufImg = createImage(this.size().width,
        this.size().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()`.

Asynchronisme de drawImage()(suite)

Le chargement de l'image par `drawImage()` 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`).

De plus elle appelle régulièrement cet appelant i.e. cet observateur d'image, objet d'une classe qui implémente l'interface `ImageObserver` (ce qui est la cas pour la classe `Component`) pour qu'il demande à être redessiné par le lancement de `update()`.
Ceci explique les nombreux passages dans `paint()`.

Asynchronisme de drawImage()(suite)

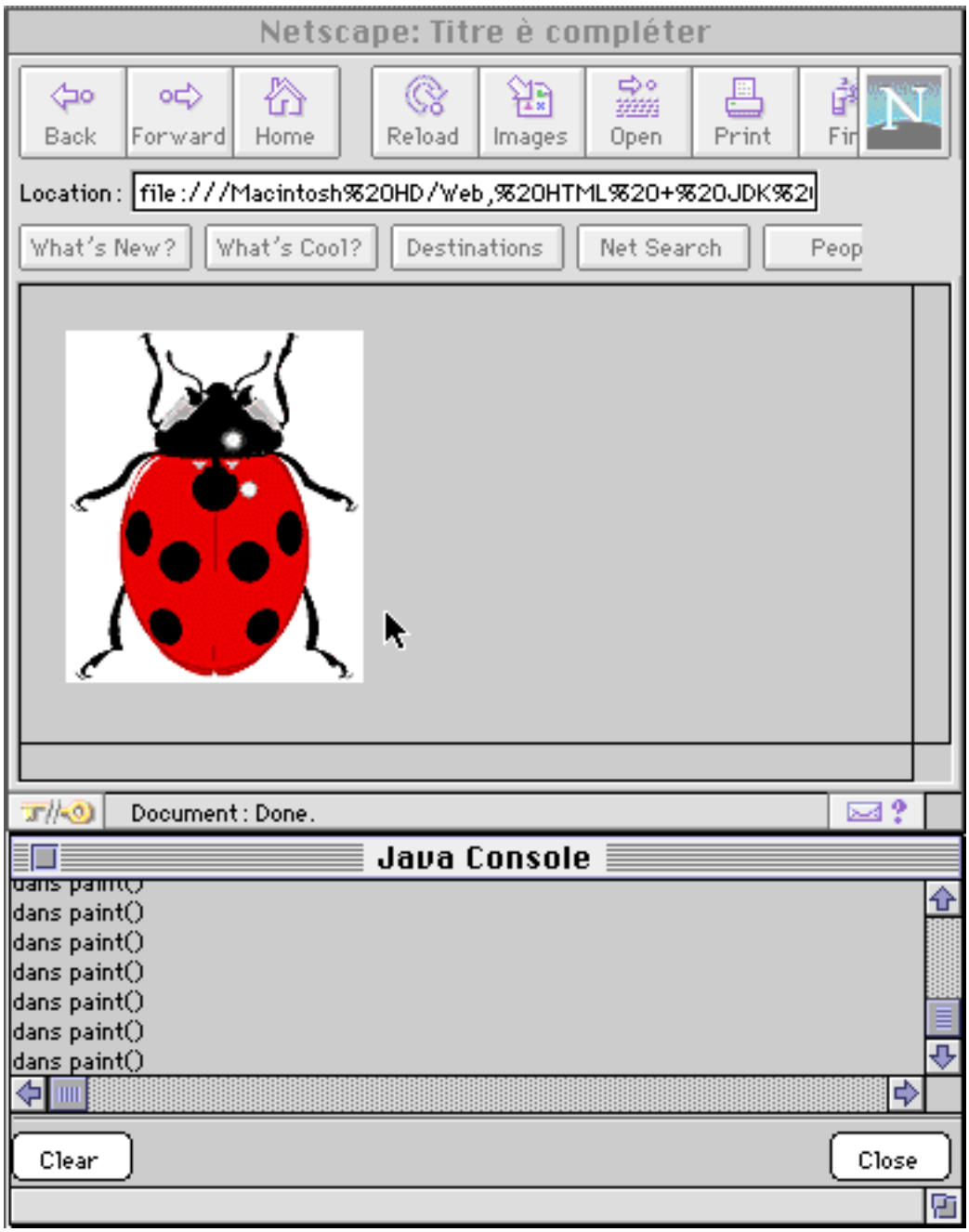


Image dans une application Java

Dans une applet, pour repérer une image, on utilise la méthode `getImage()` de la classe `Applet`. Pour une application Java, on ne peut utiliser cette méthode. On utilise alors la méthode `getImage()` de la classe `Toolkit` (qui représente l'environnement d'exécution de l'application Java) et on écrit :

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image im = tk.getImage(fichierImage)
```

Image dans une application Java (suite)

Par exemple la version application Java de l'applet ci dessus est :

```
import java.awt.*;
public class LadyBug extends Frame {
        Image bugimg;
        public static void main(String args[ ]) {
            Toolkit tk = Toolkit.getDefaultToolkit();
            Frame fr = new LadyBug(tk, "Coccinelle");
            fr.resize(200, 200);
            fr.show();
        }
        public LadyBug (Toolkit tk, String st) {
                super(st);
                bugimg = tk.getImage("ladybug.gif");
    }
        public void paint(Graphics g) {
                g.drawImage(bugimg,10,10,this);
        }
}
```

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 synchronized 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 (elles sont toutes synchronized)

`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`.

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"