Applet Fundamentals

Table of Contents

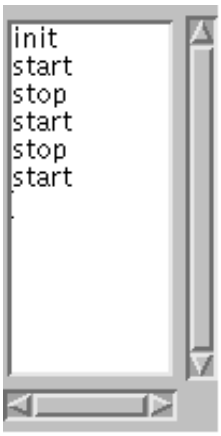
6. Applet Fundamentals

In this chapter we revisit the topic of applets. The goal is to understand fully the mechanism behind applets. What happens when the applet is loaded, or when the applet is scrolled away. Why is there no `main`, but an `init`, why do you never explicitly call the `paint` routine?

- The Life Cycle of an Applet
- Applet Graphics

6.1 Applet Life Cycle

The life cycle of an applet is determined by methods that are automatically called at its birth, its death and when it is momentarily away. We discuss below the details. To illustrate the life cycle calls, here is an applet that displays all calls in both a text area and the java console.



1. The Begin
2. Away for a Moment
3. Back Again
4. The End
5. A Minimal Appletviewer

The Begin

An applet extends a panel, so it "is" a component of the AWT. But how come it runs as a program?

1. The browser starts a Java Virtual Machine (JVM) and gives it control over the part of the screen that is specified width and height of the applet.
2. This JVM will do all things that belong to any java application, like starting the AWT-Input handler and the garbage collector. (See User Threads).
3. Next it will create an instance of the applet as specified in the class file of the applet.
4. The JVM sees the window of the browser as a kind of a Java AWT frame, and adds the applet as a panel to this frame.
5. As the main routine is in fact already running, there is an other method, `init`, for the initialization of the applet. It is called automatically by the JVM.
6. After the call to init, a call to `start` is done and then a call to the `paint` routine.

Away for a Moment

Once a browser has loaded an applet, this applet will stay running. Also when the applet is not visible, it is still running in the background. This is even the case when you have clicked to other URLs. Only when you quit the browser, the applet will exit as well.

The `stop` method is exactly meant to keep in control of the applet at these moments where the applet becomes not visible. By implementing this method, you can stop computations or "threads" like animations. In general you should stop parts of the programs that consume resources.

Back Again

Of course, once you can stop things, you should be able to restart them as well. For this purpose, there is the `start` method. This method is also automatically called at the beginning, just after `init`. When you ask the browser to reload a page containing the applet, it will call stop, immediately followed by start.

The End

Just before exiting, so when the browser quits, the applet performs a call to the `destroy` method. Here some clean-up code can be inserted, however this is in practice seldomly necessary. Please note the difference with the stop method.

A Minimal Appletviewer

As we now have all ingredients that make up an applet, we can mimick the browser with a simple Java application. The JDK appletviewer offers more functionality, but essentially does the same. So all we have to do is to make a frame, and then perform the necessary calls to init, start, stop and destroy. Here is the essential part of the code.

```
class AppletFramer extends Frame {

  Applet applet;

  AppletFramer (String appletname, int width, int height) {
    setLayout(new BorderLayout());
    try {  // get the class from its name and instantiate it
      applet = (Applet) Class.forName(appletname).newInstance();
    } catch (Exception e) {
      System.out.println(e);
    }
    resize(width, height);
    add("Center", applet); // the applet is a panel
    applet.init();
    applet.start();
    show();
  }

  public boolean handleEvent (Event evt) {
    if (evt.id == Event.WINDOW_DESTROY) {
      applet.destroy();
      dispose();
      System.exit(0);
    } else if (evt.id == Event.WINDOW_ICONIFY) {
      applet.stop();
    } else if (evt.id == Event.WINDOW_DEICONIFY) {
      applet.start();
    }
    return super.handleEvent(evt);
  }
}
```

6.2 Applet Graphics

1. Graphics in Applets
2. Graphics Object
3. Screen Refresh
   - `paint` method
   - `update` method
4. Buffering

Graphics in Applets

The idea behind the graphics of an applet is that you paint the pixels of the rectangular part of the screen that is controlled by the applet. As Java is an object-oriented language, a *graphics object* is attached to the applet, and there is a `paint` method that will take care of the coloring of the pixels.

The pixels are counted *downward and to the right* and the upper left corner is the origin. This direction is convention in computer graphics, and is the same in which you read a book. Note it is not the mathematical convention of graphics, where the origin is in the lower left corner.

Graphics Object

There is not only a graphics object for the applet, there is a graphics object attached to any component. Often it is called the *graphics context*. It contains several instance variables, that determine the effect of drawing routines like `drawText` and `drawLine`:

1. Color, with methods `setColor` and `getColor`
2. Font, with methods `setFont` and `getFont`
3. Clip region: this is a restriction of the area of pixels that are affected by the paint routine. Outside the region the pixels will not change. Default clip region is the complete component, the associated methods are `clipRect` and `getClipRect`.

Most other methods of the Graphics class are graphics routines like `drawLine`, `fillRect`, and `drawText`. Note that there is also a method `drawImage` for displaying any GIF- or JPEG images. In xxx there is more on image loading.

There are two points to note:

- The graphics object does not contain the lines, ovals, etc. as objects, neither does it contain the pixels as objects. It just represents the screen area and paints it according to the paint routine. After a paint call, it forgets about the state of pixels.
- You implement the paint method for a generic Graphics object

  ```
  public void paint (Graphics g) { ... }
  ```

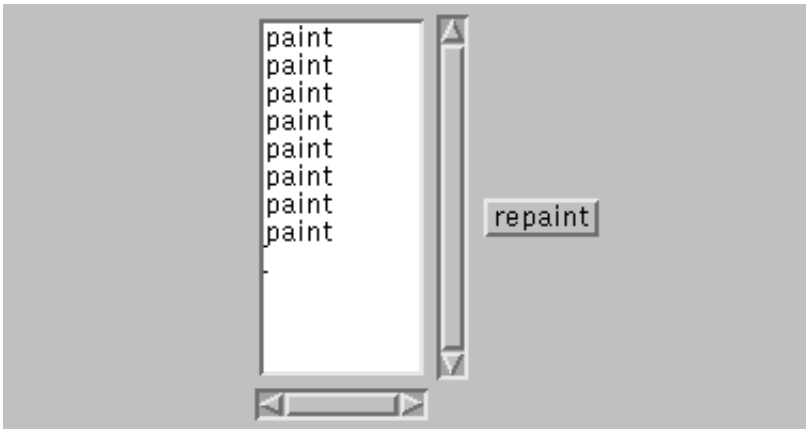
  but you hardly ever explicitly call it for the graphics object that is attached to the applet in your code (then it should read something like `this.paint(getGraphics())`). Instead the paint method is called automatically by the *Screen Updater* of the applet when a screen refresh is needed. This is an example of a "callback routine".

Screen Refresh

The paint routine is not only called when the applet starts, it can be useful when the screen needs a refresh afterwards. We distinguish two types of causes for screen refreshes:

- from within the applet (when e.g. a button is pressed, or an animation runs), which calls the `update` method and
- from the outside (when e.g. the user scrolls, or another window is moved on top), which results in a call to the `paint` method.

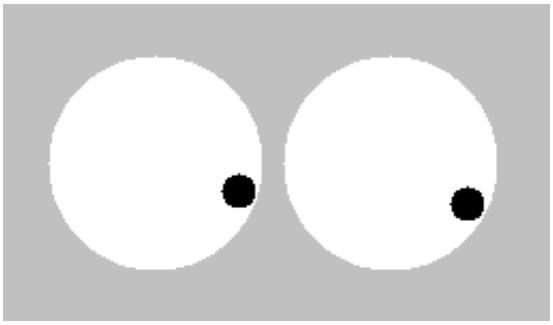
Similar to the life cycle applet, we can log the paint and update calls in a text area to see when they are exactly done.



`paint` method

The `paint` routine is used when a refresh is demanded by an action coming from the outside. The screen area is erased, colored with the background color, and the paint routine performs all its steps. This all happens automatically for instance when the browser moves to an other page, and back to the page with applet, or when the window is iconified, and deiconified again. To be more precise, the action concerning the window generates an event, and the event dispatcher of the applet reacts with a call to paint.

Note that in the Xeyes applet below, the behaviour with respect to outside actions is good, but the internal events coming from the movements of the mouse cause some flickering. The code is in Eyes.java



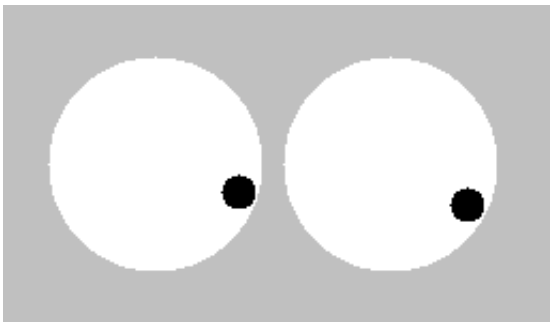
`update` method

When the action that requires a screen refresh comes from within the applet, a more subtle way is possible. Instead of a complete redrawing of the screen, you can specify precisely which pixels should be redrawn. For this goal, you can implement the method `update(Graphics g)`. However, the default implementation of update consists of a complete erase of the screen, followed by a paint call.

Similar to the paint routine, you will seldomly call the update directly. If you need a refresh, say because a button was pressed, you add a `repaint()` method to your code for the action. This will generate an event, and the event dispatcher will call update as soon as its turn is there. If many repaint calls arrive at the event dispatcher in a short time interval, they all can be collapsed, and then only one update call is done. This favours efficiency, but implies that you have to program the update method quite carefully.

As a first example we take the movement of the eyes as in the paint method, but now in the update method. We "forget" that the screen is now not redrawn from scratch. The code is UpdatedEyes1.java

As a second example we take a correct implementation. Now the update first colors the old position of the black spots white again. Note that in the beginning the black spots are not there, as in the paint only the white irises are drawn. The code is in UpdatedEyes.java.



Buffering

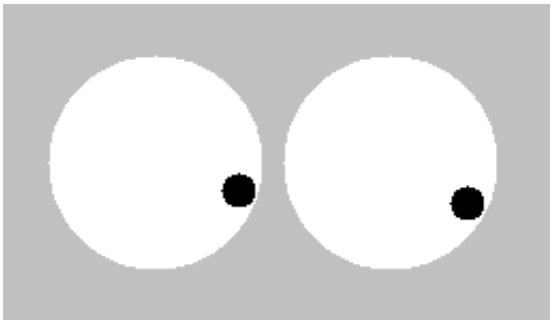
What it is?

The flickering effect that you seen when a series of paint calls is done, can also be reduced in another way. It is called *buffering*, or *double buffering* to distinguish from a hardware buffer for the video. The idea is to build up an off-screen image, and then map it to the screen with a single `drawImage`. A complete erase of the picture is prevented, all pixels get the right color immediately, and the flickering

is much reduced.

The advantage is that is quite simple to use, you just insert some standard code, and adapt the paint and update routines in a standard way. There is a disadvantage which can be relevant if the applet is time critical: with buffering you still draw a complete image. So it might be more efficient to restrict to the part that is really changing. You can do this for instance with help of the clip region, or you can only update the part of the off-screen image that is really changing. Or you can use a clever combination of these approaches.

As example we take again the Xeyes. Note that in the beginning the black spots are not there, as in the paint only the white irises are drawn. The code is in BufferedEyes.java.



How does it work?

1. First create a so-called off-screen image and attach a graphics object to it.

```
public class BufferedEyes extends Applet {

  ...

  Dimension offDimension = this.size();
  Image offImage = createImage(offDimension.width, offDimension.height);
  Graphics offGraphics = offImage.getGraphics();

  ...
}
```

   Note that in the BufferdEyes example, the declaration and the instantiation are separate, which makes it more dynamic as the size of the applet might change.
2. Now rewrite the update as to work only on the graphics object of the off-screen image. However, we start with a screen erase.

```
public void update (Graphics g) {

    offGraphics.setColor(getBackground());
    offGraphics.fillRect(0, 0, offDimension.width, offDimension.height);

    offGraphics.setColor(Color.white);
    offGraphics.fillOval(100, 100, 100, 100);
    offGraphics.fillOval(210, 100, 100, 100);
    offGraphics.setColor(Color.black);
    offGraphics.fillOval((int)(lx-8), (int)(ly-8), 16, 16);
```

```
        offGraphics.fillOval((int)(rx-8), (int)(ry-8), 16, 16);

        g.drawImage(offImage, 0, 0, this);
    }
```

3. Add a call to drawImage in the update method to map the off-screen image to the screen.

```
    public void update (Graphics g) {

        offGraphics.setColor(getBackground());
        offGraphics.fillRect(0, 0, offDimension.width, offDimension.height);

        offGraphics.setColor(Color.white);
        offGraphics.fillOval(100, 100, 100, 100);
        offGraphics.fillOval(210, 100, 100, 100);
        offGraphics.setColor(Color.black);
        offGraphics.fillOval((int)(lx-8), (int)(ly-8), 16, 16);
        offGraphics.fillOval((int)(rx-8), (int)(ry-8), 16, 16);

        g.drawImage(offImage, 0, 0, this);
    }
```

4. Adapt paint as to work with the update implementation.

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

Exercise: why do we work here with the update method, and not with the paint method?