Tutorial: Building a Java applet

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Tutorial tips 2
2. Java, development, and applets 3
3. Loading and displaying images 9
4. Exceptions and MediaTracker class 13
5. Offscreen image buffering 15
6. Image rotation algorithm using copyArea 18
7. Threading and animation 20
8. Graphic output methods and applet parameters 24
9. Wrapup 27

Section 1. Tutorial tips

Should I take this tutorial?

This tutorial walks you through the task of building a graphical Java applet. Along the way, you'll learn Java syntax and work with Java class libraries. It requires that you know some object-oriented programming.

Navigation

Navigating through the tutorial is easy:

* Select Next and Previous to move forward and backward through the tutorial.
* When you're finished with a section, select the next section. You can also use the Main and Section Menus to navigate the tutorial.
* If you'd like to tell us what you think, or if you have a question for the author about the content of the tutorial, use the Feedback button.

Section 2. Java, development, and applets

Introduction

This section describes the Java language, the development process, and what an applet is. After completing this section, you should be able to:

* Describe the syntax of a Java class
* Understand the basic structure of an applet and how it interacts with a Web browser
* Code a simple Java applet source file
* Write output to the system console
* Code HTML to invoke an applet
* Compile and execute a Java applet using the appletviewer

Anatomy of a class

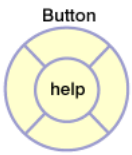
Any Java source file (.java) has this physical structure:

```
import statements;
class definition {
     instance variable definitions;
     method definition (argumentList) {
       local variable definitions
         statements;

       } // end of method definition

     // more methods ...
     } // end of class definition
```

A *class* is used to instantiate specific objects (each with possibly different values) in the heap. For example, the figure below shows help as an instance of class Button.

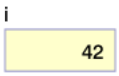


Primitive and object variables

Variables are represented on the stack as either:

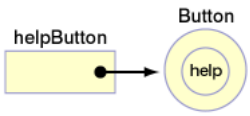
* A built-in primitive type (byte, short, int, long, char, float, double, or boolean) that uses *value semantics*

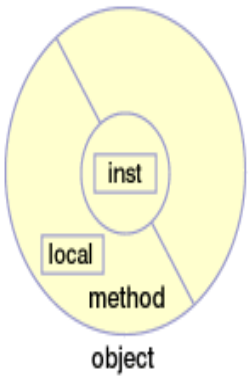
```
int i = 42;  /* i holds value (42) */
```



* An object type (extended from java.lang.Object) that uses *reference semantics* (like a pointer)

```
Button helpButton = new Button("Help");
/* helpButton is an object ref */
```





Lifetime of a variable

A variable has a *storage class*, which sets its lifetime.

* *Local variables* are local to a block of code, that is, allocated at entry to a block and discarded at exit. (A block of code can be either a class or a method.)
* *Instance variables* are local to an object, that is, allocated when an object is instantiated and discarded when it is garbage-collected.
* *Class (static) variables* are local to a class, that is, allocated when a class is loaded and discarded when it is unloaded. Static variables are not associated with objects and the classes they are defined in cannot be instantiated.

An object (referred to by a variable) is marked for garbage collection when there are no references to it.

C-like Java

Most of the C programming language statements are present in the Java language:

```
float sqrt(float value) {
/* compute a square root badly! */
  float guess;
  int loops;

  if (value <= 0)
    return 0;

  else
    guess = value/2;
    for (loops = 0; loops < 5; loops++) {
      guess = (guess + value /guess) /2;
    }

  return guess;
}
```

As well as `if/else`, `for`, and `return`, the Java language has C's `while`, `do/while`, and `switch/case/default` statements.

Messages and object communication

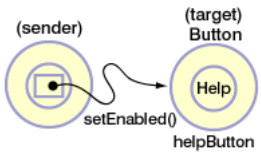
Objects use *messages* to communicate.

Valid messages are represented by a public *interface*.

Messages in Java correspond to *method calls* (invocations) in C.

Messages must have a *reference* to the target object to send a message.

```
/* Send help button a msg */
helpButton.setEnabled(false);
```

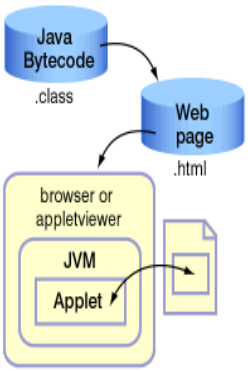


If no object is specified, the current object *(this)* is assumed, for example, `f()` implies `this.f()`.

Java development process

The `javac` command compiles Java source code (.java) into *bytecode* (.class).

These bytecodes are loaded and executed in the Java virtual machine (JVM), which is embeddable within other environments, such as Web browsers and operating systems.



Displaying applets

An *applet* is a Java program that is referenced by a Web page and runs inside a Java-enabled Web browser.

An applet begins execution when an HTML page that "contains" it is loaded.

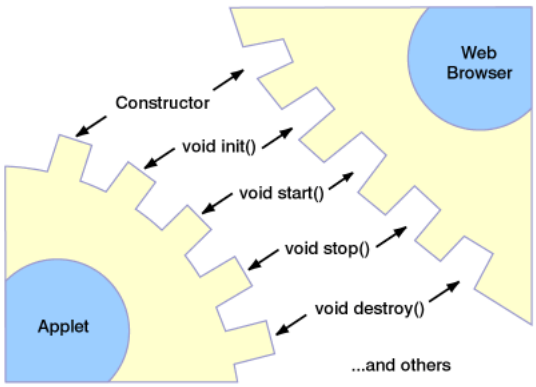
Either a Java-enabled Web browser or the appletviewer is required to run an applet.

The browser detects an applet by processing an Applet HTML tag, for example:

```
<APPLET CODE=ImgJump WIDTH=300 HEIGHT=200>
</APPLET>
```

Web browser-to-applet interface

This figure shows the browser-to-applet interface.



Structure of a simple applet

By *inheriting* from java.applet.Applet, the necessary structure is defined to coordinate with a Web browser. `Extends` is the keyword in the Java language used to inherit classes.

```
public class MyApplet extends java.applet.Applet {

   public MyApplet( ) { //create applet

     System.out.println("In ctor");
   }
   public void init( ) { // initialize
     System.out.println("In init");
   }
   public void start( ) { // display page
     System.out.println("In start");
   }
   public void stop( ) { // leave page
     System.out.println("In stop");
   }
   public void destroy( ) {// clean up
     System.out.println("In destroy");
   }
}
```

The `init()` method is run only when the applet first starts; `start()` is executed when the applet is displayed or refreshed.

Note that standard out (or output for the `println` command) is sent to the console window if you are running within a browser or to a DOS window if you are running your applet from appletviewer via a DOS window.

Activity: Basic applet and HTML using development tools

Create a Panorama Applet class and HTML file. Override the init method and print out a trace message. Test and verify the message is printed.

Here are the steps:

1. Create `Panorama.java` using a text editor.
   - In the file, add a public Panorama class and have it inherit from `java.applet.Applet`.
   - In the class, add an `init` method that takes no arguments and returns void.
   - In `init`, issue `System.out.println` of "In Panorama.init". This provides a message when the method is invoked that traces the flow through the applet.
   - Continue to add trace output to all methods you add to the applet.
   - Save the file and exit the editor. Your source code should look like *this source code* .
2. Compile using `javac`, and correct errors.
3. Create `Panorama.html` using a text editor. In `Panorama.html`, add an applet tag to invoke the `Panorama.class` with a width of 600 and a height of 130. Your file should look like *this HTML file* .
4. Save both your Java source file and HTML file in the same directory.
5. Run HTML using appletviewer. Did a blank frame appear, and did the line "In ..." appear separately on the console? If it did, quit or close the appletviewer and continue with the tutorial.

Section 3. Loading and displaying images

Introduction

This section describes how to load and display an image. After completing this section, you should be able to:

* Describe what a Java package is
* Import a package containing additional Java types
* Declare a reference to an image object
* Load an image in an applet
* Code a paint method
* Draw an image into a Graphics object

Packages

A *package* is a named group of classes for a common domain: `java.lang`, `java.awt`, `java.util`, `java.io`. Packages can be imported by other source files making the names available:

```
import java.awt.*;     // All classes
import java.awt.Image; // One class
```

Explicit type name: `java.awt.Image i1;`

Implicit type name: `import java.awt.*; Image i2;`

The `java.lang` package is automatically imported by the compiler.

The java.lang package

The `java.lang` package contains more than 20 classes, of which the most useful are System, String, and Math. It also contains the Thread class, the Runnable interface, and the various wrapping classes (such as Integer).

* `java.lang.System` class provides the standard streams `in, out,` and `err` as public class variables.
* `java.lang.String` class contains methods that provide functions similar to C's strxxx functions, including `charAt, compareTo, concat, endsWith equals, length, replace, startsWith, subString, toLowerCase`, and `trim`.
* `java.lang.Math` class contains a number of mathematical methods such as `abs, sin, cos, atan, max, min, log, random`, and `sqrt`. It also contains `E` and `PI` as class constants (static final).

Loading and drawing images

The typical way to load images in Applets is via the getImage() method.

```
Image getImage(URL) // Absolute URL
Image getImage(URL, String) // Relative URL
```

For example:

```
Image img = getImage(getDocumentBase(), "x.gif");
```

This example returns a reference to an image object that is being asynchronously loaded. The `getDocumentBase()` method returns the address of the current Web site where the applet is being executed. The `x.gif` is the actual image being loaded

After the image is loaded, you would typically render it to the screen in the Applet's paint method using the Graphics method. For example:

```
g.drawImage(img, 0, 0, this); // img is the image that is drawn on the
                              // screen in the 0, 0 position.
```

Graphics class

A Graphics object is usually only obtained as an argument to update and paint methods:

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

The Graphics class provides a set of drawing tools that include methods to draw:

* rectangles (`drawRect, fillRect`)
* ovals (`drawOval, fillOval`)
* arcs (`drawArc, fillArc`)
* polygons (`drawPolygon, fillPolygon`)
* rounded rectangles (`drawRoundRect, fillRoundRect`)
* strings (`drawString`)
* images (`drawImage`)

For example:

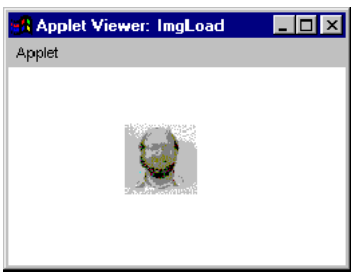
```
g.drawImage(i, 0, 0, this);
```

Simple applet that loads and draws Image

```
<applet code=ImgLoad width=300 height=200>
</applet>

import java.awt.*;
  public class ImgLoad extends java.applet.Applet {
    Image i;
    public void init() {
      System.out.println("In init");
      i = getImage(getDocumentBase(), "Test.gif");
    }
    public void paint(Graphics g) {
      System.out.println("In paint");
      int x = (int)(Math.random() * size().width);
      int y = (int)(Math.random() * size().height);
      g.drawImage(i, x, y, this);
    }
  }
```

Simple page viewed by appletviewer



Run the Image Jump Applet

Activity: Load and display image

In the `init` method, read in the image `Panorama.gif`. Override the `paint` method and draw the image along with a trace message. Test and verify the message is printed and the image is displayed.

Here are the steps:

1. Above and outside the class, add an import of `java.awt` package at the start of the file.
2. In the class at the same level as the `init` method, make an instance variable, img, for the image to be loaded and drawn. Its class type is Image.
3. In **init** after the output statement, set this instance variable by calling Applet's `getImage` method. Pass it the `getDocumentBase()` and the string "Panorama.gif".
4. In the class beneath the `init` method, add a `paint` method. This method receives a Graphics object, g, as an argument.
5. In `paint`, output the tracing message "In Panorama.paint".
6. In `paint`, send the drawImage message to g object passing the image (img), x (0), y (0), and the Applet (this).
7. When you have completed these steps, your file should look like *this source code* .
8. Copy `Panorama.gif` from *this file* to the directory where you are storing your source. Compile and run. Did the image get displayed? Does the `paint` method get called multiple times? It should because the image is asynchronously loaded.

Section 4. Exceptions and MediaTracker class

Introduction

This section describes handling exceptions and using the MediaTracker class to synchronize image loading. After completing this section, you should be able to:

* Code try-catch blocks to handle exceptions
* Code a MediaTracker to synchronize image loading

Exceptions

Run-time errors are called exceptions and are handled using try-catch:

```
int i, j;
i = 0;
try {
   j = 3/i;
   System.out.println("j=" +j);
}
catch (ArithmeticException e) { // handler
   e.printStackTrace();
   System.out,println("Exception ' " + e +
    " 'raised - i must be zero");
}
```

At run time, this code will display and trace the message:

```
"Exception 'java.lang.Arithmetic: / by zero' raised - i must be zero"
```

Using MediaTracker class

The `getImage` method loads images asynchronously. This means that users can start interacting with a program before it's ready, or that early phases of animation don't look right because some images are not fully loaded.

The MediaTracker class keeps track of the loading of images. Three key methods are:

* `addImage` -- adds image to list of objects being tracked
* `checkAll` -- checks if all images have finished loading
* `waitForAll` -- blocks until all images are loaded

Currently MediaTracker supports only images, not audio.

Synchronous image loading

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

public class ShowMedTrk extends Applet {
   Image i;
   public void init( ) {
     i = getImage( getDocumentBase( ), "Test.gif");
     MediaTracker mt = new MediaTracker(this);
     mt.addImage(i,0); // add to group 0
     try {
       mt.waitForAll( ); }
     catch(InterruptedException e) {
       e.printStackTrace( ); }
   }
   public void paint(Graphics g) {
     g.drawImage(i, 0, 0, this);
   }
}
```

Activity: Add a media tracker

At the end of the `init` method, create a media tracker, add the image to it, and wait for it to be loaded.

Here are the steps:

1. In the class, make an instance variable, mt, that will reference a media tracker object. Its type is MediaTracker.
2. In `init`, after getting the image, create a new media tracker object passing in the Applet(this).
3. In `init`, after creating the media tracker, send to the media tracker the message `addImage` passing it the image (img) to be tracked, and put it in group 0.
4. In `init`, after adding the image, send to media tracker the message to wait for all images (in our case only one) to finish loading.
5. When you have completed these steps, your file should look like *this source code* .
6. Compile. Any errors? Was it from a missing exception? If so continue. Otherwise, fix your syntax errors.
7. Don't forget to add a try-catch to handle InterruptedException. In the catch, print out a trace of the execution stack using the exception's `printStackTrace( )` method.
8. Compile and run. Did the image appear all at once (not incrementally)? Does the `paint` method now get called only once? It should because you are synchronously loading the image.

Section 5. Offscreen image buffering

Introduction

This section describes offscreen image buffering. After completing this section, you should be able to:

* Declare primitive integer local variables
* Query the height and width of an image
* Create an offscreen image buffer
* Extract a Graphics object from an offscreen image
* Draw an image into an offscreen image buffer
* Draw an offscreen image buffer to the screen

Offscreen image buffering

It is important to show a smooth transition going from one image to another within an animation. (Animations will be discussed in detail in Section 7.) Double-buffering is a technique that helps you accomplish this. You would draw all of your images into an off-screen "buffer" and then copy the contents of the buffer to the screen all at once. This avoids the flickering that you might see with animations that don't use the off-screen buffering procedure. Images are simply erased and redrawn over and over again. The buffer is actually an off-screen Java Image object.

Note that when you render into a Swing component, Swing automatically double-buffers the display.

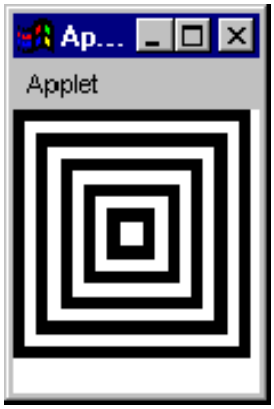


Image class and offscreen buffers

An image's width and height can be queried:

```
int iw = img.getWidth (null);
int iw = img.getHeight (null);
```

Pre-render graphics into an *offscreen image buffer* and just draw it later as one operation:

```
// Set up the off-screen image (buf) for
// double-buffering
buf = createImage (100,100);
Graphics bufg = buf.getGraphics( );

// Draw into the off-screen buffer
for  (int i = 0; i < 50; i += 5)  {
   bufg.setColor(i % 10 == 0 ?
      Color.black : Color.white);
   bufg.fillRect(i, i, 100 -i * 2,

   100 - i * 2) ;    }

// Now copy the off-screen buffer onto the screen
g.drawImage(buf,0,0,null);
```

Activity: Begin the rotate algorithm and create a secondary image

At the end of the `init` method, create a secondary image that has the same height as the original but with a width equal to the original's plus the amount to rotate (initially set to 100). Also draw the original image into the secondary image's graphics context. In the `paint` method, draw the newly filled secondary image onto the applet's display area. The same image should display, but now it is being drawn from the secondary buffer. Now you are ready to make it rotate.

Here are the steps:

1. In the class, make instance variables for the original image's width (iw) and height (ih) as integers, another (secondary) image (buf) of type Image, its graphics context(bufg) of type Graphics and an integer shift amount (delta x) for rotating the image. Start the shift amount at 100 which should help you test the rotation algorithm. Later you will change this to be smaller, but for now make it 100.
2. In `init`, after the media tracker code, get the original Image's (img) width and height into two instance variables, iw and ih respectively. Pass in "null" to each method as its parameter (this shows there is no image observer).
3. In `init`, after waiting for the image to be fully loaded, create a secondary image using Applet's `createImage`. Pass into it two arguments: the extended width which includes the shift amount (iw + shift), and the height (ih).
4. In `init` next, get the secondary image's graphic context, bufg, using Image's `getGraphics` and put it in an instance variable. This will be used to draw into the secondary image in the next step.
5. In `init`, draw the original image into the secondary image using the secondary Graphic's context, bufg, and drawImage. Pass it the original image (img) , x (0), y (0), and no image observer (null).
6. In `paint`, change the `drawImage` to now draw the secondary image (buf) into the original graphics context, g.
7. When you have completed these steps, your file should look like *this source code* .
8. Compile and run. The secondary image should display, which should look just the same as the previous activity. If so, continue to the next activity.
9. Is the applet area white? If so, go back and check your code. Make sure you draw the original image (img) into the secondary buffer's graphic context (bufg) in `init` and then later in `paint` you draw the secondary buffer's image into the original's graphics context (g).

Section 6. Image rotation algorithm using copyArea

Introduction

This section describes the algorithm for rotating an image using the copyArea method of the Graphics class. After completing this section, you should be able to:

* Code an `advance` method to incrementally rotate an image.
* Use `copyArea` to move areas of pixels from one location to another.
* Develop an image rotation algorithm.

Image rotation algorithm overview

1. Create buffer with extension of N columns.
2. Shift image to the right N columns filling in the extension.
3. Wrap extension (N columns) back to the beginning of the image.
4. Display buffer (excluding extension).

Graphics copying of image pixels

A rectangle of image's pixels can be copied from one location to another using the Graphics class' `copyArea` method:

```
Image i = createImage(iw, ih);
Graphics gc = i.getGraphics();
gc.copyArea(x, y, w, h, dx, dy);
```

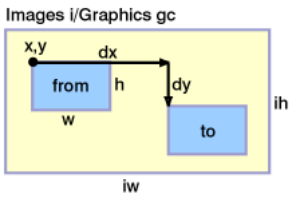


Image rotation algorithm specifics

Create a buffer with an extension of dx columns: **i = createImage(iw + dx, ih);**

1. Shift the image to the right dx pixels: **gc.copyArea(0, 0, iw, ih, dx, 0);**
2. Wrap the extension (of dx pixels) back to the beginning: **gc.copyArea(iw, 0, dx, ih, -iw, 0);**

Activity: Complete the rotate algorithm using copyArea

Add an `advance` method which will later be called to animate the image. In it, output a trace message then rotate the image by copying the image twice using `copyArea`. Test the `advance` method by calling it in the `init` method with the previously defined shift amount of 100. Verify that the image has been shifted by comparing it to output from the previous activity.

Here are the steps:

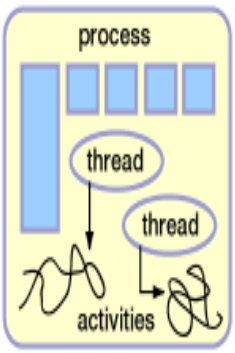
1. In the class, add an `advance` method to the Applet.
2. In `advance`, add the standard tracing message.
3. In `advance`, copy the pixels in the secondary graphics context, bufg, and rotate the image. Call Graphic's `copyArea` twice to accomplish this.
4. In `init`, call `advance` (to test the shift algorithm). This will be removed in a later step, but will allow you test the shift algorithm now.
5. When you have completed these steps, your file should look like *this source code* .
6. Compile and run. Did the shifted image get displayed? You should see a slight visual discontinuity about one inch from the left side. Can you find it?
7. Verify that the image has been shifted by comparing it to output from the previous activity.

Section 7. Threading and animation

Introduction

This section describes concurrency using threads and interfaces, and also describes how to deactivate a thread using its `sleep` method. After completing this section, you should be able to:

* Create a thread and start it
* Stop a thread and nullify the reference to it
* Deactivate a thread using the `sleep` method



Processes, threads, and activities

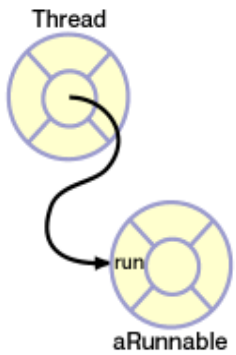
On most computer processors, multiple processes execute multiple programs.

A *process* encapsulates many things including:

* Address space
* Current state of the computation (machine registers, call stack, and so on)
* A unit associated with allocation of resources (for example, open files)

A process can have many *threads*. A thread embodies only the state (management) of an activity (aka lightweight processes).

An *activity* is a sequence of operations (thread of control or flow of communication).



Threads and runnable objects

The `java.lang.Thread` class is the base for all objects that can behave as threads.

A Thread object is conceptually just an activity. Such an activity needs a focal point (that is, what code it needs to run).

Each Thread needs a reference to a Runnable object whose `run()` method it will execute in.

To facilitate this protocol, Java supplies a simple interface `java.lang.Runnable`:

```
public interface Runnable {
   public abstract void run( )
}
```

Roles as interfaces

An *interface* encapsulates a coherent set of services and constants, for example, a role.

An *object*, in order to participate in various relationships, needs to state that it fulfills a particular role with the `implements` keyword.

```
public class X implements Runnable { ... }
```

All methods inherited from an interface must be implemented (or redeclared as abstract). The implementation does not necessarily have to have any body code.

```
public void run( ) {
  ...
}
```

A class can both extend one class *and* implement one or more interfaces.

An applet as runnable

Many times an applet will have one additional (usually continuous) activity, for example, an animation loop. In such cases it is quite likely the class will both extend Applet *and* implement Runnable:

```
class ImgJump extends Applet implements Runnable {
 private Thread t;
 // ...
```

Its `start` method will create and start a thread:

```
t = new Thread(this);
t.start( );
```

Its `stop` method will stop and nullify the thread:

```
if ( t != null ) {
  t.stop( );
  t = null;
}
```

Basic animation

Animation is the display of a series of pictures over time to give the illusion of motion. A basic computer animation loop consists of:

* Advancing to the next picture (in an offscreen buffer)
* Updating the screen by repainting it
* Delaying for some period of time (typically 1/24 of a second for film or 1/30 of a second for video)

For example:

```
public void run( )
{
  while (true)
  {
   advance( ); // advance to the next frame
   repaint( ); // repaint the screen
   try { Thread.sleep(33); } // delay
   catch(InterruptedException e) { }
  }
}
```

Runnable applet with sleep

```
public class ImgJump extends java.applet.Applet implements Runnable {
   int x, y, Thread t;
   public void start( )
   {
     t = new Thread(this); t.start( ); }
     public void stop( )
     {
       if (t != null) { t.stop( ); t = null; }
     }
     public void run( )
     {
       while(true) {
        x = (int) (Math.random()* size().width);
        y = (int) (Math.random()* size().height);
       }
       repaint();
       try{ Thread.sleep(2000);  }
       catch(InterruptedException e) { return; }
     }
   }
 }
```

Activity: Threading and image animation

Override the `start` method to create and start a thread. Make the Applet extend the Runnable interface and add a `run` method. Loop forever in the `run` method calling the `advance` method. Change the shift amount from 100 to 1. After the call to the `advance` method, delay the thread's execution for 200 milliseconds.

Here are the steps:

1. In the class, add a `start` method to the applet (no need to add stop yet). In `start`, add the standard tracing method.
2. In the class, create a new instance variable, t, to hold a thread.
3. In `start`, create a new thread and start the thread.
4. Change the Applet to inherit from Runnable (implement Runnable).
5. In the class add a `run` method. Match the method specification found in Runnable. In `run`, add the standard tracing message.
6. In `run`, after the tracing message, add an infinite loop using `while(true)`. This is your animation loop.
7. In the class, now reset the amount to shift to 1 (was 100).
8. In `init`, remove the invocation of `advance` and move it to within the `run` method's infinite loop.
9. In `run`, after the call to `advance`, add a call to Applet's `repaint`. In `run`, after the call to `repaint`, call Thread's `sleep` (a class method). Sleep the thread for 200 milliseconds. Don't forget to put a try-catch around it to handle the InterruptedException. If the exception occurs, print out an execution stack trace and return from the `run` method.
10. When you have completed these steps, your file should look like *this source code* .
11. Compile and test. Does the animation work? Does the image scroll?

Section 8. Graphic output methods and applet parameters

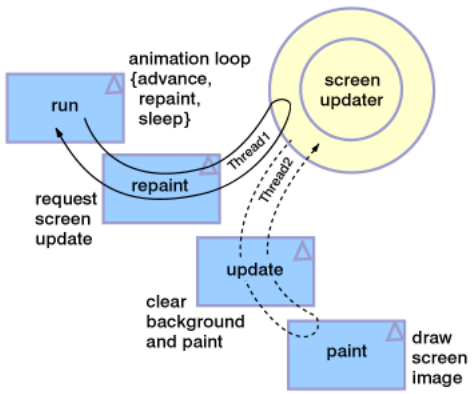
Introduction

This section describes the `repaint`, `update`, and `paint` methods along with how they interact. After completing this section, you should be able to:

* Describe the functionality of `repaint`, `update` and `paint`
* Reduce flicker by overriding the `update` method and have it only call the `paint` method

---

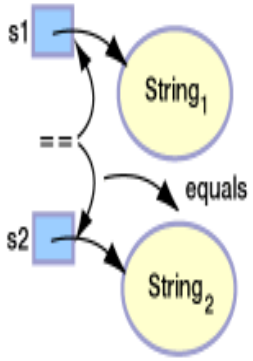
Graphic output threading



The repaint, update, and paint methods

These methods define graphics output:

* Use the `repaint()`, `paint()`, and `update()` methods to generate graphic output. The default `repaint` indirectly invokes `update`.
* The default `update` method clears the background (potential source of flicker) then calls `paint`.
* The `paint` method draws the output. The `paint` method alone is called when old areas are re-exposed (for example, moving an obscured window).



String class

Strings are objects, defined using the Unicode 1.1.5 character set (16-bit international character set).

Strings are constructed from literals, char arrays, and byte arrays. They are *bounds checked* with a `length()` method.

```
String s1 = "Testing";
int l = s1.length( )
```

Concatenation operations:

```
String s2 = s1 + " 1 ";
s2 += "2 3";
```

Comparison operations:

```
if (s1 == s2)       // Object reference check
if (s1.equals(s2) ) // Contents check
```

Note, when you are comparing values, you should use the `.equals()` method.

Passing parameters to an applet

Zero or more parameters (encoded as name-value string pairs) can be specified in the HTML using the PARAM tag:

```
<APPLET CODE=ImgJump WIDTH=300 HEIGHT=200>
<PARAM NAME=image VALUE="Test.gif">
</APPLET>
```

An applet (in the `init` method) can access each PARAM tag by using the `getParameter` method. Specifying the NAME string returns the VALUE STRING. If the NAME string is not found, null is returned:

```
String imgName = getParameter("image");
if (imgName == null) ... // set default
else ... // decode or convert
```

Activity: Display quality and parameterize

Override the `update` method to only call the `paint` method and not clear the background. Comment out all trace messages. In the `init` method get and decode parameter for the image name.

Here are the steps:

1. In all methods, comment out all debugging messages.
2. In the class, add (that is, override and replace) the Applet's `update` method and pass it a Graphics context, g. Have this method call `paint` and pass it the Graphics object, g. This will replace the default `update` method that clears the background and then calls `paint`.
3. In `init`, just before getting the image, use `getParameter` and get the parameter value for the GIF file name (set the local variable named imgName) using the parameter name of "image". If the value is null, then use "Panorama.gif". Modify the call to `getImage` that follows to pass in imgName.
4. When you have completed these steps, your file should look like *this source code* .
5. Modify your HTML file between the applet and end-applet tags and add a parameter (param) tag for the image name. Initially set this to the same value as your previously hardcoded values. Your HTML file should look like this HTML source.
6. Compile and run. Does the image flicker less? Does it scroll faster? Try reducing the sleep time to 1 and see what happens. Does it make you feel nauseated?
7. Try using a different image in your parameter tag. Be sure this image is in the same directory as all of your `.class` and `.html` files. Also, be sure to adjust the WIDTH and HEIGHT parameters in the APPLET tag to match the size of the image.

Section 9. Wrapup

Summary

In this tutorial, we have introduced a type of Java program called a Java applet. Unlike a Java application that executes from a command window, an applet is a Java program that runs in a browser or in the appletviewer test utility. Now that you have completed this tutorial, you should have a thorough understanding of the basic features and syntax of an applet, as well as image techniques in your applet. From here, you may want to progress to more Java tutorials. Happy trails!

Resources

Here are some other tutorials for you to try:

* *Introduction to the Java Foundation Classes*
* *Java language essentials*

This article provides a scenario for using Java applets in an educational setting:

* *A Walk in the Park*

Search for free applets from the *Java Boutique* .

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.