6-1-2006

Building a steganography program including how to load, process, and save JPEG and PNG files in java.

Mary F. Courtney

SCHOOL OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

TECHNICAL REPORT

Number 227, June 2006

Building a Steganography Program

Including How to Load, Process, and Save JPEG and PNG files in Java

Mary F. Courtney
Allen Stix

PACE UNIVERSITY
IVAN G. SEIDENBERG
SCHOOL OF COMPUTER SCIENCE
AND INFORMATION SYSTEMS

The present paper was published in

Mathematics and Computer Education
Winter 2006; Volume 40, Number 1; Pages 19-35

Mary F. Courtney is Professor of Computer Science at Pace University, based in Westchester, where she has been teaching courses in programming, algorithms, and graphics for over 25 years.

Professor Courtney holds a B.A. in Mathematics from the College of Mount Saint Vincent, an M.A. in Mathematics from Lehman College, an M.S. in Computer Science from Pace University, and an Ed.D. in the College Teaching of Mathematics with an emphasis in Computer Science from Teachers College, Columbia University.

Allen Stix is Associate Professor in the Computer Science Department at Pace University, based in Westchester.

BUILDING A STEGANOGRAPHY PROGRAM INCLUDING HOW TO LOAD, PROCESS, AND SAVE JPEG AND PNG FILES IN JAVA

Mary F. Courtney and Allen Stix
Department of Computer Science
Pace University
Pleasantville, New York 10570-2799
mcourtney@pace.edu, astix@pace.edu

ABSTRACT

Instructors teaching beginning programming classes are often interested in exercises that involve processing photographs (i.e., files stored as .jpeg). They may wish to offer activities such as color inversion, the color manipulation effects achieved with pixel thresholding, or steganography, all of which Stevenson *et al.* [4] assert are sought by students. However, instructors may not find textual support for these topics since widely used introductory textbooks do not illustrate the needed techniques, and the professional Java literature obscures these straightforward operations in elaborate discussions. This paper presents the code to make image processing accessible through fragments used to hide a text message in a read-in JPEG file, store the image as a PNG file, and extract the message from the stored image.

INTRODUCTION

Computer Science 1 (CS1) and Computer Science 2 (CS2) instructors report that today's students are motivated more by assignments involving graphics, graphical user interfaces, and image processing than by traditional exercises involving numbers, strings, and data structures. To this end, graphics may be used to teach about methods, derivation, and polymorphism. Digressions into Graphical User Interfaces (GUIs) may be used to spice up learning about more general and pertinent concepts. However, teachers may shy away from image processing because of the difficulty of reading-in an image, displaying the image, converting the image to an array that may be manipulated, converting that array back to an image, and finally saving this new image. Yet articles advocating steganography as an assignment [2], [4] so intrigued us that we were motivated to spend a good deal of time studying the manuals.

This article shows the code, in Java 1.4, for accomplishing all the

actions that we found interesting. The segments are presented, cookbook fashion, with documentation but a minimum of explanation. The ensconcing context is a program for embedding a text message within an image read-in as a .jpeg file (which could just as well have been a .png file) and storing the image as a .png file. Also included are reading-in the stored .png file and extracting the message. In other words, the context is a system for steganography. Space does not allow the entirety of the code to be given here, but the full code may be obtained from the authors. As teachers, we were impressed with the potential of building a steganography system as a project for students nearing the end of CS1 or starting CS2. Such a project would offer tremendous opportunities to exercise problem-solving skills as well as a great glimpse into the problems associated with building and debugging a software system larger than the typical assignment. En route, students would learn about the RGB color model and the operators for the bitwise AND, the bitwise OR, the left shift, and the right shift, and would use some new APIs from Java and simple instances of the try-catch clause. Yet, for all this, the only presupposed experience is with the basic procedural constructs, one-dimensional arrays, and building a class with instance variables, constructors, and instance methods.

BACKGROUND: STEGANOGRAPHY, IMAGE FILES, AND THE RGB COLOR MODEL

Steganography, i. e., message hiding, predates electronic computation by over two thousand years. The ancient Greeks tattooed messages on the heads of messengers who were dispatched after their hair grew back. Later, invisible ink was used to embed secret text within letters [5]. In computing, the bits comprising a text are scattered within a picture in such a way that their presence is imperceptible to the eye. Steganography in computing is usually encountered in data assurance classes in conjunction with cryptography. The intent of steganography is to conceal the presence of a message. The intent of encryption is to scramble a message so that it is indecipherable to all but the intended recipients. Good explanations of digital steganography may be found in [2], [4], and [7].

Because steganographic applications read and write image files, the different storage formats become a practical issue. Photographs, as from a digital camera, are customarily stored as JPEGs. JPEG stands for Joint Photographic Experts Group. This storage format allows for over 16 million different colors, and files in this format are found with the extensions .jpeg, .jpe, and .jpg. Clip art and line drawings are customarily

stored as GIFs. GIF stands for Graphics Interchange Format. This format allows for 256 colors, and files in this format have the extension .gif. PNG stands for Portable Network Graphics (and is pronounced "ping" as in ping-pong). The JPEG and GIF formats emerged around 1990. The first version of the PNG format appeared in 1996 and version 1.1 (the current version) appeared in 1998. The PNG format is more versatile than the other two, but the trade-offs complicate discussions of best choice. An excellent practical introduction and comparison is given in [1].

The advantage of working at the high level of abstraction afforded by the application programming interface of standard Java is that the formatting details are non-issues. As our students say, all you have to know is how to call a method on an object. Java versions 1.4 and up read JPEGs, PNGs, and GIFs using the same objects, methods, and procedures. Likewise, they store JPEGs and PNGs with the same code. It is crucial that files with a hidden message, for the steganography system present here, be saved as PNGs. This is because PNG compression is lossless while JPEG compression is lossy. What is lost are the visually insignificant bits that hold the embedded message.

Each individually addressable spot on the grid making up the display screen is a pixel. Every pixel is composed of a red sub-pixel, a green sub-pixel, and a blue sub-pixel, and the brightness of the light emitted by each may be set to an intensity within the range of 0 to 255 inclusive. The intensities of the sub-pixels determine the pixel's color. The RGB (red, green, and blue) intensities of {135, 206, 235} give skyblue [7]. The brightest pure yellow is {255, 255, 0}. When all three sub-pixels are at an intensity of 0 the color is black. When all three are at 255 the color is white. Between black and white are the grays, produced when the intensities of the sub-pixels are equal.

The binary representation of 255, the setting for maximum intensity, requires eight bits: {1111, 1111}. A group of eight bits is a byte. Thus, for every pixel, three bytes of storage are required by the RGB color model. Notice that when we represented the bit-string, for readability we separated the high-order half-byte or ("nibble") from the low-order nibble with a comma.

Steganographic message hiding takes advantage of the limits of human thresholds for color perception. For instance, a pixel consisting of {133, 208, 236} would almost certainly be indistinguishable from a pixel at the defined value for skyblue, {135, 206, 235}. Processing works by distributing the bits in the ASCII representation of characters over the low-order bits in a succession of sub-pixels. For instance, the ASCII value of 'A' is 65, {0100, 0001}. Each of these bits, respectively, may be

inserted as the lowest-order bit in a succession of eight sub-pixels. Similarly, these bits may be taken in pairs, replacing the two lowest-order bits in four sub-pixels. For the present demonstration of steganography, we worked in units of nibbles: the high-order nibble of a character was moved into the low-order nibble of a sub-pixel, and the low-order nibble of this character was moved into the lower-order nibble of the next sub-pixel.

AN ACCESSIBLE, LARGE-SIZED PROJECT OFFERING MANY LESSONS

As we wrote the code for our steganography system we were impressed with its potential to serve as a substantial project which tied together everything from CS1. It requires much problem-solving of many types.

You can encourage diagrams to guide the development of code for analyzing procedural processes (e.g., distributing the bits from a character's byte into the eight-bit segments of ints storing a pixel's R, G, B values). At the design level, students will need to decide whether to mark the end of the embedded message with a sentinel, begin the message with a character count, or call the system a prototype and simply display a fixed number of characters. If this last alternative is chosen and more characters are extracted than were in the message, trouble can result with "garbage" which is non-printable (i.e., values in the collating sequence of less than 32, which is the space). Another fundamental design decision is the format of storage of the bits from the message. If the message is stored one bit per sub-pixel, eight sub-pixels are needed to store each character. With three sub-pixels per pixel (a red, a green, and a blue), this puts one character into $2\frac{2}{3}$ pixels. Should the remaining sub-pixel be left unused, giving a clean packaging of one character per clump of three pixels, or should storage of the next character start right there? More elaborate formats could be devised than using the sub-pixels in consecutive pixels. Furthermore, the message's embedding does not need to begin with the pixel in the image's first row and first column. An interesting experiment could be to display the original image and the message-laden image side-by-side to determine the amount of perceptible distortion. This could be enhanced with a discussion of its mathematical description and considerations of steganalysis.

Object-oriented design can be made more or less prominent depending upon the students' readiness to consider deriving the responsibilities of "server" objects and the needs of "client" objects. A server that comes to mind immediately is fashioned on the idea of the

StringTokenizer. Objects would be instantiated with the message, and each call to the nextClump() method would return the next bit (or sequence of bits) to be placed within the succeeding sub-pixel. We wrote nextClump() to return a String holding a nibble (that is, four bits) such as "0100". A fancy MessageTokenizer might have a constructor that accepts both the message and an int specifying the length of clumps.

This project could also provide insight into the area of software testing. Students could learn to construct drivers to verify the functional correctness of individual modules instead of waiting until the entire program is done which is the wrong approach. They would get practice following the values of variables. Writing utilities to help with testing, such as a method for displaying ints in a binary format such as {0000 0000, 0110 1101, 1001 0111, 0100 0011} or a utility to display bytes, would be recognized as a time-saver.

This project, with a collection of parts that may be built and tested individually, introduces the need for the methodical organization of software development. It exemplifies the problems addressed by software engineering which are not evident in a short program written to illustrate a single construct or for the sake of a narrow piece of problem-solving.

Finally, there is the issue of the system's delivery, which is up to you as the instructor. It could be presented to hypothetical users as an application programming interface for use in a console environment, or it could be presented in with a graphical user interface. Incidentally, it is not pretentious to term this program a *system* inasmuch as it has interdependent parts similar to any other kind of information storage and retrieval system.

THE ESSENTIAL PARTS OF THE PROGRAM

The remainder of this paper presents the central segments of the steganography program that we wrote. While the complete program is not present, every operation is exemplified. We make no claims about the elegance of our logical constructions nor any other aspect of our code. Clarity and correctness were our chief concerns.

READING AND DISPLAYING AN IMAGE

What follows is a complete program for reading-in and displaying a JPEG file. The program is comprised of the two classes, Main1 and ImageHolder, each in a file of its own. For the program to work, the JPEG file must be in the same directory as the program's files, or its complete path must be specified. Remember that when paths are given as

String literals within code, double backslashes are needed. The file being displayed is WaterLilies.jpg.

Images inside a Java program need to be stored in an instance of class descending from java.awt.Container such as a class declared to extend java.awt.Canvas. The Canvas class is commonly subclassed to display images and drawings. Nothing in this short program that reads-in and displays an image is superfluous. The comments marking particular locations indicate where statements will be inserted in the enhanced version, given next, in which the read-in image is stored within an int array for processing. Images from PNG and GIF files may be read-in with no modification apart from the file's name.

```
//file: Main1.java
import javax.swing.JFrame;
import java.awt.Container;

class Main1
 {
  public static void main (String[] args)
   {
    JFrame frame = new JFrame ();
    frame.setSize(500,450);
    frame.setVisible(true);
    frame. setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    String fileName = "WaterLilies.jpg" ; //Name of file holding image

    Container contentPane = frame.getContentPane();
    contentPane.setLayout(null);

    ImageHolder picture = new ImageHolder(fileName);
    picture.setBounds(10, 10, 450, 400);
    contentPane.add(picture);

    //Point 1   <========
   }
 }

//file:  ImageHolder.java
import java.awt.Canvas;
import java.awt.Graphics;
import java.awt.MediaTracker;
import java.awt.Image;
import java.awt.Toolkit;

//Point 2   <========

class ImageHolder extends Canvas
 {
  Image image = null;
```

```
  ImageHolder(String fileName)  //constructor
   {
    MediaTracker tracker = new MediaTracker(this);
    image = Toolkit.getDefaultToolkit().getImage(fileName);
    tracker.addImage(image, 0);

    try
     {
      tracker.waitForID(0); // throws a checked Exception
     }
    catch(InterruptedException e)
     {
     }
   }
  public void paint(Graphics g)
  {
    g.drawImage(image, 0, 0, this);
  }

  //Point 3   <========
}
```

STORING THE IMAGE AS AN ARRAY

A java.awt.image.PixelGrabber object copies an image into an int array, putting one pixel from the image into each int element. Pixels are copied in a row-major fashion. First, the top row of pixels from the image is copied, going left-to-right. Then, the second row of pixels is copied, going left-to-right, and so on. The pixel on the left-end of the top row goes into the element with subscript 0. If the image were w pixels wide and h pixels high, it would be comprised of w*h pixels, and the array's length would be w*h.

An int in Java is comprised of 32 bits. Alternately, we could say that an int in Java is comprised of four bytes. (In Java, a variable of type byte is eight bits. When interpreted numerically, the byte's leftmost bit is construed as its sign.) Looking at the four consecutive clumps of bytes that constitute an int from left to right, relative to the storage of a pixel, the first byte is unused. The second byte (bits 9 through 16) stores the value of the red sub-pixel. The third byte (bits 17 through 24) stores the value of the green sub-pixel. And the fourth byte (bits 25 through 32) stores the value of the blue sub-pixel.

Three additions to the program above are needed to create this array and copy-in the image's representation so that it is available for manipulation:

The comment labeled Point 1 is at the bottom of the main() method

inside the class Main1. At that point, install the following statement. It activates the method to be added to the ImageHolder class that returns the array. While the present program does nothing with this array, you could add a statement that displays its length.

```
int[] intArrayWithPixels = picture.getImageAsIntArray();
```

The comment labeled Point 2 is inside the ImageHolder class. At the comment labeled Point 2, install this:

```
import java.awt.image.PixelGrabber;
```

At the comment labeled Point 3, inside the ImageHolder class, install the following method:

```
int[] getImageAsIntArray()
 {
  int imageWidth  = image.getWidth(this);
  int imageHeight = image.getHeight(this);

  int[] imageGrabbed = new int[imageWidth * imageHeight];

  PixelGrabber grabber = new PixelGrabber(image, 0, 0,
      imageWidth, imageHeight, imageGrabbed, 0, imageWidth);
  try
   {
    grabber.grabPixels();  //throws a checked Exception
   }
  catch(InterruptedException e)
   {
   }
  return imageGrabbed;
 }
```

CONVERTING THE ARRAY BACK TO AN IMAGE AND SAVING THE IMAGE

The following program, comprised of Main2 and an enhanced version of ImageHolder, provides the techniques for capturing images, creating the int array that may be used to manipulate their pixels, and for saving the array as an image. These techniques will create accessible steganography exercises as well as exercises involving image inversion, thresholding, linear mapping, and histogram computation as described by Hunt [3, p. 88]. To keep the code unencumbered, this program does not create a JFrame for the sake of displaying the image. To view the saved image, use either *Windows Explorer* or *MyComputer* to access the folder.

The image is saved as a PNG file because PNG compression is lossless, which we need for steganography. With lossy compression, as

when saving an image as a JPEG, the low-order bits in sub-pixels are sacrificed. These are the ones representing embedded content.

However, in case you want to save an image in JPEG format, the change in the code is minimal. Replace the png with jpg in these two statements copied from the program immediately below:

```
File file = new File ("savedImage.png");
ImageIO.write(image, "png", file);
```

The ImageHolder class is augmented with "get" methods that return the image's height and width.

```
//file: Main2.java
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

class Main2
 {
  public static void main ( String [] args)
   {
    String fileName = "WaterLilies.jpg" ;
    ImageHolder originalPicture = new ImageHolder(fileName);

    int width = originalPicture.getWidth();
    int height = originalPicture.getHeight();

    originalPicture.setBounds(10, 10, width, height);

    int[] imageArray = originalPicture.getImageAs_int_Array();

    BufferedImage image =
        new BufferedImage (width, height, BufferedImage.TYPE_INT_RGB);
    image.setRGB(0, 0, width, height, imageArray, 0, width);
    try
     {
      System.out.println("Saving to savedImage.png" );
      File file = new File("savedImage.png" );
      ImageIO.write(image, "png" , file);
     }
    catch(IOException e)
     {
     }
  }
 }
//file: ImageHolder.java
import java.awt.Canvas;
import java.awt.Graphics;
import java.awt.MediaTracker;
import java.awt.Image;
```

```
import java.awt.Toolkit;
import java.awt.image.PixelGrabber;

class ImageHolder extends Canvas          //version 3
 {
  Image image;

  ImageHolder(String fileName)  //constructor
   {
    MediaTracker mediaTracker = new MediaTracker(this);
    image = Toolkit.getDefaultToolkit().getImage(fileName);
    mediaTracker.addImage(image, 0);

    try
     {
      mediaTracker.waitForID(0); // throws a checked Exception
     }
    catch(InterruptedException e)
     {
     }
   }

  int[] getImageAs_int_Array()
   {
    int  imageWidth  = image.getWidth(this);
    int  imageHeight = image.getHeight(this);
    int[] imageGrabbed = new int[imageWidth * imageHeight];

    PixelGrabber pixelGrabber = new PixelGrabber(image, 0, 0,
       imageWidth, imageHeight, imageGrabbed, 0, imageWidth);
    try
     {
      pixelGrabber.grabPixels();  //throws a checked Exception
     }
    catch(InterruptedException e)
     {
     }

    return imageGrabbed;
   }

  public int getHeight()   //public because overrides a public method
   {
    return image.getHeight(this);
   }

  public int getWidth()   //public because overrides a public method
   {
    return image.getWidth(this);
   }
}
```

EMBEDDING THE MESSAGE

Our approach to embedding the message is to copy the characters, nibble by nibble, into the low-order nibble of consecutive sub-pixels. The coding constructs are illustrated by the following method, installRed(). Its argument is the subscript of the element in the array holding the current pixel, and it gets the current nibble from the message by a call on a specialized object fashioned to work like a StringTokenizer. Its action is to overwrite the four low-order bits of the red sub-pixel with the four bits from the message. The red sub-pixel is in the second clump of 8 bits, starting from the left, in the int from the array. The first clump of 8 bits is unused in representing the image, so it is ignored.

Within installRed(), the first task is to create and use a mask for replacing the four low-order bits of the red sub-pixel with zeros. The next task is to get a nibble (half a character) from the message with a call on a MessageTokenizer object to nextNibble(). For the character 'A', that has an ASCII value of 65 (which is 0100 0001 in binary), the first call to nextNibble() returns 4 (which is 0000 0100) and the next call returns 1 (which is 0001). This nibble is re-expressed in binary as a String of 0s and 1s embedded within a 24-character String placed to be aligned with the red sub-pixel's low-order bits. In other words, this nibble will be the 17th through the 20th characters in a String with 0s in all other places. The 24-character String is converted to an int, and this int is ORed with the int storing the pixel. ORing like this replaces the 0s masked into the red sub-pixel with the nibble from the message but does not change any of the other pixel's other bits.

The steganography program we wrote includes an installGreen() method and an installBlue() method that are similar in function and construction to installRed().

```
//method overwrites the low order bits in the red sub-pixel
//with a nibble from the message

void installRed(int subscriptIntoPixelHolderArray)
 {
 Integer intObj = Integer.valueOf("111100001111111111111111" , 2);
 int firstMask = intObj.intValue();

 //puts 0000 in the low order nibble of the red subpixel
 image[subscriptIntoPixelHolderArray] =
          image[subscriptIntoPixelHolderArray] & firstMask;

 byte nibble = tokenizer.nextNibble();
 String nibbleAs0s_and_1s = convert_byte_ToBinaryString(nibble);

 String nibbleAligned =
```

```
            "0000"  + nibblesAs0s_and_1s + "0000000000000000" ;
  intObj = Integer.valueOf (nibbleAligned,2);
  int secondMask = intObj.intValue();
  image[subscriptIntoPixelHolderArray] =
            image[subscriptIntoPixelHolderArray]  |  secondMask;
 }
String convert_byte_ToBinaryString(byte b)
 {
  Byte ByteObj = new Byte(b);
  int x = ByteObj.intValue();
  String str = Integer.toBinaryString(x);
  return padWithLeadingZeros(str);
 }
String padWithLeadingZeros(String str)  //brings str to a length of 4
 {
  int originalLength = str.length();
  for (int j = 0;  j < (4 - originalLength);  j++)
    {
     str = "0"  + str;
    }
  return str;
 }
```

The MessageTokenizer class is below. The String holding the message is passed to the constructor. As all Strings in Java, it is a sequence of characters in Unicode, which means that each character is stored in two bytes (16 bits). Because the Unicode collating sequence is the same as the ASCII collating sequence for regular letters, numerals, and special symbols, we only need to be concerned with values small enough to be stored in a byte. Accordingly, as an intermediate step, the message is put into an array of bytes (messageInBytes). Each element of this array is split in half, bitwise, and the respective halves are stored as two elements in an array called nibbles. The four leftmost bits of the value from messageInBytes go into nibbles[i] and the four rightmost bits go into nibbles[i+1]. To get the four leftmost bits, the original byte is right shifted by four bits. To get the four rightmost bits, the original byte is ANDed with 15 (which is 0000 1111).

The nextNibble() method, similar to the StringTokenizer's nextToken() method, returns the subsequent value in the nibbles array. To do this, the MessageTokenizer object uses a private instance variable called cursor that keeps track of its place.

```
//file:  MessageTokenizer.java

class MessageTokenizer
```

```
{
 byte[] nibbles;
 int cursor = -1;
 MessageTokenizer(String message)  //constructor
  {
   nibbles = new byte [message.length() * 2];
   byte[] messageInBytes = message.getBytes();
   int nibbleCount =0;
   for(int i = 0;  i <messageInBytes.length;  i++)
    {
     nibbles[nibbleCount] =
        (byte)(messageInBytes[i] >> 4); //right shift of 4 bits
                                        //to get the left hand bits
     nibbleCount++;
     nibbles[nibbleCount] =
        (byte)(messageInBytes[i] & 15);  //masking off the left bits
                                         //to get the right hand bits
     nibbleCount ++;
    }
 }
 byte nextNibble()
 {
  if ((cursor + 1) < nibbles.length)
   {
    cursor ++;
    return  nibbles[cursor];
   }
  else
   return  (byte) ' ';
 }
}
```

EXTRACTING THE MESSAGE

The first step is to retrieve the image with the embedded message and to recover the message as an int array using the services of a java.awt.image.PixelGrabber object. This may be done with the same ImageHolder class built for embedding the message.

The logic for recovering the message needs to reverse that of embedding the message. Consecutive sub-pixels are used in embedding, skipping none. This design maximizes the space available in the image but makes for the awkward three-to-two storage of characters across pixels. The nibbles of every other character are stored across two pixels, meaning in the blue sub-pixel in one element of the array and in the red sub-pixel within the next.

The message is recovered in the constructor of our MessageExtractor. The centerpiece is a while loop, each iteration of which processes two elements from the imageGrabbed array and concatenates three characters onto the String constituting the extractedMessage. A simple 20 elements of the array are processed, regardless of the message's actual length. This is fine for the purpose of "prototyping" or "proof of concept," but would not fit the bill for a finished product.

Specialized methods called extractRed(), extractGreen(), and extractBlue() pull the values of nibbles from respective sub-pixels out of an imageGrabbed element. These methods work similarly, with a mask and, for the nibbles in red and green sub-pixels, a right shift. To exemplify, extractRed() is shown.

A character's high nibble and low nibble are put together by left-shifting the high, then ORing it with the low. The reason for testing an assembled character to be within the range of the space (ASCII 32) and ASCII 127 before appending it to the extractedMessage is to be sure it is a normal, printable character. Because the while loop may process beyond the last character of the message, bogus values may be encountered. Certain values designate carriage control (e.g., backspace) and could perturb the extracted message, mimicking a logical fault.

```
//file: MessageExtractor.java

class MessageExtractor
{
  int[] imageGrabbed;
  String assembledMessage= "" ;

  MessageExtractor(String name_of_PNG_file)
  {
    ImageHolder picture = new ImageHolder(name_of_PNG_file);
    imageGrabbed = picture.getImageAs_int_Array();

    int cursor = 0;  //subscript of current element in imageGrabbed

    while(cursor < 20)  //20 elements stores 30 characters
    {
      //extracting the first character
      int high = extractRed(cursor);
      int low  = extractGreen(cursor);

      byte together = (byte)((high<<4) | low);
      char nextChar = (char)together;

      if (nextChar >= ' ' && nextChar <= (char)127)
          assembledMessage = assembledMessage + nextChar;

      //extracting the second character
```

```
        high = extractBlue(cursor);
        cursor++;              //on to next element of imageGrabbed
        low = extractRed(cursor);

        together = (byte)((high <<4) | low);
        nextChar = (char)together;

        if (nextChar >= ' ' && nextChar <= (char)127)
            assembledMessage = assembledMessage + nextChar;

        //extracting the third character
        high = extractGreen (cursor);
        low  = extractBlue(cursor);

        together = (byte)(high <<4 | low);
        nextChar = (char)together;

        if (nextChar >= ' ' && nextChar <= (char)127)
            assembledMessage = assembledMessage + nextChar;

        cursor++;              //on to next element of imageGrabbed
      }

     System.out.println(" The message is "   + assembledMessage);
   }
  int extractRed(int cursor)
   {
    //create a mask for extracting the message nibble
    //in the lower four bits of the red sub-pixel
    Integer intObj = Integer.valueOf("00001111000000000000000"  , 2);
    int mask = intObj.intValue();

    //extract the nibble
    int messageNibble = imageGrabbed[cursor] & mask;

    //move the nibble to the right-end of the int
    messageNibble = messageNibble >> 16;

    return messageNibble;
   }

  //  int extractGreen(int cursor)  goes here
  //  int extractBlue(int cursor)   goes here

}
```

CONCLUSION

The given code uses a built-in facility for performing binary-to-integer conversions. It also uses a facility that represents a captured JPEG as a straightforward int array and another that turns the array back into an image. Programming is increasingly taking place in an environment of abstractions, and it is interesting that abstractions are provided both for

operations that used to be considered part and parcel of fundamental programming as well as for operations that are extremely elaborate.

Some instructors view the latter abstractions as legitimate but feel that the former are depriving modern-day students of essential learning experiences. We disagree. It has always been the job of high-level programming languages to provide the necessary tools for the *ad hoc* problem domain. FORTRAN provided operators for floating point arithmetic, facilities for input and output, and a square-root facility. COBOL made file-processing easy and made dollar and cents arithmetic exact with binary coded decimal. APL supplied abstractions for manipulating vectors and matrices, and SNOBOL supplied abstractions for manipulating strings. If Java gives us more, it is for two reasons: (i) It is possible to supply more and (ii) The age of graphical user interfaces, multimedia, and the Internet requires more.

Programming, however, has not been reduced only to calling methods on objects, as the forgoing exercise should demonstrate. Nor have patterns and object-oriented design obsoleted logical problem-solving. Problem-solving and the traditional issues of efficient and robust design remain central.

Quoting Bill Gates [8], "In a certain sense...the curriculum has changed, but say somebody came for an interview and they said, 'Hey, I read the *Art of Computer Programming*, that's all I ever read, and I did all the problems,' I would hire them right then...even if they didn't do the double-star problems...just the fact that they'd read the whole book.... Those are the kinds of things you need to know to be a good programmer...the kinds of algorithmic thinking that's promoted there."

Continuing, Mr. Gates says of a computer science curriculum suitable for today, "So, in a sense, we want to teach the same thing, but we'd like to teach it in a forum that's most interesting.... But what you're really teaching the person about is pretty much the same as what you wanted to teach them 30 years ago." Yet it is pedagogically important to "make sure that the person learning those things feels like they're doing something very cool and very interesting."

REFERENCES

1. B. Crispen, and K. Crispen, *GIF, JPEG, or PNG* (2004). http://toolkit.crispen.org/formats/index.html .
2. K. Hunt, "A Java Framework for Experimentation with Steganography", Proceedings of the Thirty-Sixth Technical Symposium on Computer Science Education, *ACM SIGCSE Bulletin*, Volume 37, No. 1, pp. 282-286, ISSN: 0097-8418 (2005).

3. K. Hunt, "Using Image Processing to Teach CS1 and CS2", *ACM SIGCSE Bulletin*, Volume 35, No. 4, pp. 86-89, ISSN: 0097-8418 (2003).
4. D. E. Stevenson, M. R. Wick, and S. J. Ratering, "Streganography and Cartography: Interesting Assignments that Reinforce Machine Representation, Bit Manipulation, and Discrete Structures Concepts", Proceedings of the Thirty-Sixth Technical Symposium on Computer Science Education, *ACM SIGCSE Bulletin*, Volume 37, No. 1, pp. 277-281, ISSN: 0097-8418 (2005).
5. B. Schneier, *Secrets and Lies: Digital Security in a Networked World*, Wiley Publishing, Inc., Indianapolis, Indiana, pp. 245-246, ISBN: 0-471-45380-3 (2000).
6. K. J. Walsh, *RGB to Color Name Mapping (Triplet and Hex)*, 2005, http://eies.njit.edu/~kevin/rgb.txt.html and http://web.njit.edu/~walsh/rgb.html.
7. H. Wang and S. Wang, "Cyber Warfare: Steganography vs. Steganalysis", *Communications of the ACM*, Vol. 47, No. 10, pp. 76-82, ISSN: 0001-0782 (2004).
8. Remarks by Bill Gates, Chairman and Chief Software Architect, Microsoft Corporation, and Maria Klawe, Dean of Engineering and Applied Science, Princeton University, at the Microsoft Research Faculty Summit 2005 in Redmond, Washington on July 18, 2005, http://www.microsoft.com/billgates/speeches/2005/07-18FacultySummit.asp .