Understanding the Huffman Data Compression Algorithm in Java

Learn how to write a Java program that exposes the inner workings of the Huffman lossless data compression algorithm. Apply the algorithm to different test messages.

Published: May 2, 2006
By Richard G. Baldwin

Java Programming Notes # 2442

Preface

This is the second lesson in a series of lessons that will teach you about data and image compression. The series began with the lesson entitled Understanding the Lempel-Ziv Data Compression Algorithm in Java *(commonly known as LZ77).*

Different variations of the LZ algorithms, the Huffman algorithm, and other compression algorithms are often combined in data and image compression programs. For example, numerous sources on the web indicate that commercially available zip programs often incorporate something called **DEFLATE**. According to Wikipedia,

> *"**DEFLATE** is a lossless data compression algorithm that uses a combination of the LZ77 algorithm and Huffman coding. It was originally defined by Phil Katz for version 2 of his PKZIP archiving tool, and was later specified in RFC 1951."*

This lesson will teach you about Huffman coding.

Future lessons will cover a variety of compression schemes, including:

- Run-length data encoding

- GIF image compression
- JPEG image compression

Use at your own risk

The programs that I will provide in these lessons are provided for educational purposes only. If you use these programs for any purpose, you are using them at your own risk. I accept no responsibility for any damages that you may incur as a result of the use of these programs.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings and figures while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

I particularly recommend that you study my earlier lessons in the section entitled References in preparation for understanding the material in this lesson.

General Background Information

Most of us use data or image compression on a daily basis without even thinking about it. If you use one of the popular zip programs to archive your data, you are using a program that typically implements several different data compression algorithms in combination. If you take pictures with a digital camera, you are probably creating files that describe images using the JPEG image compression format.

What is Huffman coding?

According to Wikipedia,

> *"... **Huffman coding** is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol."*

Other variable-length entropy coding systems

Huffman coding is not the only encoding scheme to use variable-length code based on the probability of occurrence of the characters in a message. For example, the International Morse Code, originally created by Samuel Morse in the mid-1830s is a variable-length code, apparently based on his concept of the probability of occurrence of the letters in the English alphabet. For example, the code for the letter E is the shortest code. The letter E occurs very frequently in English text. The longest code is for the number 0. Excluding the numbers, however, the two longest codes are for the letters Q and W. The letter Q doesn't appear very often in English Text.

> *(As an aside, one might surmise that similar probability tables were used to develop the QWERTY keyboard where the key for the letter E is relatively easy to strike. On the other hand, the key for the letter Q is more difficult for many people to strike because it requires the use of the little finger on the left hand.)*

Morse is not a prefix-free code

While Morse code is easy for a trained radio operator to understand *(this author was a radio operator in the U.S. Air Force for several years)*, the code has a characteristic that makes it difficult, or at least inefficient to implement using a computer. In particular, the code sequences for the most probable characters occur as the beginning parts of the code sequences for other characters. *(For example, the entire code sequence for the letter E occurs as the first element in the codes for many other letters.)* A trained human can easily deal with that situation but a significant amount of overhead is required to cause a computer to deal with it.

Huffman is a prefix-free code

Huffman coding solves this problem. The methodology used for Huffman coding results in a prefix-free code. A prefix-free code is one in which the bit coding sequence representing some particular character is never a prefix of the bit coding sequence representing any other character. For example, here is a possible bit sequence for a Huffman code on an alphabet with four characters where D is the most probable and A is the least probable:

```
A 110
D 0
C 10
B 111
```

Code length is based on probability of occurrence

As with Morse code, the methodology used for the Huffman coding causes the bit coding sequence to be shortest for the most frequently occurring characters, and causes the coding sequence to be longest for the least frequently occurring characters. Unlike the Morse code, however, the probabilities for Huffman are usually determined on a message-by-message basis instead of being based on some general overall expectation of the probability of occurrence of the characters.

Message-by-message probabilities

The upside of determining the probabilities on a message-by-message basis is that the encoding can be optimized for each specific message. The downside is that the probability encoding table used to encode a specific message must also be used to decode the message. The requirement to transport the encoding table along with the message adds some overhead to the overall process.

An optimal encoding scheme

Once again, according to Wikipedia,

> *"Huffman was able to design the most efficient compression method of this type: no other mapping of individual source symbols to unique strings of bits will produce a smaller average output size when the actual symbol frequencies agree with those used to create the code."*

Some caution is advised

Wikipedia goes on to caution us,

> *"Assertions of the optimality of Huffman coding should be phrased carefully, because its optimality can sometimes accidentally be over-stated. For example, arithmetic coding ordinarily has better compression capability, because it does not require the use of an integer number of bits for encoding each source symbol. LZW coding can also often be more efficient, particularly when the input symbols are not independently-distributed, because it does not depend on encoding each input symbol one at a time (instead, it batches up a variable number of input symbols into each encoded syntax element). The efficiency of Huffman coding also depends heavily on having a good estimate of the true probability of the value of each input symbol."*

As mentioned earlier, some data compression schemes combine Huffman coding with one of the Lempel-Ziv *(LZ)* coding schemes to get the best of both worlds.

A binary tree

As you will soon see, Huffman coding works by creating a binary tree of nodes, with each node being either a leaf node or an internal node.

> *(A picture of an actual Huffman binary tree, along with a corresponding example, can be seen at the Binary Essence web site. Additional Huffman binary trees are also shown in Figure 6, Figure 8, and Figure 10 later in this lesson.)*

All nodes are initially leaf nodes, and there is one leaf node for every character in the message being compressed. Then the leaf nodes are combined with internal nodes to form the tree.

As mentioned above, here is one leaf node for each character in the message. A leaf node contains the character and the frequency of usage for that character.

Internal nodes contain links to two child nodes plus a *frequency* which is the sum of the frequencies of the two child nodes.

A variable-length bit sequence

A different, variable-length bit sequence is assigned to each character used in the message. The specific bit sequence assigned to an individual character is determined by tracing out the path from the root of the tree to the leaf that represents that character. By convention, bit '0' represents following the left child when tracing out the path and bit '1' represents following the right child when tracing out the path.

Path lengths are different

The tree is constructed such that the paths from the root to the most frequently used characters are short while the paths to less frequently used characters are long. This results in short codes for frequently used characters and long codes for less frequently used characters.

Preview

The program named Huffman01

In this lesson, I will present and explain a program named **Huffman01**, which illustrates the encoding and subsequent decoding of a text message using the Huffman encoding algorithm.

Get and use an encoder object

Overall control of this operation of this program takes place in the **main** method. I will begin by instantiating an object of the **HuffmanEncoder** class and by invoking the **encode** method on that object from the **main** method.

Create a frequency chart

Inside the encode method, I will invoke the **createFreqData** method to create a frequency table that identifies each of the individual characters in the original message and the number of times *(frequency)* that each character appears in the message being compressed.

Create the leaves and construct the tree

Next, I will invoke the **createLeaves** method to create a **HuffLeaf** object for each character identified in the frequency table. I will store the **HuffLeaf** objects in a **TreeSet** object. Each **HuffLeaf** object encapsulates the character that it represents as well as the number of times that the character appears in the original message *(frequency)*.

Then I will invoke the **createHuffTree** method to assemble the **HuffLeaf** objects into a Huffman tree *(a **HuffTree** object)*. A Huffman tree is a special form of a binary tree consisting of properly linked **HuffNode** and **HuffLeaf** objects.

When the **createHuffTree** method returns, the **HuffTree** object will remain as the only object stored in the **TreeSet** object that previously contained all of the **HuffLeaf** objects. This is because all of the **HuffLeaf** objects will have been combined with **HuffNode** objects to form the tree. When two **HuffLeaf** objects are combined with a single **HuffNode** object, the two **HuffLeaf** objects are removed from the **TreeSet** object, and the **HuffNode** object is added to the **TreeSet** object.

Create the bit codes

Following that, I will invoke the **createBitCodes** method, which uses the Huffman tree in a recursive manner to create a bit code for each character in the message.

> *(As mentioned earlier, the bit codes are different lengths with the shorter codes corresponding to the characters with a high frequency value and the longer codes corresponding to the characters with the lower frequency values.)*

The **createBitCodes** method populates a data structure that is used to encode the message and is also required later to decode the encoded message.

Dealing with a difficult bit-manipulation challenge

At this point in the execution of the program, I know the variable-length bit code that is required to replace each character in the original message to produce a Huffman-encoded message.

> *(The compression provided by Huffman encoding depends on the frequently used characters having short bit codes and the less frequently used characters having longer bit codes.)*

Although I know the bit code required to replace each character in the original message, a direct transformation from characters in the message to a stream of contiguous bit codes is something of a challenge. The computer's memory is organized on 8-bit boundaries. I am unaware of any capability in Java that allows the memory to be viewed simply as a continuous sequence of individual bits.

> *(Note that it may be possible to accomplish this by using a Java BitSet object. I may give that a try someday when I have the time.)*

This program deals with this challenge in a way that is straightforward, but is probably inefficient from both a speed and memory requirements viewpoint.

Not a production compression program

This program was specifically designed to serve its primary purpose of education. No thought or effort was given to speed, efficiency, memory utilization, or any other factor that would be important in a program written for production data compression purposes. In some cases, the program was purposely made less efficient *(in the name of clarity)* by using two or more statements to accomplish a task that could be accomplished by a single more complex statement.

The solution to the bit-manipulation challenge

The solution to the bit-manipulation challenge mentioned above was to do a simple table lookup and to create a long **String** object consisting of only 1 and 0 characters. Each character in the original message is represented by a substring that matches the required bit code.

This is easy to accomplish because *(unlike a long sequence of bits)* there are no artificial boundaries requiring the length of the **String** to be some multiple of a fixed number of characters.

I will invoke the **encodeToString** method to encode the message into a **String** representation of the bits that will make up the final encoded message. After the String containing 1 and 0 characters representing the bits in the Huffman-encoded message is created, this **String** will be processed to produce the Huffman-encoded message in a binary bit stream format.

Creating the binary bit stream format

Creation of the Huffman-encoded message in a binary bit stream format is accomplished using another lookup table containing 256 entries *(the number of possible combinations of eight bits)*.

I will invoke the **buildEncodingBitMap** method to populate a lookup table that relates eight bits represented as a **String** to every possible combination of eight actual bits. Then I will invoke the **encodeStringToBits** method to encode the **String** representation of the bits that make up the encoded message into the actual bits that make up the encoded message.

Extraneous characters at the end

The **encodeStringToBits** method doesn't handle the end of the message very gracefully for those cases where the number of required bits is not a multiple of 8. The method simply adds enough "0" characters to the end of the **String** to cause the length to be a multiple of 8. This will usually result in extraneous characters at the end of the decoded message later.

Some mechanism must be found to eliminate the extraneous characters when decoding the message later. This program assumes that the length of the original message is preserved and provided to the decoding software along with the required decoding table. Since the length of the decoded message must match the length of the original message, this value is used to eliminate extraneous characters at the end of the decoded message.

Having created the Huffman-encoded message in a binary bit stream format, I will return the encoded message from the **encode** method back to the **main** method.

Hexadecimal display

At this point in the program, the message has been Huffman encoded. Back in the **main** method, I will provide the capability to display the binary encoded data in Hexadecimal format for comparison with the original message.

Decode the encoded message

The program continues the demonstration by decoding and displaying the Huffman-encoded message.

I will begin the decoding process by instantiating a **HuffmanDecoder** object from within the **main** method. Then I will invoke the **decode** method on the **HuffmanDecoder** object to decode the message.

I will pass the encoded message along with a reference to a data structure containing encoding particulars and the length of the original message to the **decode** method so that extraneous characters on the end can be eliminated.

Decode from binary to String representation

Inside the **decode** method, I will invoke the **buildDecodingBitMap** method to create a decoding bit map, which is essentially the reverse of the encoding bit map that was used to encode the original message.

I will invoke the **decodeToBitsAsString** method to decode the encoded message from a binary bit stream representation to a **String** of 1 and 0 characters that represent the actual bits in the encoded message.

Decode from string representation back to the original characters

I will invoke the **buildHuffDecodingTable** method to create a Huffman decoding table by swapping the keys and the values from the Huffman encoding table received as an incoming parameter by the **decode** method.

Finally, I will invoke the **decodeStringBitsToCharacters** method to decode the **String** containing only 1 and 0 characters that represent the bits in the encoded message. This will produce a replica of the original message that was subjected to Huffman encoding.

Remove extraneous characters and return the decoded message

I will write the resulting decoded message into a **String** object and return the **String** with any extraneous characters at the end having been removed.

Display results at critical points in the process

Numerous opportunities will be provided to enable code that will display information that is useful towards gaining an understanding of the Huffman encoding algorithm. I will present and discuss much of that information in this lesson.

Program testing

The program was tested using J2SE 5.0 and WinXP. The program requires J2SE 5.0 or later to support generics.

Sample messages

Three test messages are hard-coded into the program. You can switch among those messages by enabling and disabling them using comments and by then recompiling the program. You can also insert your own test message and recompile the program to see the result of compressing your message.

Discussion and Sample Code

The class named Huffman01

I will discuss this program in fragments. You can view a complete listing of the program in Listing 45 near the end of the lesson.

The class definition for the class named **Huffman01** along with the **main** method begins in Listing 1.

```
public class Huffman01{

  public static void main(String[] args){

    Hashtable
<Character,String>huffEncodeTable;
```

Listing 1

Listing 1 declares a data structure that is used to communicate encoding particulars from the Huffman encoder to the Huffman decoder. Because the encoding particulars are different for every message, this is necessary for the decoder to be able to decode the encoded message.

Test messages

Listing 2 creates and displays the raw test message that will be encoded. The message is displayed 48 characters to the line.

```
/*
    //The following test message was copied
directly from
```

```
    // an Internet news site.  It is probably
    // representative of typical English text.
    String rawData = "BAGHDAD, Iraq Violence
increased "
    + "across Iraq after a lull following the
Dec. 15 "
    + "parliamentary elections, with at least
two dozen "
    + "people including a U.S. soldier killed
Monday in "
    + "shootings and bombings mostly targeting
the Shiite-"
    + "dominated security services. The Defense
Ministry "
    + "director of operations, Brig. Gen. Abdul
Aziz "
    + "Mohammed-Jassim, blamed increased
violence in the "
    + "past two days on insurgents trying to
deepen the "
    + "political turmoil following the
elections. The "
    + "violence came as three Iraqi opposition
groups "
    + "threatened another wave of protests and
civil "
    + "disobedience if allegations of fraud are
not "
    + "properly investigated.";
*/
/*
    String rawData = "Now is the time for all
good men "
    + "to come to the aid of their country.";
*/

    //Use the following test message or some
other
    // similarly short test message to
illustrate the
    // construction of the HuffTree object.
    String rawData = "AAAAABBBBCCCDDE";

    System.out.println("Raw Data");
    display48(rawData);
```

Listing 2

As mentioned earlier, you can modify the comment indicators to enable any one of the test messages in Listing 2, or you can insert a test message of your own and then recompile the program.

Listing 2 invokes the utility method named **display48** to display the message 48 characters to the line. That method is straightforward and shouldn't require an explanation. You can view the method in its entirety in Listing 45.

Program output

The text at the top of Figure 1 was produced by the code in Listing 2 after modifying the comment indicators to enable the news story shown in Listing 2.

```
Raw Data
BAGHDAD, Iraq Violence increased across Iraq aft
er a lull following the Dec. 15 parliamentary el
ections, with at least two dozen people includin
g a U.S. soldier killed Monday in shootings and
bombings mostly targeting the Shiite-dominated s
ecurity services. The Defense Ministry director
of operations, Brig. Gen. Abdul Aziz Mohammed-Ja
ssim, blamed increased violence in the past two
days on insurgents trying to deepen the politica
l turmoil following the elections. The violence
came as three Iraqi opposition groups threatened
another wave of protests and civil disobedience
if allegations of fraud are not properly invest
igated.

Number raw data bits: 5048
Number binary encoded data bits: 2816
Compression factor: 1.7926136363636365

Binary Encoded Data in Hexadecimal Format
b889531b2812a0668f6b9c2eaf9759dceb2c86d996148
9a3c1ae734124336fabffe82bffa8dcf7e9f4e50563e1a8
778c1fe684bcb0b8fcae7a9d0668dd3ee69df2c93a6356b
54717de1af3e773afeadb9efcdae3c4f92bf6e2186fdff
9b689a7b30bb9f27b553cf7a667b68b5c5e7bd343493f17
4b5e679efd3e9d11ef74911b543ced26d90ad50e91721b
bcb283e52f4e50602e8744f3f248badc057341aa0d5e16
9ea7419ae21d78f94cb8f84c5b2bfc263ba3b44d7ad0c26c
2235644e84668bf684db73acb21b6bbd5f2eb3b9f4fa778
c93a6356b3a9a9f73a2a179794d202f3dfa6b589fbe9f4
ef2bfd3cb6fe95421aefe82bffa8dcf7e9f4e3f2b9ea741f
297a75deaf9759cb684e64d3e813a3c1ae7b579e5274f53e
bc295f134fa05a4b9b667a9f4868cd74ea8378152525333d
b2faef7f5b92a29b71759dd066ffcded3d4e9aa0d032b6c
c073d4ef2bc21f1773dd293d7b49be

Decoded Data
BAGHDAD, Iraq Violence increased across Iraq aft
er a lull following the Dec. 15 parliamentary el
ections, with at least two dozen people includin
g a U.S. soldier killed Monday in shootings and
bombings mostly targeting the Shiite-dominated s
ecurity services. The Defense Ministry director
of operations, Brig. Gen. Abdul Aziz Mohammed-Ja
```

```
ssim, blamed increased violence in the past two
days on insurgents trying to deepen the politica
l turmoil following the elections. The violence
came as three Iraqi opposition groups threatened
another wave of protests and civil disobedience
if allegations of fraud are not properly invest
igated.
```
Figure 1

A compression factor of 1.793

The statistics following the test message in Figure 1 show that the compression factor achieved by the Huffman algorithm for this particular message was 1.793. This can also be thought of as a compression ratio of 0.558. In other words, the compressed message requires 55.8 percent of the number of bits required by the original uncompressed message.

As you will see later, different messages result in different compression factors.

The binary encoded test message

The block of text near the middle of Figure 1 shows the binary version of the encoded message displayed in Hexadecimal format. At the surface, this may appear to be longer than the original message. Recall, however, that in the display of the original message at the top of Figure 1, each character represents eight bits. However, in the Hexadecimal display, each character represents only four bits. As mentioned above, the number of actual bits in the compressed message was only 55.8 percent of the number of bits in the original message.

The decoded message

The bottom block of text in Figure 1 is a replica of the original message that was produced by decoding the binary version of the encoded message. Hopefully, this is an exact copy of the original message at the top of Figure 1, which is a requirement for a lossless compression algorithm.

Display raw data length

Listing 3 gets and displays the length of the test message in bits, as shown in the upper half of Figure 1.

```
    int rawDataLen = rawData.length();

    System.out.println("\nNumber raw data bits:
"
                                    +
rawData.length() * 8);
```

Listing 3

Instantiate a HuffmanEncoder object

Listing 4 instantiates a new object of the **HuffmanEncoder** class. The instance method named **encode** belonging to that object will be used to encode the test message using the Huffman compression algorithm.

```
    HuffmanEncoder encoder = new
HuffmanEncoder();
```

Listing 4

Encode the Message

Listing 5 begins by instantiating a new **Hashtable** object that will be passed to the **encode** method to be populated with encoding particulars. This object will be used to encode the message and will also be required later to decode the message.

Still in the **main** method, Listing 5 invokes the **encode** method of the **HuffmanEncoder** object to perform the actual encoding. *(Control is transferred from the **main** method to the **encode** method.)*

The test message and the **Hashtable** mentioned above are passed as parameters to the **encode** method.

```
    huffEncodeTable = new
Hashtable<Character,String>();

    ArrayList<Byte> binaryEncodedData =
encoder.encode(

rawData,huffEncodeTable);
```

Listing 5

The encoded message is received back later as bytes stored in an **ArrayList** object.

The HuffmanEncoder class

At this point, I am going to set the **main** method aside and explain the **encode** method of the **HuffmanEncoder** class. I will return to the discussion of the **main** method later.

The **HuffmanEncoder** class definition begins in Listing 6. An object of this class can be used to encode a message using the Huffman encoding algorithm.

```

class HuffmanEncoder{
  String rawData;
  TreeSet <HuffTree>theTree = new
```

```
TreeSet<HuffTree>();
  ArrayList <Byte>binaryEncodedData =
                                     new
ArrayList<Byte>();
  Hashtable <Character,Integer>frequencyData =
                        new
Hashtable<Character,Integer>();
  StringBuffer code = new StringBuffer();
  Hashtable <Character,String>huffEncodeTable;
  String stringEncodedData;
  Hashtable <String,Byte>encodingBitMap =
                              new
Hashtable<String,Byte>();

Listing 6
```

Declaration of instance variables

Listing 6 contains the declaration of several instance variables along with the initialization of some of them. I will discuss the instance variables in conjunction with the code that uses them later.

> *(By the way, in case you are unfamiliar with the syntax of the boldface statement in Listing 6 that declares a reference to and instantiates a new* ***TreeSet*** *object, see my earlier lesson entitled Generics in J2SE 5.0, Getting Started.)*

The encode method

The **encode** method begins in Listing 7.

```
  ArrayList<Byte> encode(
              String rawData,
              Hashtable
<Character,String>huffEncodeTable){
    //Save the incoming parameters.
    this.rawData = rawData;
    this.huffEncodeTable = huffEncodeTable;

Listing 7
```

The **encode** method encodes an incoming **String** message using the Huffman encoding algorithm. The method also receives a reference to an empty data structure of type **Hashtable**. This data structures is populated with encoding particulars. These encoding particulars are used to encode the message. They are also required later by the **decode** method to decode and transform the encoded message back into the original **String** version.

In order to keep this method simple, pad characters may be appended onto the end of the original message when it is encoded. This is done to cause the number of bits in the encoded message to be a multiple of eight, thus causing the length of the encoded message to be an integral number of bytes. Additional code would be required to avoid this at this point. However, it is easy to eliminate the extraneous characters during decoding if the length of the original message is known.

The code in Listing 7 saves the incoming parameters in a pair of local variables.

Display original message as bits

Listing 8 shows the first of several opportunities throughout this program to remove comment indicators and cause information of interest be displayed.

```
/*
    System.out.println("\nRaw Data as Bits");
    displayRawDataAsBits();
*/
```

Listing 8

By removing the comment indicators to enable the two statements shown in Listing 8, you can display the original message as a stream of bits. This can be visually compared with a similar display for the encoded message later to illustrate the amount of compression provided by the encoding process.

Display the message as a stream of bits

The bottom portion of Figure 2 shows the result of enabling these two statements and running the program using the test message shown at the top of Figure 2. *(This test message is much shorter than the test message from Figure 1.)*

```
Raw Data
Now is the time for all good men to come to the
aid of their country.

Number raw data bits: 552

Raw Data as Bits

010011100110111101110111001000000110100101110011
001000000111010001101000011001010010000001110100
011010010110110101100101001000000110011001101111
011100100010000001100001011011000110110000100000
011001110110111101101111011001000010000001101101
011001010110111000100000011101000110111100100000
011000110110111101101101011001010010000001110100
011011110010000001110100011010000110010100100000
011000010110100101100100001000000110111101100110
001000000111010001101000011001010110100101110010
001000000110001101101111011101010110111001110100
011100100111100100101110
```

Figure 2

The method named displayRawDataAsBits

The utility method named **displayRawDataAsBits**, which is invoked in Listing 8, is relatively straightforward and shouldn't require an explanation. You can view the method in its entirety in Listing 45.

Create a frequency chart

Listing 9 invokes the method named **createFreqData** to create a frequency chart that identifies each of the individual characters in the original message and the number of times *(frequency)* that each character appears in the message.

```
    createFreqData();
    //For illustration purposes only, enable 
the following
    // statement to display the contents of the 
frequency
    // chart created above.
/*
    displayFreqData();
*/
```

Listing 9

Listing 9 also optionally invokes the method named **displayFreqData** to display the results.

The createFreqData method

Listing 10 shows the **createFreqData** in its entirety.

```
  void createFreqData(){
    for(int cnt = 0;cnt < 
rawData.length();cnt++){
      char key = rawData.charAt(cnt);
      if(frequencyData.containsKey(key)){
        int value = frequencyData.get(key);
        value += 1;
        frequencyData.put(key,value);
      }else{
        frequencyData.put(key,1);
      }//end else
    }//end for loop
  }//end createFreqData
```

Listing 10

The **createFreqData** method creates the frequency chart described above. The results are stored in a **Hashtable** with the characters being the *keys* and the usage frequency values of each character being the corresponding **Hashtable** *values* for those *key*.

The code in Listing 10 is relatively straightforward and shouldn't require a detailed explanation.

The displayFreqData method

The method named **displayFreqData** that is called in Listing 9 can be viewed in its entirety in Listing 45. It is too simple to require an explanation.

The frequency data

The bottom portion of Figure 3 shows the frequency data for each of the characters contained in the test message shown at the top of Figure 3.

```
Raw Data
Now is the time for all good men to come to the
aid of their country.

Number raw data bits: 552

Frequency Data
. 1
  15
N 1
y 1
w 1
u 1
t 7
s 1
r 3
o 9
n 2
m 3
l 2
i 4
h 3
g 1
f 2
e 6
d 2
c 2
a 2
```
Figure 3

And the results are...

As you can see, for this message, the space character occurred most frequently for a total of 15 occurrences. The character 'o' occurred next most frequently with nine occurrences followed by the 't' with seven occurrences. The period character and the characters 'N', 'y', 'w', 'u', 's', and 'q' tied for last place with only one occurrence each.

Three special classes

At this point, I need to explain the following three special classes in order for everything that follows to make sense:

- HuffTree
- HuffLeaf
- HuffNode

The HuffTree class

The **HuffTree** class is the abstract superclass of the other two classes in the above list. Objects of the **HuffNode** and **HuffLeaf** classes are used to construct the binary tree mentioned earlier.

The class definition for the **HuffTree** class begins in Listing 11. Note that this class implements the **Comparable** interface.

```
abstract class HuffTree implements Comparable{

  int frequency;

  public int getFrequency(){
    return frequency;
  }//end getFrequency

Listing 11
```

The Comparable interface

In case you are unfamiliar with the use and purposes of the **Comparable** interface, see Part 1 and Part2 my earlier lesson on that topic. Pay particular attention to the discussion of the requirement for objects that will be stored in a **TreeSet** object to implement the interface and to define the **compareTo** method.

The frequency property

Listing 11 declares the property variable named **frequency**.

Listing 11 also provides the property method named **getFrequency**, which when called on an object of the class, will return the value of the **frequency** property.

> *(In the event that you are unfamiliar with properties in Java, see the lessons on properties in the References section.)*

The compareTo method

The method named **compareTo** is declared by the **Comparable** interface and therefore must be defined by the **HuffTree** class. Here is some of what Sun has to say about the method:

"Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object."

Listing 12 shows the definition of the **compareTo** method in the **HuffTree** class. Along with the property named **frequency**, the **compareTo** method is inherited by both **HuffNode** and **HuffLeaf**. Listing 12 also signals the end of the **HuffTree** class definition.

```
  public int compareTo(Object obj){
    HuffTree theTree = (HuffTree)obj;
    if (frequency == theTree.frequency){
      //The objects are in a tie based on the
frequency
      // value.  Return a tiebreaker value
based on the
      // relative hashCode values of the two
objects.
      return (hashCode() - theTree.hashCode());
    }else{
      //Return negative or positive as this
frequency is
      // less than or greater than the
frequency value of
      // the object referred to by the
parameter.
      return frequency - theTree.frequency;
    }//end else
  }//end compareTo

}//end HuffTree class

Listing 12
```

Purpose and description of the compareTo method

The purpose of the **compareTo** method in this program is to make it possible for a collection object of the **TreeSet** class to compare two objects of the **HuffTree** class *(**HuffNode** or **HuffLeaf** objects)* to determine which is greater when sorting the objects into ascending order.

The **compareTo** method compares the object on which it is invoked to another object whose reference is received as an incoming parameter. The method guarantees that sorting processes that depend on this method, such as **TreeSet** objects, will sort the objects into a definitive order.

If the **frequency** property values of the two objects are different, the sort is based on the **frequency** values.

If the **frequency** values of the two objects are equal, the objects are sorted based on their relative **hashCode** values. Thus, if the same two objects with the same **frequency** value are compared two or more times during the execution of the program, those two objects will always be sorted into the same order. There is no chance of an ambiguous tie as to which object should be first except for the case where an object is compared to itself using two references to the same object.

The HuffNode class

The inner class named **HuffNode** is used to construct a node object in the Huffman tree. The class definition is shown in its entirety in Listing 13.

```
  class HuffNode extends HuffTree{

    private HuffTree left;
    private HuffTree right;

    //HuffNode constructor
    public HuffNode(
               int frequency,HuffTree 
left,HuffTree right){
      this.frequency = frequency;
      this.left = left;
      this.right = right;
    }//end HuffNode constructor

    public HuffTree getLeft(){
      return left;
    }//end getLeft

    public HuffTree getRight(){
      return right;
    }//end getRight

  }//end HuffNode class

Listing 13
```

HuffNode extends HuffTree

As you should expect from the previous discussion, the **HuffNode** class extends the **HuffTree** class. The class declares two instance variables named **left** and **right**. The purpose of these instance variables can best be explained by referring to the picture of the Huffman binary tree on the Binary Essence web site.

The **HuffNode** class is used to create internal node objects in the Huffman tree. *(See the node labeled 38 on the Binary Essence web site for example.)* Except for the root node, each such internal node is either a *left* or *right* child of a parent node. Also, each such node holds references to two child objects, which may be either **HuffNode** objects or **HuffLeaf** objects. The references to those two child objects are stored in the instance variables named **left** and **right** in Listing 13.

*(The two instance variables named **left** and **right** can hold references to **HuffNode** objects or **HuffLeaf** objects because they are declared as type*

HuffTree. ***HuffTree*** *is the abstract superclass of both* ***HuffNode*** *and* ***HuffLeaf****.)*

Beyond the explanation given above, the **HuffNode** code shown in Listing 13 is pretty straightforward and shouldn't require further explanation.

The HuffLeaf class

The inner class named **HuffLeaf** is used to construct a leaf object in the Huffman tree. The class definition for the **HuffLeaf** class is shown in its entirety in Listing 14.

```
  class HuffLeaf extends HuffTree{

    private int value;

    //HuffLeaf constructor
    public HuffLeaf(int value, int frequency){
      this.value = value;
      //Note that frequency is inherited from
HuffTree
      this.frequency = frequency;
    }//end HuffLeaf constructor

    public int getValue(){
      return value;
    }//end getValue

  }//End HuffLeaf class
```

Listing 14

Once again, as you should expect from the previous discussion, the **HuffLeaf** class extends the **HuffTree** class.

Two instance variables

The class declares a single instance variable named **value** and inherits another instance variable named **frequency** from the **HuffTree** class. The purpose of these instance variables can once again be explained by referring to the picture of the Huffman binary tree on the Binary Essence web site.

The **HuffLeaf** class is used to create leaf objects in the Huffman tree. *(See the leaf nodes labeled A, B, C, D, and E on the Binary Essence web site for example.)* Each such leaf node is either a *left* or *right* child of a parent node of type **HuffNode**.

(Leaf node B is the left child of the parent node labeled 22 and leaf node C is the right child of the parent node labeled 22.)

Leaf nodes do not hold references to child objects. Rather, they hold two values. One value is the representation of a character or symbol that appears in the message being compressed. The other value is the frequency, or the number of times that the character appears in the message being compressed. These two values are stored in the instance variables named **frequency** and **value** belonging to an object of the **HuffLeaf** class.

HuffNode objects also contain a value

I didn't mention it in the earlier discussion of the **HuffNode** objects because it didn't seem to be germane at that point. However, in addition to the two references to child objects that are held by **HuffNode** objects, **HuffNode** objects also hold a frequency value. In the case of a **HuffNode** object, the frequency value is the sum of the frequency values of its two child objects.

For example, in the picture of the Huffman binary tree on the Binary Essence web site, the internal node labeled 22 holds a frequency value of 22, which is the sum of the frequency values of the child nodes labeled B and C. The frequency value held by child node B is 12, and the frequency value held by child node C is 10. The sum of these two frequency values is 22, the value held by their parent node.

The character values held by those two child nodes are respectively B and C.

Now that we understand the **HuffTree**, **HuffNode**, and **HuffLeaf** classes, I can resume the discussion of the **encode** method of the **HuffmanEncoder** class.

Create a set of HuffLeaf objects

Returning now to the discussion of the **encode** method, Listing 15 invokes the **createLeaves** method to create a **HuffLeaf** object for every character identified in the frequency chart that was created in Listing 9.

```
    createLeaves();
```

Listing 15

The createLeaves method

The **createLeaves** method is shown in its entirety in Listing 16.

```
  void createLeaves(){
    Enumeration <Character>enumerator =

frequencyData.keys();
    while(enumerator.hasMoreElements()){
      Character nextKey =
enumerator.nextElement();
      theTree.add(new HuffLeaf(

nextKey,frequencyData.get(nextKey)));
```

```
    }//end while
  }//end createLeaves
```

Listing 16

As mentioned above, the **createLeaves** method creates a **HuffLeaf** object for every character identified in the frequency chart. The **HuffLeaf** objects are stored in a **TreeSet** object. Each **HuffLeaf** object encapsulates the character as well as the number of times that the character appears in the original message that is being compressed.

> *(In case you are unfamiliar with the Enumeration interface, see my earlier lesson entitled Vectors, Hashtables, and Enumerations.)*

Create the Huffman tree

Listing 17 invokes the **createHuffTree** method to assemble the collection of **HuffLeaf** objects stored in the **TreeSet** object into a Huffman tree *(a **HuffTree** object)*, also stored in the same **TreeSet** object.

```
    createHuffTree();
```

Listing 17

A Huffman tree is a special form of a binary tree consisting of properly linked **HuffNode** objects and **HuffLeaf** objects.

When the **createHuffTree** method in Listing 17 returns, the **HuffTree** object remains as the only object stored in the **TreeSet** object that previously contained all of the **HuffLeaf** objects. This is because all of the **HuffLeaf** objects have been combined with **HuffNode** objects to form the single **HuffTree** object.

The createHuffTree method

The **createHuffTree** method begins in Listing 18.

```
  void createHuffTree(){

    //Enable the following statements to see 
the original
    // contents of the TreeSet object. Do this 
only for
    // small trees because it generates lots of 
output.
/*
    System.out.println("\n\nDisplay Original 
TreeSet");
    Iterator <HuffTree> originalIter = 
theTree.iterator();
```

```
    while(originalIter.hasNext()){
      System.out.println(
                      "\nHuffNode, HuffLeaf, or
HuffTree");
      displayHuffTree(originalIter.next(),0);
    }//end while loop
    //End code to display the TreeSet
*/
```

Listing 18

When enabled, the code in Listing 18 displays the contents of the **TreeSet** object before the effort is begun to combine the **HuffLeaf** objects with **HuffNode** objects to create the **HuffTree** object.

> *(Note the caution in Listing 18 against enabling this display code for large trees, meaning messages that contain lots of different characters.)*

A short example

Figure 4 shows the output produced by the code in Listing 18 *(and the code from some earlier listings)* for the short message shown at the top of Figure 4.

```
Raw Data
AAAAABBBBCCCDDE

Number raw data bits: 120

Frequency Data
A 5
E 1
D 2
C 3
B 4


Display Original TreeSet

HuffNode, HuffLeaf, or HuffTree
Leaf:E
Back
HuffNode, HuffLeaf, or HuffTree
Leaf:D
Back
HuffNode, HuffLeaf, or HuffTree
Leaf:C
Back
HuffNode, HuffLeaf, or HuffTree
Leaf:B
Back
HuffNode, HuffLeaf, or HuffTree
Leaf:A
```

Back
Figure 4

Five HuffLeaf objects

The frequency chart for this message appears near the top of Figure 4. From this frequency chart *(and also from the text of the test message at the top)*, we can see that the **TreeSet** object should contain five **HuffLeaf** objects encapsulating the character values A, B, C, D, and E.

The bottom portion of Figure 4 shows the five **HuffLeaf** objects.

> *(I manually colored the individual elements with alternating colors of blue and red to make them visually distinguishable. I will have more to say about the display format later.)*

At this point, it important to note that the five **HuffLeaf** objects have been sorted into ascending order based on their frequency values as shown in the frequency chart near the top of Figure 4. I will be referring back to that fact later.

The displayHuffTree method

Listing 18 invokes the **displayHuffTree** method to produce the output shown in the bottom portion of Figure 4. I will be calling that method again later in a more significant way *(insofar as the output format is concerned)* so I will defer any further discussion until then.

Create the Huffman tree

The code that creates the Huffman tree begins in Listing 19. Before getting into the details of the code, here is a general description of the behavior of the code that creates the Huffman tree

Overall, the code assembles the collection of **HuffLeaf** objects into a **HuffTree** object. A **HuffTree** object is a special form of a binary tree consisting of properly linked **HuffNode** objects and **HuffLeaf** objects.

When the operation has been completed, the **HuffTree** object remains as the only object stored in the **TreeSet** object that previously contained all of the **HuffLeaf** objects. This is because, at that point in the execution of the code, all of the **HuffLeaf** objects have been removed from the **TreeSet** object and combined with **HuffNode** objects to form the Huffman tree *(as represented by the single **HuffTree** object)*.

Displays at runtime

The **createHuffTree** method contains two sections of code that can be enabled to display:

1. The contents of the original TreeSet object.

2. The contents of the TreeSet object for each iteration during which **HuffLeaf** objects are being combined with **HuffNode** objects to form the final **HuffTree** object.

You have already seen an example of the first display in Figure 4. You will see an example of the second display later in Figure 5.

These displays are very useful for understanding how the Huffman tree is actually constructed.

Steps in constructing the HuffTree object

The **HuffTree** object is constructed by performing the following operations:

1. Extracting pairs of **HuffLeaf** and/or **HuffNode** objects from the **TreeSet** object in ascending order based on their frequency values.
2. Using the pair of extracted objects to construct a new **HuffNode** object where the two extracted objects become children of the new **HuffNode** object, and where the frequency value stored in the new **HuffNode** object is the sum of the frequency values in the two child objects.
3. Removing the two original **HuffLeaf** and/or **HuffNode** objects from the **TreeSet** object and adding the new **HuffNode** object to the **TreeSet** object. The position of the new **HuffNode** object in the sorted **TreeSet** object is determined by its frequency value relative to the other **HuffNode** and/or **HuffLeaf** objects in the collection. The new **HuffNode** object will eventually become a child of another new **HuffNode** object unless it ends up as the root of the **HuffTree** object.
4. Continuing this process until the **TreeSet** object contains a single object of type **HuffTree**.

Iterate on the TreeSet object

Listing 19 shows the beginning of a **while** loop that iterates on the **TreeSet** object until the number of elements contained in the object is equal to 1. When the number of elements in the **TreeSet** object has been reduced to 1, all of the **HuffNode** and **HuffLeaf** elements in the collection will have been combined into a single element of type **HuffTree**.

```
    while(theTree.size() > 1){
      //Get, save, and remove the first two
elements.
      HuffTree left = theTree.first();
      theTree.remove(left);
      HuffTree right = theTree.first();
      theTree.remove(right);
```

Listing 19

As we saw in Figure 4, the collection of **HuffLeaf** objects in the **TreeSet** object is initially sorted into ascending order based on the frequency values encapsulated in the objects.

The code in Listing 19 removes and saves the first two objects in the collection, which are the objects containing the lowest frequency values.

> *(Initially, these two objects are both **HuffLeaf** objects, but later they may be either **HuffLeaf** objects or **HuffNode** objects or both.)*

Create and save a HuffNode object

Listing 20 combines the two saved objects into a new **HuffNode** object *(forming a three-object sub tree)* and adds the sub tree to the **TreeSet** object.

```
      HuffNode tempNode = new
HuffNode(left.getFrequency()
                        +
right.getFrequency(),left,right);
      theTree.add(tempNode);
```

Listing 20

Thus, each pass through the while loop removes the first two elements from the **TreeSet** collection, uses those two elements to create a sub tree, and adds the sub tree back to the collection. This process continues until the sub tree is the final tree, at which point the number of elements in the collection has been reduced to one and the **while** loop terminates.

Display the contents of the TreeSet object

Still in the **while** loop, Listing 21 contains code that can be enabled to display the contents of the **TreeSet** object at the end of each iteration.

```
      //Enable the following statements to see
the HuffTree
      // being created from HuffNode and
HuffLeaf objects.
      // Do this only for small trees because
it will
      // generate a lot of output.
/*
      System.out.println("\n\nDisplay Working
TreeSet");
      Iterator <HuffTree> workingIter =
theTree.iterator();
      while(workingIter.hasNext()){
        System.out.println(
                      "\nHuffNode, HuffLeaf, or
HuffTree");
        displayHuffTree(workingIter.next(),0);
      }//end while loop
      //End code to display the TreeSet
*/
    }//end while
```

```
  }//end createHuffTree
```

Listing 21

Listing 21 also signals the end of the **while** loop and the end of the **createHuffTree** method.

The program output

The bottom portion of Figure 5 shows the output produced during the first iteration of the **while** loop by enabling the display code in Listing 21.

```
Raw Data
AAAAABBBBCCCDDE

Number raw data bits: 120


Display Working TreeSet

HuffNode, HuffLeaf, or HuffTree
Leaf:C
Back
HuffNode, HuffLeaf, or HuffTree
Left to 1 Leaf:E
Back Right to 1 Leaf:D
Back Back
HuffNode, HuffLeaf, or HuffTree
Leaf:B
Back
HuffNode, HuffLeaf, or HuffTree
Leaf:A
Back
```

Figure 5

At the end of the first iteration...

At the end of the first iteration of the **while** loop, the collection contains four different elements. *(Recall from Figure 4 that the collection originally contained five elements.)*

I colored the four elements in Figure 5 with alternating colors of blue and red to make them visually distinguishable.

In Figure 4, the first two elements in the collection were the **HuffLeaf** objects encapsulating the characters E and D. In Figure 5, those two elements have been removed and combined with a **HuffNode** object to form a sub tree, which is shown as the first red element in Figure 5.

The sub tree

Figure 6 is an attempt to show a picture of the sub tree, which is the second *(red)* element in the collection shown in Figure 5. *(Hopefully this picture will survive the various publishing programs to which this lesson will be subjected)*

```
           Root
Level 0     O
           / \
Level 1   E   D

Figure 6
```

The sub tree consists of three parts. The root of the sub tree is a **HuffNode** object *(shown by the O in Figure 6)*. The left child of the node is a **HuffLeaf** object *(shown by the E in Figure 6)*. The right child of the node is also a **HuffLeaf** object *(shown by the D in Figure 6)*.

The format of the printed description

Conceptually, we can think of the root node as existing at Level 0 in the tree and the two child nodes existing at Level 1 in the tree. That brings us to the format used to describe the elements in Figure 5.

Consider the first line in the description of the sub tree in Figure 5, which reads HuffNode, HuffLeaf, or HuffTree. This line simply serves as a separator between the elements.

Beginning with the second line in the description of the sub tree, the text is a verbal description of how to start at the root and to *traverse* the tree. This text describes the sub tree by the traversal path. It says:

- Starting at the root *(Level 0 implied)*, go down and to the left to Level 1. There you will find a Leaf that encapsulates the character E.
- Then go back up one level. This will return you to Level 0. Go down and to the right to Level 1. There you will find a Leaf that encapsulates the character D.
- Then go back up two levels, which will pop you out of the top of the tree.

Adding the sub tree to the TreeSet collection

After the E and D leaves were removed from the collection and combined with a node to form a sub tree, that sub tree was added back to the collection as shown in Listing 20.

The node that forms the root of the sub tree was assigned the frequency value 3, which is the sum of the frequency values for D and E as shown in the frequency chart in Figure 4. As a result, the frequency value for the root of the sub tree has the same frequency value as the leaf for the character C.

Tied for position in the collection

This causes the new sub tree to be tied for position in the sorted collection with the leaf for the character C. It also causes the frequency value for the sub tree to be less than the frequency value for the B leaf which has a frequency value of 4.

Given these values, the sub tree could have ended up as either the first or the second element in the collection shown in Figure 5, because it tied for the first position with the C leaf. However, it had to be closer to the beginning than the B leaf. As it turns out, the tie-breaker methodology used in the **compareTo** method in Listing 12 placed it after the C leaf in Figure 5.

Output following the second iteration

Figure 7 shows the state of the **TreeSet** object following the completion of the second iteration of the **while** loop that began in Listing 19.

```
HuffNode, HuffLeaf, or HuffTree
Leaf:B
Back
HuffNode, HuffLeaf, or HuffTree
Leaf:A
Back
HuffNode, HuffLeaf, or HuffTree
Left to 1 Leaf:C
Back Right to 1 Left to 2 Leaf:E
Back Right to 2 Leaf:D
Back Back Back
```
Figure 7

At this point, the number of elements in the collection has been reduced from the four elements shown in Figure 5 to the three shown in Figure 7. The reduction was accomplished by removing the C leaf in the first element and the sub tree from the second element in Figure 5 and using them to construct a larger sub tree. That larger sub tree is shown as the last element in Figure 7.

> *(See if you can create a sketch of the new sub tree on the basis of the traversal description in Figure 7.)*

The new sub tree

The pictorial representation of this larger sub tree, shown as the last element in the collection in Figure 7, is shown in Figure 8.

```
                Root
Level 0           O
                 / \
Level 1         C   O
                   / \
Level 2           E   D
```
Figure 8

The Huffman tree

Skipping ahead to the last iteration, Figure 9 shows the state of the **TreeSet** object following the final iteration of the **while** loop. At this point, the collection contains only one element, and it is the Huffman tree produced by combining all of the leaf elements shown in Figure 4 with nodes to produce the tree.

```
HuffNode, HuffLeaf, or HuffTree
Left to 1 Left to 2 Leaf:C
Back Right to 2 Left to 3 Leaf:E
Back Right to 3 Leaf:D
Back Back Back Right to 1 Left to 2 Leaf:B
Back Right to 2 Leaf:A
Back Back Back
Figure 9
```

Pictorial representation of the Huffman tree

The pictorial representation of the Huffman tree described in Figure 9 is shown in Figure 10

```
                     Root
Level 0               X
                     / \
                   0/   \1
                   /     \
                  /       \
Level 1          X         X
               0/ \1     0/ \1
Level 2        C   X     B   A
                 0/ \1
Level 3          E   D
Figure 10
```

In Figure 10, I used the character X to indicate an internal node in place of the character O that was used to depict the internal nodes in the earlier pictures of the trees. This is because I wanted to decorate the tree in Figure 10 with 0 and 1 characters and I wanted to avoid confusion between the character for an internal node and the zero character.

> *(You will see the reason for the decoration of the tree using 0 and 1 characters later.)*

Hopefully you now understand how the **TreeSet** object containing a collection of **HuffLeaf** objects is processed to produce a single **HuffTree** object, which is the required Huffman tree.

The displayHuffTree method

The output shown in Figure 4, Figure 5, Figure 7, and Figure 9 was produced by the invocation of the **displayHuffTree** method in Listing 18 and Listing 21. You can view the **displayHuffTree** method in its entirety in Listing 45. This is a recursive method, which is very similar to another recursive method named **createBitCodes** that I will discuss later, so I will leave it up to you to understand the inner workings of the **displayHuffTree** method on your own at this point.

Recap

At this point, I have developed the frequency information for the characters in the short test message at the top of Figure 4. I displayed the frequency information in Figure 4.

I used the frequency information to create a **HuffLeaf** object for every character in the message where the **HuffLeaf** object encapsulates the character and the frequency information for that character. I displayed the **HuffLeaf** objects in Figure 4 also.

Then I combined those **HuffLeaf** objects with **HuffNode** objects to create a **HuffTree** object. I displayed intermediate pictorial versions of the **HuffTree** object in Figure 6 and Figure 8, and displayed the final version in Figure 10. Figure 10 shows the Huffman binary tree for the short test message at the top of Figure 4.

If we compare the Huffman tree in Figure 10 with the frequency information in Figure 4, we see that the leaf objects representing the characters with the highest frequency of usage in the message *(C, B, and A)* occur closest to the root of the tree *(Level 2)* and those with the lowest frequency of usage *(E and D)* occur further from the root of the tree *(Level 3).*

Traversing the Huffman tree

The next task is to use the Huffman tree to create a unique binary bit code for every character used in the original test message.

> *(That is the reason that I decorated the tree in Figure 10 with 0 and 1 characters.)*

The way that I will create the binary codes is to traverse the path from the root of the binary tree to each leaf of the tree, keeping track of the decorations that I encounter along the way. Each time I encounter a 0, I will append a 0 to the bit code for that character. Each time I encounter a 1, I will append 1 to the bit code for that character.

Manually generated bit codes

Figure 11 shows the result of manually performing this operation for each of the leaves in Figure 10 in left-to-right order for the leaves.

```
C 00
E 010
D 011
B 10
A 11
```
Figure 11

Now that you know how the task is accomplished, we will take a look at the code that automates this process.

Automate the task of creating bit codes

Back in the **encode** method, Listing 22 invokes the method named **createBitCodes** to automate the task of traversing the Huffman tree for the purpose of creating a unique bit code for every character described by a leaf of the tree. A reference to the single **HuffTree** object stored in the **TreeSet** object is passed to the **createBitCodes** method.

```
    createBitCodes(theTree.first());
```

Listing 22

A recursive method

The **createBitCodes** method traverses the Huffman tree in a recursive manner to create a bit code for each character in the message. The bit codes are different lengths with the shorter codes corresponding to the characters with a high frequency value and the longer codes corresponding to the characters with the lower frequency values.

Note that the method call extracts the reference to the Huffman tree from the **TreeSet** object and passes that reference to the method. This is necessary because the **createBitCodes** method is recursive and can't conveniently work directly with the **TreeSet** object.

The **createBitCodes** method populates the data structure that is used to encode the message and required later to decode the encoded message.

Recursion in Java

In order to understand the method named **createBitCodes**, you must understand recursion. Unfortunately, none of the several hundred Java programming tutorials that I have published in recent years, *(several of which use recursion)*, are dedicated to an understanding of recursion in Java.

> *(The writing of a tutorial on recursion has been on my list of things to do for several years now.)*

Fortunately, if you Google the keywords **Java** and **recursion**, you will find a variety of references, some good, and some not so good. One that seems to be very good can be found on the web site of the City University, London.

The createBitCodes method

The **createBitCodes** method begins in Listing 23. This method traverses the Huffman tree in a recursive manner to create a bit code for each character in the message. The bit codes are different lengths with the shorter bit codes corresponding to the characters with a high usage frequency value and the longer bit codes corresponding to the characters with the lower frequency values.

This method receives a reference to the Huffman tree that was earlier contained as the only object in the **TreeSet** object.

A Huffman encoding table

The **createBitCodes** method creates a Huffman encoding table as a **Hashtable** object that relates the variable-length bit codes to the characters in the original message. The bit codes are constructed as objects of type **StringBuffer** consisting of sequences of the characters 1 and 0 and converted to type **String** for storage in the **Hashtable**.

The traversal path

Each bit code describes the traversal path from the root of the Huffman tree to a leaf on that tree. Each time the path turns to the left, a 0 character is appended onto the **StringBuffer** object and becomes a part of the resulting bit code. Each time the path turns to the right, a 1 character is appended onto the **StringBuffer** object.

When a leaf is reached at the end of the traversal path, the character stored in that leaf is retrieved and put into the **Hashtable** object as a key. A **String** representation of the **StringBuffer** object is used as the value for that key in the **Hashtable**.

The final result

At completion, the **Hashtable** object contains a series of keys consisting of the original characters in the message and a series of corresponding values as **String** objects *(consisting only of 1 and 0 characters)* representing the bit codes that will eventually be used to encode the original message.

The Hashtable object that is populated by this method is the data structure that is used to encode the message and is required later to decode the encoded message.

Test for the type of node

If you understand recursion in Java, you should find the **createBitCodes** method to be straightforward. If not, you will probably find it to be very difficult to understand.

Listing 23 begins by testing the node to determine if it is an internal node or a leaf node.

```
  void createBitCodes(HuffTree tree){
    if(tree instanceof HuffNode){
      // This is a node, not a leaf.  Process
it as a node.
```

```
      //Cast to type HuffNode.
      HuffNode node = (HuffNode)tree;
      // Get and save the left and right
branches
      HuffTree left = node.getLeft();
      HuffTree right = node.getRight();

      //Append a 0 onto the StringBuffer
object.  Then make
      // a recursive call to this method
passing a
      // reference to the left child as a
parameter.  This
      // recursive call will work its way all
the way down
      // to a leaf before returning.  Then it
will be time
      // to process the right path.
      code.append("0");
      createBitCodes(left);

Listing 23
```

If you understand recursion, the comments in Listing 23 should cause the code in Listing 23 to be self explanatory.

Process the right leg

Listing 24 picks up at the point where the recursive process has finally returned from its traversal down the left leg of the binary tree. The code in Listing 24 processes the right leg of the binary tree.

```

      //Return to here from recursive call on
left child.

      //Delete the 0 from the end of the
StringBuffer
      // object to restore the contents of that
object to
      // the same state that it had before
appending the 0
      // and making the recursive call on the
left branch.
      //Now we will make a right turn.  Append
a 1 to the
      // StringBuffer object and make a
recursive call to
      // this method passing a reference to the
right child
      // as a parameter.  Once again, this
recursive call
      // will work its way all the  way down to
```

```
a leaf
      // before returning.
      code.deleteCharAt(code.length() -
1);//Delete the 0.
      code.append("1");
      createBitCodes(right);

      //Return to here from recursive call on
right child.

      //Delete the character most recently
appended to the
      // StringBuffer object and return from
this recursive
      // call to the method.  The character is
deleted
      // because control is being transferred
back one
      // level in the recursive process and the
      // StringBuffer object must be restored
to the same
      // state that it had when this recursive
call was
      // made.
      code.deleteCharAt(code.length() - 1);
```

Listing 24

Process a leaf node

Listing 23 and Listing 24 were both concerned with the processing of internal nodes in the binary tree. Listing 25 picks up at the point where it has been determined that the node is a leaf node instead of an internal node. Listing 25 processes the leaf node.

```
    }else{
      //This is a leaf.  Process it as such.
      //Cast the object to type HuffLeaf.
      HuffLeaf leaf = (HuffLeaf)tree;

      //Put an entry into the Hashtable.  The
Hashtable
      // key consists of the character value
stored in the
      // leaf. The value in the Hashtable
consists of the
      // contents of the StringBuffer object
representing
      // the path from the root of the tree to
the leaf.
      // This is the bitcode and is stored in
the Hashtable
      // as a String consisting of only 1 and 0
characters.
      huffEncodeTable.put((char)(
leaf.getValue()),code.toString());
    }//end else

  }//end createBitCodes
```

Listing 25

Understanding the inner workings

Hopefully, you have been able to understand the inner workings of the method named **createBitCodes**. If not, just take my word for it that the behavior of the method is to automate the process that I described earlier in the section entitled Traversing the Huffman tree. Even if you don't understand exactly how the method named **createBitCodes** does its job, you should understand the outcome of invoking that method.

Display the bit codes

The code in Listing 26 can be enabled to display the bit codes that were created above and which are used to populate the Huffman encoding table.

```
    //For purposes of illustration only, enable
the
    // following two statements to display a
table showing
    // the relationship between the characters
in the
    // original message and the bitcodes that
will replace
    // those characters to produce the Huffman-
encoded
    // message.
/*
    System.out.println();
    displayBitCodes();
*/
```

Listing 26

The displayBitCodes method

You can view the displayBitCodes method in its entirety in Listing 45. This method is very straightforward and should not require further explanation.

Sample bit code display

The bottom portion of Figure 12 shows the bit codes for the characters in the short message at the top of Figure 12. Compare these bit codes with the bit codes that I produced manually in Figure 11. Except for the display order, you should find that they match exactly.

```
Raw Data
AAAAABBBBCCCDDE

Number raw data bits: 120


Message Characters versus Huffman BitCodes
A 11
E 010
D 011
C 00
B 10
```
Figure 12

(If you investigate deeply enough, you will find that the method named ***createBitCodes*** *develops the bit codes and populates the* ***Hashtable*** *in the same order as shown in Figure 11. However, when an* ***Enumeration*** *object is used to retrieve and display the codes in the* ***displayBitCodes*** *method, the enumerator retrieves them in a different order.)*

Bit codes for a longer message

The bottom portion of Figure 13 shows the variable-length bit codes for the somewhat longer test message shown at the top of Figure 13.

```
Raw Data
Now is the time for all good men to come to the
aid of their country.

Number raw data bits: 552


Message Characters versus Huffman BitCodes
. 100111
  00
N 111000
y 011101
w 110010
u 100110
t 010
s 011100
r 11101
o 101
n 10000
m 11110
l 10001
i 0110
h 11111
g 110011
f 01111
e 1101
d 111001
c 11000
```

```
a 10010
Figure 13
```

Whereas the bit codes in Figure 12 were all either two bits or three bits in length, the bit codes in Figure 13 range from two bits *(00)* at the shortest to six bits *(111000 for example)* at the longest.

Bit codes for a much longer message

The bottom portion of Figure 14 shows the bit codes for the characters in the even longer message shown at the top of Figure 14.

```
Raw Data
BAGHDAD, Iraq Violence increased across Iraq
aft
er a lull following the Dec. 15 parliamentary
el
ections, with at least two dozen people
includin
g a U.S. soldier killed Monday in shootings and
bombings mostly targeting the Shiite-dominated
s
ecurity services. The Defense Ministry director
of operations, Brig. Gen. Abdul Aziz Mohammed-
Ja
ssim, blamed increased violence in the past two
days on insurgents trying to deepen the
politica
l turmoil following the elections. The violence
came as three Iraqi opposition groups
threatened
another wave of protests and civil disobedience
if allegations of fraud are not properly invest
igated.

Number raw data bits: 5048

Message Characters versus Huffman BitCodes
V 000100000     U 101110000     T 01010010      S
10001000
M 10001001      J 000110101     I 10001111      H
000110110
G 01010011      D 0101000       B 101110001     A
0001001
5 000100001     1 000110100     . 000111        -
00010001
, 0001100         110           z 10001110      y
000101
w 1000110       v 1011101       u 010101        t
1001
s 0100          r 0000          q 10111001      p
111100
o 1010          n 0111          m 100001        l
11111
```

```
k 000110111     i 1110          h 111101        g
101111
f 100000        e 001           d 10110         c
01011
b 1000101       a 0110
```

Figure 14

The bit codes for the message in Figure 14 range from three bits for the 'e' and the space character to nine bits for several characters including 'U', 'V', and 'k'.

> *(Note that I manually rearranged the bit codes in Figure 14 so that they would fit on a single screen for easier viewing.)*

Encode message into a String of 1 and 0 characters

Back in the **encode** method, Listing 27 invokes the **encodeToString** method to encode the message into a **String** representation of the bits that will make up the final encoded message.

```
    encodeToString();
```

Listing 27

The **encodeToString** method also provides the optional capability to display the **String** showing the bit values that will appear in the final Huffman-encoded message. This can be useful for comparing back against the bits in the original message for purposes of evaluating the amount of compression provided by encoding the message.

The encodeToString method

The **encodeToString** method is shown in its entirety in Listing 28.

```
  void encodeToString(){
    StringBuffer tempEncoding = new 
StringBuffer();
    for(int cnt = 0;cnt < 
rawData.length();cnt++){
      //Do a table lookup to get the substring 
that
      // represents the bitcode for each 
message character.
      // Append those substrings to the string 
that
      // represents the Huffman-encoded 
message.
      tempEncoding.append(huffEncodeTable.get(
                                  
rawData.charAt(cnt)));
    }//end for loop

    //Convert the StringBuffer object to a 
String object.
    stringEncodedData =
tempEncoding.toString();

    //For illustration purposes, enable the
following two
    // statements to display the String showing
the bit
    // values that will appear in the Huffman-
encoded
    // message.  Display 48 bits to the line
except for
    // the last line, which may be shorter, and
which may
    // not be a multiple of 8 bits.
/*
    System.out.println("\nString Encoded
Data");
    display48(stringEncodedData);
*/
  }//end encodeToString
```

Listing 28

The **encodeToString** method encodes the message into a **String** representation of the bits that will make up the final encoded message. The **String** consists of only 1 and 0 characters where each character represents the state of one of the bits in the Huffman-encoded message. Also for illustration purposes, this method optionally displays the **String** showing the bit values that will appear in the Huffman-encoded message.

The **encodeToString** method is completely straightforward and shouldn't require further explanation.

Build an encoding bit map

Back in the **encode** method, Listing 29 invokes the **buildEncodingBitMap** method to populate a lookup table that relates eight bits represented as a **String** to every possible combination of eight actual bits.

```
    buildEncodingBitMap();
```

Listing 29

You can view the **buildEncodingBitMap** method in Listing 45. It is so straightforward as to not warrant an explanation here.

Encode the String representation into actual bits

Listing 30 invokes the method named **encodeStringToBits** to encode the **String** representation of the bits that make up the encoded message to the actual bits that make up the encoded message.

```
    encodeStringToBits();
```

Listing 30

The encodeStringToBits method

The **encodeStringToBits** method is shown in its entirety in Listing 31.

```

  void encodeStringToBits(){
    //Extend the length of the 
stringEncodedData to cause
    // it to be a multiple of 8.
    int remainder = 
stringEncodedData.length()%8;
    for(int cnt = 0;cnt < (8 - 
remainder);cnt++){
      stringEncodedData += "0";
    }//end for loop

    //For illustration purposes only, enable 
the following
    // two statements to display the extended
    // stringEncodedData in the same format as 
the
    // original stringEncodedData.
/*
    System.out.println("\nExtended String 
Encoded Data");
    display48(stringEncodedData);
*/
    //Extract the String representations of the 
required
    // eight bits.  Generate eight actual 
matching bits by
    // looking the bit combination up in a 
table.
    for(int cnt = 0;cnt < 
stringEncodedData.length();
                                                
cnt += 8){
      String strBits  = 
stringEncodedData.substring(
                                                
cnt,cnt+8);
      byte realBits = 
encodingBitMap.get(strBits);
      binaryEncodedData.add(realBits);
    }//end for loop
  }//end encodeStringToBits
```

Listing 31

The purpose of the **encodeStringToBits** method is to create actual bit data that matches the 1 and 0 characters in the **stringEncodedData** that represents bits with the 1 and 0 characters.

Pad characters at the end

Note that this method doesn't handle the end of the data very gracefully for those cases where the number of required bits is not a multiple of 8. The method simply adds enough "0" characters to the end to cause the length to be a multiple of 8. This may result in extraneous characters at the end of the decoded message later. However, it isn't difficult to remove the extraneous characters at decode time as long as the length of the original message is known.

For illustration purposes, this method may optionally display the extended version of the **stringEncodedData** for comparison with the non-extended version.

Output format

The binary Huffman-encoded data produced by this method is stored in a data structure of type **ArrayList <Byte>**.

Once you understand the methodology, the code in Listing 31 is straightforward and should not require further explanation.

Return from the encode method

Back in the **encode** method, the code in Listing 32 returns the binary Huffman-encoded data, terminates the **encode** method, and returns control to the **main** method where it was called in Listing 5.

```
    return binaryEncodedData;
  }//end encode method
```

Listing 32

Listing 32 also signals the end of the class definition for the class named **HuffmanEncoder**.

Meanwhile, back in the main method...

Listing 33 displays the number of bits in the encoded message and then computes and displays the compression factor.

```
    System.out.println("Number binary encoded data
bits: "
```

```
                           +
binaryEncodedData.size() * 8);
    System.out.println("Compression factor: "
      +
(double)rawData.length()/binaryEncodedData.size());
```

Listing 33

An example of compression factor

Listing 33 produces the compression-factor output shown in the bottom portion of Figure 15 for the test message shown in the top of Figure 15.

```
Raw Data
BAGHDAD, Iraq Violence increased across Iraq
aft
er a lull following the Dec. 15 parliamentary
el
ections, with at least two dozen people
includin
g a U.S. soldier killed Monday in shootings and
bombings mostly targeting the Shiite-dominated
s
ecurity services. The Defense Ministry director
of operations, Brig. Gen. Abdul Aziz Mohammed-
Ja
ssim, blamed increased violence in the past two
days on insurgents trying to deepen the
politica
l turmoil following the elections. The violence
came as three Iraqi opposition groups
threatened
another wave of protests and civil disobedience
if allegations of fraud are not properly invest
igated.

Number raw data bits: 5048
Number binary encoded data bits: 2816
Compression factor: 1.7926136363636365
```

Figure 15

Another example of compression factor

Similarly, the bottom portion of Figure 16 shows the compression factor achieved for the test message shown in the top of Figure 16.

```
Raw Data
Now is the time for all good men to come to the
aid of their country.

Number raw data bits: 552
Number binary encoded data bits: 272
```

```
Compression factor: 2.0294117647058822
Figure 16
```

Comparing the results in Figure 15 with the results in Figure 16, you can see that the compression results are highly dependent on the content of the message, even for the case where the message is composed entirely of English text.

Display the encoded message in hexadecimal format

The test message has now been Huffman encoded. Continuing with the **main** method, Listing 34 invokes the method named **hexDisplay48** to display the **binaryEncodedData** in hexadecimal format, 48 characters per line.

```
    System.out.println(
            "\nBinary Encoded Data in
Hexadecimal Format");
    hexDisplay48(binaryEncodedData);
    System.out.println();

Listing 34
```

The method named hexDisplay48

The method named **hexDisplay48** can be viewed in its entirety in Listing 45. The code in this method is completely straightforward and shouldn't require further explanation.

Sample hexadecimal display of Huffman-encoded data

The bottom portion of Figure 17 shows a hexadecimal display of the binary Huffman-encoded version of the message shown at the top of Figure 17

```
Raw Data
Now is the time for all good men to come to the
aid of their country.

Number raw data bits: 552
Number binary encoded data bits: 272
Compression factor: 2.0294117647058822

Binary Encoded Data in Hexadecimal Format
e2e43382fe89bda3ef49462676f27b602a62fb4545fd24dc
95e2feb74c59a0baece0
Figure 17
```

Comparing hexadecimal and raw data displays

When comparing the hexadecimal display with the display of the raw data, remember that each raw-data character represents eight bits while each hexadecimal character represents only four bits. Therefore, even though the hexadecimal display appears to be about as long as the raw-data display, the number of bits required to represent the Huffman-encoded version of the message in Figure 17 is only 49.2 percent of the number of bits required to represent the raw unencoded version. This results in a compression factor of 2.029 for this particular message.

Having encoded the message, the time has come to decode it to produce an exact *(lossless)* replica of the original message.

Decode the Message

Still in the **main** method, I will continue the demonstration by decoding the encoded message. I will accomplish this by instantiating an object of the **HuffmanDecoder** class and invoking the **decode** method on that object as shown in Listing 35.

```
    HuffmanDecoder decoder = new
HuffmanDecoder();


    String decodedData =
decoder.decode(binaryEncodedData,

huffEncodeTable,

rawDataLen);
```

Listing 35

(As you will soon see, it is somewhat easier to decode a Huffman-encoded message than it is to encode it.)

Information required to decode the message

Listing 35 passes the encoded message to the decode method of the **HuffmanDecoder** object along with a reference to the data structure containing the encoding particulars as well as the length of the original message.

(The length of the original message is used to eliminate extraneous characters on the end of the decoded message.)

The HuffmanDecoder class

Putting the **main** method aside for awhile, the beginning of the class definition of the **HuffmanDecoder** class begins in Listing 36.

```
class HuffmanDecoder{
  Hashtable <String,Character>huffDecodeTable =
                              new
Hashtable<String,Character>();
```

```
  String stringDecodedData;
  String decodedData = "";
  Hashtable <Byte,String>decodingBitMap =
                              new
Hashtable<Byte,String>();
  ArrayList <Byte>binaryEncodedData;

  //The following structure contains
particulars as to how
  // the original message was encoded, and must
be received
  // as an incoming parameter to the decode
method along
  // with the encoded message and the length of
the
  // original message.
  Hashtable <Character,String>huffEncodeTable;
  //Used to eliminate the extraneous characters
on the end.
  int rawDataLen;
```

Listing 36

An object of the **HuffmanDecoder** class can be used to decode a Huffman-encoded message given the encoded message, a data structure containing particulars as to how the original message was encoded, and the length of the original message.

The code in Listing 36 gets things started by declaring several instance variables and initializing some of them. I will discuss the instance variables in conjunction with the code that uses them.

The decode method

Listing 37 shows the beginning of the **decode** method of the **HuffmanDecoder** class.

```
  String decode(ArrayList
<Byte>binaryEncodedData,
               Hashtable
<Character,String>huffEncodeTable,
               int rawDataLen){
    //Save the incoming parameters.
    this.binaryEncodedData = binaryEncodedData;
    this.huffEncodeTable = huffEncodeTable;
    this.rawDataLen = rawDataLen;
```

Listing 37

The **decode** method receives a Huffman-encoded message along with a data structure containing particulars as to how the original message was encoded and the length of the original message. It decodes the original message and returns the decoded version as a **String** object.

Listing 37 saves the incoming parameters in instance variables that were declared in Listing 36.

Create a decoding bit map

Listing 38 invokes the **buildDecodingBitMap** method, which is essentially the reverse of the encoding bit map that was used to encode the original message.

```
    buildDecodingBitMap();
```

Listing 38

The buildDecodingBitMap method

The **buildDecodingBitMap** method can be viewed in its entirety in Listing 45. This method populates a lookup table that relates eight bits represented as a **String** to eight actual bits for all possible combinations of eight bits.

This is essentially a reverse lookup table relative to the **encodingBitMap** table that is used to encode the message. The only difference between the two is a reversal of the key and the value in the Hashtable that contains the table.

Decode from binary to String

Listing 39 invokes the **decodeToBitsAsString** method to decode the encoded message from a binary representation to a **String** of 1 and 0 characters that represent the actual bits in the encoded message.

```
    decodeToBitsAsString();
```

Listing 39

The decodeToBitAsString method

This method, which can be viewed in its entirety in Listing 45 is very straightforward. Therefore no further discussion is warranted.

Build a decoding table

Listing 40 invokes the **buildHuffDecodingTable** method to create a Huffman decoding lookup table by swapping the keys and values from the Huffman encoding table received as an incoming parameter by the decode method.

```
    buildHuffDecodingTable();
```

Listing 40

The buildHuffDecodingTable method

You guessed it! Once again, this method, which can be viewed in Listing 45, is too simple to deserve further discussion.

Decode the message

Listing 41 invokes the **decodeStringBitsToCharacters** method to decode the String containing only 1 and 0 characters that represent the bits in the encoded message. This produces a replica of the original message that was subjected to Huffman encoding.

```
    decodeStringBitsToCharacters();
```

Listing 41

The decodeStringBitsToCharacters method

The **decodeStringBitsToCharacters** method is shown in its entirety in Listing 42.

```
  void decodeStringBitsToCharacters(){
    StringBuffer output = new StringBuffer();
    StringBuffer workBuf = new StringBuffer();

    for(int cnt = 0;cnt <
stringDecodedData.length();

cnt++){
      workBuf.append(stringDecodedData.charAt(cnt));

if(huffDecodeTable.containsKey(workBuf.toString())){
        output.append(

huffDecodeTable.get(workBuf.toString()));
        workBuf = new StringBuffer();
      }//end if
    }//end for loop

    decodedData = output.toString();
  }//End decodeStringBitsToCharacters();
```

Listing 42

Two empty StringBuffer objects

The **decodeStringBitsToCharacters** method begins with one empty **StringBuffer** object referred to by the variable named **workBuf** and another empty **StringBuffer** object referred to by the variable named **output**.

The **StringBuffer** object referred to by **output** is used to construct the decoded message. The **StringBuffer** object referred to by **workBuf** is used as a temporary holding area for substring data.

Build a working string one character at a time

The method reads the **String** containing only 1 and 0 characters that represent the bits in the encoded message *(stringDecodedData)*. The characters are read from this string one character at a time. As each new character is read, it is appended to the **StringBuffer** object referred to by **workBuf**.

Compare the working string to the decoding keys

As each new character is appended to the **StringBuffer** object, a test is performed to determine if the current contents of the **StringBuffer** object match one of the keys in a **Hashtable** table that relates strings representing Huffman bit codes to characters in the original message.

When a match is found...

When a match is found, the *value* associated with that *key* is retrieved from the **Hashtable** and appended onto the **StringBuffer** object referred to by **output**. Thus, the output text is built up one character at a time.

Clear the working string

Having processed the matching key, a new empty **StringBuffer** object is instantiated and is referred to by **workBuf**. The process of reading, appending, and testing for a match is repeated until all of the characters in the string that represents the bits in the encoded message have been processed.

The original message has been reconstructed, with extra characters

At that point, the **StringBuffer** object referred to by **output** contains the entire decoded message. *(It may contain extraneous characters at the end.)* It is converted to type **String** and referred to by **decodedData**. Then the **decodeStringBitsToCharacters** method returns control to the **decode** method with the task of decoding the encoded message having been completed.

Back in the decode method...

As shown in Listing 43, the **decode** method returns control to the **main** method returning a reference to a **String** containing a replica of the original message in the process.

```
    return decodedData.substring(0,rawDataLen);
  }//end decode method
```

Listing 43

Note that the code in Listing 43 uses the known length of the original message to trim extraneous characters from the end of the decoded message.

Listing 43 signals the end of the **decode** method and the end of the **HuffmanDecoder** class.

Back in the main method...

Listing 44 invokes the **display48** method to display the decoded results, 48 characters to the line as shown near the bottom of Figure 1.

```
    System.out.println("\nDecoded Data");
    display48(decodedData);

  }//end main
```

Listing 44

And that's a wrap! I hope that you have enjoyed this lesson on Huffman encoding.

Run the Program

I encourage you to copy the code from Listing 45 into your text editor, compile it, and execute it. Experiment with it, making changes, and observing the results of your changes.

Create a variety of test messages of your own and determine the compression factor for different types of messages. For example apply the Huffman compression algorithm to raw HTML messages to see if they compress more than straight text messages. Create a simulation of an encrypted message using byte values from a random number generator as the message content and see how well that message compresses.

Summary

In this lesson, I have taught you about the inner workings of the Huffman lossless compression algorithm. I also showed you the results obtained by applying the algorithm to several different test messages.

What's Next?

Future lessons in this series will explain the inner workings behind several other data and image compression schemes, including the following:

- Run-length data encoding
- GIF image compression
- JPEG image compression

References

2440 Understanding the Lempel-Ziv Data Compression Algorithm in Java
076 Vectors, Hashtables, and Enumerations
508 JavaBeans, Properties of Beans, Simple and Indexed
510 JavaBeans, Properties of Beans, Bound Properties
512 JavaBeans, Properties of Beans, Constrained Properties
1350 Data Structures in Java: Part 1, Getting Started
1352 Data Structures in Java: Part 2, What Is a Collection?
1354 Data Structures in Java: Part 3, Purpose of Framework Interfaces
1356 Data Structures in Java: Part 4, Purpose of Implementations and Algorithms
1358 Data Structures in Java: Part 5, The Core Collection Interfaces
1360 Data Structures in Java: Part 6, Data Structures in Java: Part 6, Duplicate Elements, Ordered Collections, Sorted Collections, and Interface Specialization
1362 Data Structures in Java: Part 7, The Comparable Interface, Part 1
1364 Data Structures in Java: Part 8, The Comparable Interface, Part 2
1366 Data Structures in Java: Part 9, The Comparator Interface, Part 1
1368 Data Structures in Java: Part 10, The Comparator Interface, Part 2
1370 Data Structures in Java: Part 11, The Comparator Interface, Part 3
1372 Data Structures in Java: Part 12, The Comparator Interface, Part 4
1374 Data Structures in Java: Part 13, The Comparator Interface, Part 5
1376 Data Structures in Java: Part 14, The Comparator Interface, Part 6
1378 Data Structures in Java: Part 15, The toArray Method, Part 1
1380 Data Structures in Java: Part 16, The toArray Method, Part 2
2300 Generics in J2SE, Getting Started

Complete Program Listing

A complete listing of the program discussed in this lesson is shown in Listing 45 below.

```
/*File Huffman01.java

This program illustrates the encoding and later decoding of
a text message using the Huffman encoding technique.

This program is provided for educational purposes only. If
you use the program for any purpose, you use it at your
own risk.  The author of the program accepts no
responsibility for any damages that may result from your
use of the program.

We begin by instantiating an object of the HuffmanEncoder
class and invoking the encode method on that object.

Inside the encode method, we invoke the createFreqData
method to create a frequency chart that identifies each of
the individual characters in the original message and the
number of times (frequency) that each character appeared in
the message.

Next, we invoke the createLeaves method to create a
```

HuffLeaf object for each character identified in the frequency chart. We store the HuffLeaf objects in a TreeSet object. Each HuffLeaf object encapsulates the character as well as the number of times that the character appeared in the original message (frequency).

Then we invoke the createHuffTree method to assemble the HuffLeaf objects into a Huffman tree (a HuffTree object). A Huffman tree is a special form of a binary tree consisting of properly linked HuffNode and HuffLeaf objects.

When the createHuffTree method returns, the HuffTree object remains as the only object stored in the TreeSet object that previously contained all of the HuffLeaf objects. This is because all of the HuffLeaf objects have been combined with HuffNode objects to form the tree.

Following that, we invoke the createBitCodes method. This method uses the Huffman tree in a recursive manner to create a bit code for each character in the message. The bit codes are different lengths with the shorter codes corresponding to the characters with a high frequency value and the longer codes corresponding to the characters with the lower frequency values. The createBitCodes method populates a data structure that is required later to decode the encoded message.

At this point, we know the variable-length bitcode that is required to replace each character in the original message to produce a Huffman-encoded message. (The compression provided by Huffman encoding depends on the frequently used characters having short bitcodes and the less frequently used characters having longer bitcodes.)

Although we know the bitcode required to replace each character in the original message, a direct transformation from characters in the message to a stream of contiguous bitcodes is something of a challenge. The computer's memory is organized on 8-bit boundaries. I am unaware of any capability in Java that allows the memory to be viewed simply as a continuous sequence of individual bits.

(Note that it may be possible to accomplish this by using a Java BitSet object. I may give that a try someday when I have the time.)

This program uses a solution to this challenge that is straightford, but is probably inefficient from both a speed and memory requirement viewpoint. The solution is to do a simple table lookup in order to create a long String object consisting of only 1 and 0 characters. Each character in the original message is represented by a substring that matches the required bitcode. This is easy to accomplish because (unlike a long sequence of bits) there are no artificial boundaries requiring the length of the String to be some multiple of a fixed number of characters.

We invoke the encodeToString method to encode the message into a String representation of the bits that will make up the final encoded message.

After the String containing 1 and 0 characters representing the bits in the Huffman-encoded message is created, this String is processed to produce the Huffman-encoded message in a binary bit stream format. This is accomplished using another lookup table containing 256 entries (the number of possible combinations of eight bits).

We invoke the buildEncodingBitMap method to populate a lookup table that relates eight bits represented as a String to every possible combination of eight actual bits.

Then we invoke the encodeStringToBits method to Encode the String representation of the bits that make up the encoded message to the actual bits that make up the encoded message.

Note that this method doesn't handle the end of the message very gracefully for those cases where the number of required bits is not a multiple of 8. The method simply adds enough "0" characters to the end of the String to cause the length to be a multiple of 8. This will usually result in extraneous characters at the end of the decoded message later. Some mechanism must be found to eliminate the extraneous characters when decoding the message. This program assumes that the length of the original message is preserved and provided to the decoding software along with the decoding table. Since the length of the decoded message must match the length of the original message, this value is used to eliminate extraneous characters at the end of the decoded message.

Then we return the binaryEncodedData from the encode method to the main method. The message has now been Huffman encoded. We provide the capability to display the binaryEncodedData in Hexadecimal format at this point for comparison with the original message.

The program continues the demonstration by decoding and displaying the Huffman-encoded message.

We begin the decoding process by instantiating a HuffmanDecoder object.

Then we invoke the decode method on the HuffmanDecoder object to decode the message. We pass the encoded message along with a reference to a data structure containing encoding particulars and the length of the original message so that extraneous characters on the end can be eliminated.

```
Inside the decode method, we invoke the buildDecodingBitMap
method to create a decoding bit map, which is essentially
the reverse of the encoding bit map that was used to encode
the original message.

We invoke the decodeToBitsAsString method to decode the
encoded message from a binary representation to a String of
1 and 0 characters that represent the actual bits in the
encoded message.

We invoke the buildHuffDecodingTable method to create a
Huffman decoding table by swapping the keys and the values
from the Huffman encoding table received as an incoming
parameter by the decode method.

Finally, we invoke the decodeStringBitsToCharacters method
to decode the String containing only 1 and 0 characters
that represent the bits in the encoded message. This
produces a replica of the original message that was
subjected to Huffman encoding.  We write the resulting
decoded message into a String object and return the
decoded message with any extraneous characters at the end
having been removed.

The program was tested using J2SE 5.0 and WinXP.
Requires J2SE 5.0 to support generics.
***********************************************************/

import java.util.*;
import java.io.*;

public class Huffman01{

  public static void main(String[] args){

    //The following data structure is used to
    // communicate encoding particulars from the Huffman
    // encoder to the Huffman decoder.  This is necessary
    // for the decoder to be able to decode the encoded
    // message.  Note that this data structure must be
    // empty when it is passed to the encode method.
    Hashtable <Character,String>huffEncodeTable;

    //Begin the demonstration by applying Huffman encoding
    // to a text message.

    //Create and display the raw text message that will be
    // encoded.  Display 48 characters to the line.

    //Modify the comment indicators to enable one of the
    // following test messages, or insert a test message
    // of your own and then recompile the program.
/*
    //The following test message was copied directly from
    // an Internet news site.  It is probably
    // representative of typical English text.
    String rawData = "BAGHDAD, Iraq Violence increased "
    + "across Iraq after a lull following the Dec. 15 "
    + "parliamentary elections, with at least two dozen "
    + "people including a U.S. soldier killed Monday in "
    + "shootings and bombings mostly targeting the Shiite-"
    + "dominated security services. The Defense Ministry "
    + "director of operations, Brig. Gen. Abdul Aziz "
    + "Mohammed-Jassim, blamed increased violence in the "
    + "past two days on insurgents trying to deepen the "
    + "political turmoil following the elections. The "
    + "violence came as three Iraqi opposition groups "
    + "threatened another wave of protests and civil "
    + "disobedience if allegations of fraud are not "
    + "properly investigated.";
*/
/*
    String rawData = "Now is the time for all good men "
    + "to come to the aid of their country.";
*/

    //Use the following test message or some other
    // similarly short test message to illustrate the
    // construction of the HuffTree object.
    String rawData = "AAAAABBBBCCCDDE";

    //Enable the following two statements to display the
    // raw data 48 characters to the line.
    System.out.println("Raw Data");
    display48(rawData);

    int rawDataLen = rawData.length();

    System.out.println("\nNumber raw data bits: "
                               + rawData.length() * 8);

    //Instantiate a Huffman encoder object
    HuffmanEncoder encoder = new HuffmanEncoder();

    //Encode the raw text message.  The encoded message
    // is received back as bytes stored in an ArrayList
    // object.  Pass the raw message to the encode
    // method.  Also pass a reference to the empty data
    // structure mentioned above to the encode method where
    // it will be populated with encoding particulars
    // needed to decode the message later
    huffEncodeTable = new Hashtable<Character,String>();
    ArrayList<Byte> binaryEncodedData = encoder.encode(
                              rawData,huffEncodeTable);

    System.out.println("Number binary encoded data bits: "
                       + binaryEncodedData.size() * 8);
    System.out.println("Compression factor: "
      + (double)rawData.length()/binaryEncodedData.size());

    //The message has now been Huffman encoded. Display the
    // binaryEncodedData in Hexadecimal format, 48
    // characters per line.
    System.out.println(
            "\nBinary Encoded Data in Hexadecimal Format");
    hexDisplay48(binaryEncodedData);
    System.out.println();

    //Now continue the demonstration by decoding the
    // Huffman-encoded message.

    //Instantiate a Huffman decoder object.
    HuffmanDecoder decoder = new HuffmanDecoder();

    //Pass the encoded message to the decode method of the
    // HuffmanDecoder object.  Also pass a reference
    // to the  data structure containing encoding
    // particulars to the decode method.  Also pass the
    // length of the original message so that extraneous
    // characters on the end of the decoded message can be
    // eliminated.
    String decodedData = decoder.decode(binaryEncodedData,
                                        huffEncodeTable,
                                        rawDataLen);

    //Display the decoded results, 48 characters to the
    // line
    System.out.println("\nDecoded Data");
    display48(decodedData);

  }//end main
  //-------------------------------------------------------//

  //Utility method to display a String 48 characters to
  // the line.
  static void display48(String data){
    for(int cnt = 0;cnt < data.length();cnt += 48){
      if((cnt + 48) < data.length()){
        //Display 48 characters.
        System.out.println(data.substring(cnt,cnt+48));
      }else{
        //Display the final line, which may be short.
        System.out.println(data.substring(cnt));
      }//end else
    }//end for loop
  }//end display48
  //-------------------------------------------------------//

  //Utility method to display hex data 48 characters to
  // the line
  static void hexDisplay48(
                      ArrayList<Byte> binaryEncodedData){
    int charCnt = 0;
    for(Byte element : binaryEncodedData){
      System.out.print(
               Integer.toHexString((int)element & 0X00FF));
      charCnt++;
      if(charCnt%24 == 0){
        System.out.println();//new line
        charCnt = 0;
      }//end if
    }//end for-each
  }//end hexDisplay48
  //-----------------------------------------------------//
}//end class Huffman01
//=====================================================//

//An object of this class can be used to encode a raw text
// message using the Huffman encoding methodology.
class HuffmanEncoder{
  String rawData;
  TreeSet <HuffTree>theTree = new TreeSet<HuffTree>();
  ArrayList <Byte>binaryEncodedData =
                                   new ArrayList<Byte>();
  Hashtable <Character,Integer>frequencyData =
                        new Hashtable<Character,Integer>();
  StringBuffer code = new StringBuffer();
  Hashtable <Character,String>huffEncodeTable;
  String stringEncodedData;
  Hashtable <String,Byte>encodingBitMap =
                              new Hashtable<String,Byte>();
  //-----------------------------------------------------//

  //This method encodes an incoming String message using
  // the Huffman encoding methodology.  The method also
  // receives a reference to an empty data structure.
  // This data structures is populated with encoding
  // particulars required later by the decode method
  // to decode and transform the encoded message back
  // into the original String message.  Note that in
  // order to keep this method simple, pad characters may
  // be appended onto the end of the original
  // message when it is encoded.  This is done to cause the
  // number of bits in the encoded message to be a multiple
  // of eight, thus causing the length of the encoded
  // message to be an integral number of bytes.  Additional
  // code would be required to avoid this at this point.
  // However, it is easy to eliminate the extraneous
  // characters during decoding if the length of the
  // original message is known.
  ArrayList<Byte> encode(
            String rawData,
            Hashtable <Character,String>huffEncodeTable){
    //Save the incoming parameters.
    this.rawData = rawData;
    this.huffEncodeTable = huffEncodeTable;

    //For illustration purposes only, enable the following
    // two statements to display the original message as a
    // stream of bits.  This can be visually compared with
    // a similar display for the encoded  message later to
    // illustrate the amount of compression provided by
    // the encoding process.
/*
```

```
    System.out.println("\nRaw Data as Bits");
    displayRawDataAsBits();
*/
    //Create a frequency chart that identifies each of the
    // individual characters in the original message and
    // the number of times (frequency) that each character
    // appeared in the message.
    createFreqData();

    //For illustration purposes only, enable the following
    // statement to display the contents of the frequency
    // chart created above.
/*
    displayFreqData();
*/
    //Create a HuffLeaf object for each character
    // identified in the frequency chart.  Store the
    // HuffLeaf objects in a TreeSet object.  Each HuffLeaf
    // object encapsulates the character as well as the
    // number of times that the character appeared in the
    // original message (the frequency).
    createLeaves();

    //Assemble the HuffLeaf objects into  a Huffman tree
    // (a HuffTree object). A Huffman tree is a special
    // form of a binary tree  consisting of properly linked
    // HuffNode objects and HuffLeaf objects.
    //When the following method returns, the HuffTree
    // object remains as the only object stored in the
    // TreeSet object that previously contained all of the
    // HuffLeaf objects.  This is because all of the
    // HuffLeaf objects have been combined with HuffNode
    // objects to form the tree.
    createHuffTree();

    //Use the Huffman tree in a recursive manner to create
    // a bit code for each character in the message.  The
    // bit codes are different lengths with the shorter
    // codes corresponding to the characters with a high
    // frequency value and the longer codes corresponding
    // to the characters with the lower frequency values.
    //Note that the method call extracts the reference to
    // the Huffman tree from the TreeSet object and passes
    // that reference to the method.  This is necessary
    // because the method is recursive and cannot
    // conveniently work with the TreeSet object.
    //This method populates the data structure that is
    // required later to decode the encoded message.
    createBitCodes(theTree.first());

    //For purposes of illustration only, enable the
    // following two statements to display a table showing
    // the relationship between the characters in the
    // original message and the bitcodes that will replace
    // those characters to produce the Huffman-encoded
    // message.
```

```
/*
    System.out.println();
    displayBitCodes();
*/
    //Encode the message into a String representation
    // of the bits that will make up the final encoded
    // message.  Also,the following method may optionally
    // display the String showing the bit values that will
    // appear in the final Huffman-encoded message.  This
    // is useful for comparing back against the bits in
    // the original message for purposes of evaluating the
    // amount of compression provided by encoding the
    // message.
    encodeToString();

    //Populate a lookup table that relates eight bits
    // represented as a String to every possible combinaion
    // of eight actual bits.
    buildEncodingBitMap();

    //Encode the String representation of the bits that
    // make up the encoded message to the actual bits
    // that make up the encoded message.
    //Note that this method doesn't handle the end of the
    // message very gracefully for those cases where the
    // number of required bits is not a multiple of 8.  It
    // simply adds enough "0" characters to the end to
    // cause the length to be a multiple of 8.  This may
    // result in extraneous characters at the end of the
    // decoded message later.
    //For illustration purposes only, this method may also
    // display the extended version of the String
    // representation of the bits for comparison with the
    // non-extended version.
    encodeStringToBits();

    //Return the encoded message.
    return binaryEncodedData;
  }//end encode method
  //-----------------------------------------------------//

  //This method displays a message string as a series of
  // characters each having a value of 1 or 0.
  void displayRawDataAsBits(){
    for(int cnt = 0,charCnt = 0;cnt < rawData.length();
                                          cnt++,charCnt++){
      char theCharacter = rawData.charAt(cnt);
      String binaryString = Integer.toBinaryString(
                                             theCharacter);
      //Append leading zeros as necessary to show eight
      // bits per character.
      while(binaryString.length() < 8){
        binaryString = "0" + binaryString;
      }//end while loop
      if(charCnt%6 == 0){
        //Display 48 bits per line.
        charCnt = 0;
        System.out.println();//new line
      }//end if
      System.out.print(binaryString);
    }//end for loop
    System.out.println();
  }//end displayRawDataAsBits
  //-----------------------------------------------------//

  //This method creates a frequency chart that identifies
  // each of the individual characters in the original
  // message and the number of times that each character
  // appeared in the message.  The results are stored in
  // a Hashtable with the characters being the keys and the
  // usage frequency of each character being the
  // corresponding Hashtable value for that key.
  void createFreqData(){
    for(int cnt = 0;cnt < rawData.length();cnt++){
      char key = rawData.charAt(cnt);
      if(frequencyData.containsKey(key)){
        int value = frequencyData.get(key);
        value += 1;
        frequencyData.put(key,value);
      }else{
        frequencyData.put(key,1);
      }//end else
    }//end for loop
  }//end createFreqData
  //-----------------------------------------------------//

  //This method displays the contents of the frequency
  // chart created by the method named createFreqData.
  void displayFreqData(){
    System.out.println("\nFrequency Data");
    Enumeration <Character>enumerator =
                                      frequencyData.keys();
    while(enumerator.hasMoreElements()){
      Character nextKey = enumerator.nextElement();
      System.out.println(
               nextKey + " " + frequencyData.get(nextKey));
    }//end while
  }//end displayFreqData
  //-----------------------------------------------------//

  //This method creates a HuffLeaf object for each char
  // identified in the frequency chart.  The HuffLeaf
  // objects are stored in a TreeSet object.  Each HuffLeaf
  // object encapsulates the character as well as the
  // number of times that the character appeared in the
  // original message.
  void createLeaves(){
    Enumeration <Character>enumerator =
                                      frequencyData.keys();
    while(enumerator.hasMoreElements()){
      Character nextKey = enumerator.nextElement();
      theTree.add(new HuffLeaf(
                     nextKey,frequencyData.get(nextKey)));
    }//end while
  }//end createLeaves
  //-------------------------------------------------------//

  //This inner class is used to construct a leaf object in
  // the Huffman tree.
  class HuffLeaf extends HuffTree{

    private int value;

    //HuffLeaf constructor
    public HuffLeaf(int value, int frequency){
      this.value = value;
      //Note that frequency is inherited from HuffTree
      this.frequency = frequency;
    }//end HuffLeaf constructor

    public int getValue(){
      return value;
    }//end getValue

  }//End HuffLeaf class
  //=======================================================//

  //Assemble the HuffLeaf objects into a HuffTree object.
  // A HuffTree object is a special form of a binary tree
  // consisting of properly linked HuffNode objects and
  // HuffLeaf objects.
  //When the method terminates, the HuffTree object
  // remains as the only object stored in the TreeSet
  // object that previously contained all of the HuffLeaf
  // objects.  This is because all of the HuffLeaf
  // objects have been removed from the TreeSet object
  // and combined with HuffNode objects to form the
  // Huffman tree (as represented by the single HuffTree
  // object).
  //The method contains two sections of code that can be
  // enabled to display:
  // 1. The contents of the original TreeSet object.
  // 2. The contents of the TreeSet object for each
  //    iteration during which HuffLeaf objects are being
  //    combined with HuffNode objects to form the final
  //    HuffTree object.
  // This display is very useful for understanding how the
  // Huffman tree is constructed.  However, this code
  // should be enabled only for small trees because it
  // generates a very large amount of output.

  //The HuffTree object is constructed by:
  // 1. Extracting pairs of HuffLeaf or HuffNode objects
  //    from the TreeSet object in ascending order based
  //    on their frequency value.
  // 2. Using the pair of extracted objects to construct
  //    a new HuffNode object where the two extracted
  //    objects become children of the new HuffNode
  //    object, and where the frequency value stored in
  //    the new HuffNode object is the sum of the
  //    frequency values in the two child objects.
  // 3. Removing the two original HuffLeaf or HuffNode
  //    objects from the TreeSet and adding the new
  //    HuffNode object to the TreeSet object.  The
  //    position of the new HuffNode object in the Treeset
  //    object is determined by its frequency value
  //    relative to the other HuffNode or HuffLeaf objects
  //    in the collection. The new HuffNode object will
  //    eventually become a child of another new HuffNode
  //    object unless it ends up as the root of the
  //    HuffTree object.
  // 4. Continuing this process until the TreeSet object
  //    contains a single object of type HuffTree.
  void createHuffTree(){

    //Enable the following statements to see the original
    // contents of the TreeSet object. Do this only for
    // small trees because it generates lots of output.
/*
    System.out.println("\n\nDisplay Original TreeSet");
    Iterator <HuffTree> originalIter = theTree.iterator();
    while(originalIter.hasNext()){
      System.out.println(
                      "\nHuffNode, HuffLeaf, or HuffTree");
      displayHuffTree(originalIter.next(),0);
    }//end while loop
    //End code to display the TreeSet
*/
    //Iterate on the size of the TreeSet object until all
    // of the elements have been combined into a single
    // element of type HuffTree
    while(theTree.size() > 1){
      //Get, save, and remove the first two elements.
      HuffTree left = theTree.first();
      theTree.remove(left);
      HuffTree right = theTree.first();
      theTree.remove(right);

      //Combine the two saved elements into a new element
      // of type HuffNode and add it to the TreeSet
      // object.
      HuffNode tempNode = new HuffNode(left.getFrequency()
                          + right.getFrequency(),left,right);
      theTree.add(tempNode);

      //Enable the following statements to see the HuffTree
      // being created from HuffNode and HuffLeaf objects.
      // Do this only for small trees because it will
      // generate a lot of output.
/*
      System.out.println("\n\nDisplay Working TreeSet");
      Iterator <HuffTree> workingIter = theTree.iterator();
      while(workingIter.hasNext()){
        System.out.println(
                      "\nHuffNode, HuffLeaf, or HuffTree");
        displayHuffTree(workingIter.next(),0);
      }//end while loop
      //End code to display the TreeSet
*/
    }//end while
  }//end createHuffTree
  //-----------------------------------------------------//

  //Recursive method to display a HufTree object.  The
  // first call to this method should pass a value of 0
  // for recurLevel.
  void displayHuffTree(HuffTree tree,int recurLevel){
    recurLevel++;
    if(tree instanceof HuffNode){
      // This is a node, not a leaf.  Process it as a node.

      //Cast to type HuffNode.
      HuffNode node = (HuffNode)tree;
      // Get and save the left and right branches
      HuffTree left = node.getLeft();
      HuffTree right = node.getRight();

      //Display information that traces out the recursive
      // traversal of the tree in order to display the
      // contents of the leaves.
      System.out.print("  Left to " + recurLevel + " ");
      //Make a recursive call.
      displayHuffTree(left,recurLevel);

      System.out.print("  Right to " + recurLevel + " ");
      //Make a recursive call.
      displayHuffTree(right,recurLevel);

    }else{
      //This is a leaf.  Process it as such.
      //Cast the object to type HuffLeaf.
      HuffLeaf leaf = (HuffLeaf)tree;
      System.out.println(
                        "  Leaf:" + (char)leaf.getValue());
    }//end else

    System.out.print("  Back ");

  }//end displayHuffTree
  //-----------------------------------------------------//
  //This inner class is used to construct a node object in
  // the Huffman tree.
  class HuffNode extends HuffTree{

    private HuffTree left;
    private HuffTree right;

    //HuffNode constructor
    public HuffNode(
               int frequency,HuffTree left,HuffTree right){
```

```
      this.frequency = frequency;
      this.left = left;
      this.right = right;
    }//end HuffNode constructor

    public HuffTree getLeft(){
      return left;
    }//end getLeft

    public HuffTree getRight(){
      return right;
    }//end getRight

  }//end HuffNode class
  //=======================================================//

  //This method uses the Huffman tree in a recursive manner
  // to create a bitcode for each character in the message.
  // The bitcodes are different lengths with the shorter
  // bitcodes corresponding to the characters with a high
  // usage frequency value and the longer bitcodes
  // corresponding to the characters with the lower
  // frequency values.
  //Note that this method receives a reference to the
  // Huffman tree that was earlier contained as the only
  // object in the TreeSet object.  (Because this method is
  // recursive, it cannot conveniently work with the
  // TreeSet object.

  //This method creates a Huffman encoding table as a
  // Hashtable object that relates the variable length
  // bitcodes to the characters in the original message.
  // The bitcodes are constructed as objects of type
  // StringBuffer consisting of sequences of the characters
  // 1 and 0.
  //Each bitcode describes the traversal path from the root
  // of the Huffman tree to a leaf on that tree.  Each time
  // the path turns to the left, a 0 character is appended
  // onto the StringBuffer object and becomes part of the
  // resulting bitcode.  Each time the path turns to the
  // right, a 1 character is appended onto the
  // StringBuffer object.  When a leaf is reached, the
  // character stored in that leaf is retrieved and put
  // into the Hashtable object as a key.  A String
  // representation of the StringBuffer object is used as
  // the value for that key in the Hashtable.
  //At completion,the Hashtable object contains a series of
  // keys consisting of the original characters in the
  // message and a series of values as String objects
  // (consisting only of 1 and 0 characters) representing
  // the bitcodes that will eventually be used to encode
  // the original message.
  //Note that theHashtable object that is populated by this
  // method is the data structure that is required later
  // to decode the encoded message.
  void createBitCodes(HuffTree tree){
    if(tree instanceof HuffNode){
      // This is a node, not a leaf.  Process it as a node.

      //Cast to type HuffNode.
      HuffNode node = (HuffNode)tree;
      // Get and save the left and right branches
      HuffTree left = node.getLeft();
      HuffTree right = node.getRight();

      //Append a 0 onto the StringBuffer object.  Then make
      // a recursive call to this method passing a
      // reference to the left child as a parameter.  This
      // recursive call will work its way all the way down
      // to a leaf before returning.  Then it will be time
      // to process the right path.
      code.append("0");
      createBitCodes(left);

      //Return to here from recursive call on left child.

      //Delete the 0 from the end of the StringBuffer
      // object to restore the contents of that object to
      // the same state that it had before appending the 0
      // and making the recursive call on the left branch.
      //Now we will make a right turn.  Append a 1 to the
      // StringBuffer object and make a recursive call to
      // this method passing a reference to the right child
      // as a parameter.  Once again, this recursive call
      // will work its way all the  way down to a leaf
      // before returning.
      code.deleteCharAt(code.length() - 1);//Delete the 0.
      code.append("1");
      createBitCodes(right);

      //Return to here from recursive call on right child.

      //Delete the character most recently appended to the
      // StringBuffer object and return from this recursive
      // call to the method.  The character is deleted
      // because control is being transferred back one
      // level in the recursive process and the
      // StringBuffer object must be restored to the same
      // state that it had when this recursive call was
      // made.
      code.deleteCharAt(code.length() - 1);
    }else{
      //This is a leaf.  Process it as such.
      //Cast the object to type HuffLeaf.
      HuffLeaf leaf = (HuffLeaf)tree;

      //Put an entry into the Hashtable.  The Hashtable
      // key consists of the character value stored in the
      // leaf. The value in the Hashtable consists of the
      // contents of the StringBuffer object representing
      // the path from the root of the tree to the leaf.
      // This is the bitcode and is stored in the Hashtable
      // as a String consisting of only 1 and 0 characters.
      huffEncodeTable.put((char)(
                         leaf.getValue()),code.toString());
    }//end else

  }//end createBitCodes
  //-----------------------------------------------------//

  //This method displays a table showing the relationship
  // between the characters in the original message and the
  // bitcodes that will ultimately replace each of those
  // characters to produce the Huffman-encoded message.
  void displayBitCodes(){
    System.out.println(
           "\nMessage Characters versus Huffman BitCodes");
    Enumeration <Character>enumerator =
                                huffEncodeTable.keys();
    while(enumerator.hasMoreElements()){
      Character nextKey = enumerator.nextElement();
      System.out.println(
             nextKey + " " + huffEncodeTable.get(nextKey));
    }//end while
  }//end displayBitCodes
  //-----------------------------------------------------//

  //This method encodes the message into a String
  // representation of the bits that will make up the final
  // encoded message.  The String consists of only 1 and 0
  // characters where each character represents the state
  // of one of the bits in the Huffman-encoded message.
  //Also for illustration purposes, this method optionally
  // displays the String showing the bit values that will
  // appear in the Huffman-encoded message.
  void encodeToString(){
    StringBuffer tempEncoding = new StringBuffer();
    for(int cnt = 0;cnt < rawData.length();cnt++){
      //Do a table lookup to get the substring that
      // represents the bitcode for each message character.
      // Append those substrings to the string that
      // represents the Huffman-encoded message.
      tempEncoding.append(huffEncodeTable.get(
                                 rawData.charAt(cnt)));
    }//end for loop

    //Convert the StringBuffer object to a String object.
    stringEncodedData = tempEncoding.toString();

    //For illustration purposes, enable the following two
    // statements to display the String showing the bit
    // values that will appear in the Huffman-encoded
    // message.  Display 48 bits to the line except for
    // the last line, which may be shorter, and which may
    // not be a multiple of 8 bits.
/*
    System.out.println("\nString Encoded Data");
    display48(stringEncodedData);
```

```
*/
  }//end encodeToString
  //-----------------------------------------------------//

  //This method populates a lookup table that relates eight
  // bits represented as a String to eight actual bits for
  // all possible combinations of eight bits.
  void buildEncodingBitMap(){

    for(int cnt = 0; cnt <= 255;cnt++){
      StringBuffer workingBuf = new StringBuffer();
      if((cnt & 128) > 0){workingBuf.append("1");
        }else{workingBuf.append("0");};
      if((cnt & 64) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 32) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 16) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 8) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 4) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 2) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 1) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      encodingBitMap.put(workingBuf.toString(),
                                          (byte)(cnt));
    }//end for loop
  }//end buildEncodingBitMap
  //-----------------------------------------------------//

  //The purpose of this method is to create actual bit data
  // that matches the 1 and 0 characters in the
  // stringEncodedData that represents bits with the 1 and
  // 0 characters.
  //Note that this method doesn't handle the end of the
  // data very gracefully for those cases where the number
  // of required bits is not a multiple of 8.  It simply
  // adds enough "0" characters to the end to cause the
  // length to be a multiple of 8.  This may result in
  // extraneous characters at the end of the decoded
  // message later. However, it isn't difficult to remove
  // the extraneous characters at decode time as long as
  // the length of the original message is known.
  //For illustration purposes, this method may optionally
  // display the extended version of the stringEncodedData
  // for comparison with the non-extended version.
  //Note that the binary Huffman-encoded data produced by
  // this method is stored in a data structure of type
  // ArrayList <Byte>.
  void encodeStringToBits(){
    //Extend the length of the stringEncodedData to cause
    // it to be a multiple of 8.
    int remainder = stringEncodedData.length()%8;
    for(int cnt = 0;cnt < (8 - remainder);cnt++){
      stringEncodedData += "0";
    }//end for loop

    //For illustration purposes only, enable the following
    // two statements to display the extended
    // stringEncodedData in the same format as the
    // original stringEncodedData.
/*
    System.out.println("\nExtended String Encoded Data");
    display48(stringEncodedData);
*/
    //Extract the String representations of the required
    // eight bits.  Generate eight actual matching bits by
    // looking the bit combination up in a table.
    for(int cnt = 0;cnt < stringEncodedData.length();
                                                  cnt += 8){
      String strBits  = stringEncodedData.substring(
                                                 cnt,cnt+8);
      byte realBits = encodingBitMap.get(strBits);
      binaryEncodedData.add(realBits);
    }//end for loop
  }//end encodeStringToBits
  //-----------------------------------------------------//

  //Method to display a String 48 characters to the line.
  void display48(String data){
    for(int cnt = 0;cnt < data.length();cnt += 48){
      if((cnt + 48) < data.length()){
        //Display 48 characters.
        System.out.println(data.substring(cnt,cnt+48));
      }else{
        //Display the final line, which may be short.
        System.out.println(data.substring(cnt));
      }//end else
    }//end for loop
  }//end display48
  //-----------------------------------------------------//

}//end HuffmanEncoder class
//=======================================================//


//An object of this class can be used to decode a
// Huffman-encoded message given the encoded message,
// a data structure containing particulars as to how the
// original message was encoded, and the length of the
// original message..
class HuffmanDecoder{
  Hashtable <String,Character>huffDecodeTable =
                         new Hashtable<String,Character>();
  String stringDecodedData;
  String decodedData = "";
  Hashtable <Byte,String>decodingBitMap =
                              new Hashtable<Byte,String>();
  ArrayList <Byte>binaryEncodedData;
```

```
//The following structure contains particulars as to how
// the original message was encoded, and must be received
// as an incoming parameter to the decode method along
// with the encoded message and the length of the
// original message.
Hashtable <Character,String>huffEncodeTable;
//Used to eliminate the extraneous characters on the end.
int rawDataLen;
//-----------------------------------------------------//

//This method receives a Huffman-encoded message along
// with a data structure containing particulars as to how
// the original message was encoded and the length of the
// original message.  It decodes the original message and
// returns the decoded version as a String object.
String decode(ArrayList <Byte>binaryEncodedData,
             Hashtable <Character,String>huffEncodeTable,
             int rawDataLen){
  //Save the incoming parameters.
  this.binaryEncodedData = binaryEncodedData;
  this.huffEncodeTable = huffEncodeTable;
  this.rawDataLen = rawDataLen;

  //Create a decoding bit map, which is essentially the
  // reverse of the encoding bit map that was used to
  // encode the original message.
  buildDecodingBitMap();

  //Decode the encoded message from a binary
  // representation to a String of 1 and 0 characters
  // that represent the actual bits in the encoded
  // message.  Also, for illustration purposes only,
  // this method may optionally display the String.
  decodeToBitsAsString();

  //Create a Huffman decoding table by swapping the keys
  // and values from the Huffman encoding table received
  // as an incoming parameter by the decode method.
  buildHuffDecodingTable();

  //Decode the String containing only 1 and 0 characters
  // that represent the bits in the encoded message. This
  // produces a replica of the original message that was
  // subjected to Huffman encoding.  Write the resulting
  // decoded message into a String object referred to by
  // decoded data.
  decodeStringBitsToCharacters();

  //Return the decoded message.  Eliminate the extraneous
  // characters from the end of the message on the basis
  // of the known length of the original message.
  return decodedData.substring(0,rawDataLen);
}//end decode method
//-----------------------------------------------------//
```

```
  //This method populates a lookup table that relates eight
  // bits represented as a String to eight actual bits for
  // all possible combinations of eight bits.  This is
  // essentially a reverse lookup table relative to the
  // encodingBitMap table that is used to encode the
  // message.  The only difference between the two is a
  // reversal of the key and the value in the Hashtable
  // that contains the table.

  void buildDecodingBitMap(){
    for(int cnt = 0; cnt <= 255;cnt++){
      StringBuffer workingBuf = new StringBuffer();
      if((cnt & 128) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 64) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 32) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 16) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 8) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 4) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 2) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 1) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      decodingBitMap.put((byte)(cnt),workingBuf.
                                              toString());
    }//end for loop
  }//end buildDecodingBitMap()
  //-----------------------------------------------------//

  //This method decodes the encoded message from a binary
  // representation to a String of 1 and 0 characters that
  // represent the actual bits in the encoded message.
  // Also, for illustration purposes only, this method
  // may optionally display that String.
  void decodeToBitsAsString(){
    StringBuffer workingBuf = new StringBuffer();

    for(Byte element : binaryEncodedData){
      byte wholeByte = element;
      workingBuf.append(decodingBitMap.get(wholeByte));
    }//end for-each

    //Convert from StringBuffer to String
    stringDecodedData = workingBuf.toString();

    //For illustration purposes only, enable the following
    // two statements to display the String containing 1
    // and 0 characters that represent the bits in the
    // encoded message.
/*
    System.out.println("\nString Decoded Data");
    display48(stringDecodedData);
*/
  }//end decodeToBitsAsString
  //-----------------------------------------------------//

  //This method creates a Huffman decoding table by
  // swapping the keys and the values from the Huffman
  // encoding table received as an incoming parameter by
  // the decode method.
  void buildHuffDecodingTable(){
    Enumeration <Character>enumerator =
                                huffEncodeTable.keys();
    while(enumerator.hasMoreElements()){
      Character nextKey = enumerator.nextElement();
      String nextString = huffEncodeTable.get(nextKey);
      huffDecodeTable.put(nextString,nextKey);
    }//end while
  }//end buildHuffDecodingTable()
  //-----------------------------------------------------//

  //The method begins with an empty StringBuffer object
  // referred to by the variable named workBuf and another
  // empty StringBuffer object referred to by the variable
  // named output.  The StringBuffer object referred to by
  // output is used to construct the decoded message.  The
  // StringBuffer object referred to by workBuf is used as
  // a temporary holding area for substring data.
  //The method reads the String containing only 1 and 0
  // characters that represent the bits in the encoded
  // message (stringDecodedData).  The characters are read
  // from this string one character at a time.  As each new
  // character is read, it is appended to the StringBuffer
  // object referred to by workBuf.
  //As each new character is appended to the StringBuffer
  // object, a test is performed to determine if the
  // current contents of the StringBuffer object match one
  // of the keys in a lookup table that relates strings
  // representing Huffman bitcodes to characters in the
  // original message.
  //When a match is found, the value  associated with that
  // key is extracted and appended to the StringBuffer
  // object referred to by output.  Thus, the output text
  // is built up one character at a time.
  //Having processed the matching key, A new empty
  // StringBuffer object is instantiated, referred to by
  // workBuf, and the process of reading, appending, and
  // testing for a match is repeated until all of the
  // characters in the string that represents the bits in
  // the encoded message have been processed.  At that
  // point, the StringBuffer object referred to by output
  // contains the entire decoded message.  It is converted
  // to type String and written into the object referred to
  // by decodedData.  Then the method returns with the task
  // of decoding the encoded message having been completed.
  void decodeStringBitsToCharacters(){
    StringBuffer output = new StringBuffer();
    StringBuffer workBuf = new StringBuffer();

    for(int cnt = 0;cnt < stringDecodedData.length();
                                                    cnt++){
      workBuf.append(stringDecodedData.charAt(cnt));
      if(huffDecodeTable.containsKey(workBuf.toString())){
        output.append(
                  huffDecodeTable.get(workBuf.toString()));
        workBuf = new StringBuffer();
      }//end if
    }//end for loop

    decodedData = output.toString();
  }//End decodeStringBitsToCharacters();
  //-----------------------------------------------------//

  //Method to display a String 48 characters to the line.
  void display48(String data){
    for(int cnt = 0;cnt < data.length();cnt += 48){
      if((cnt + 48) < data.length()){
        //Display 48 characters.
        System.out.println(data.substring(cnt,cnt+48));
      }else{
        //Display the final line, which may be short.
        System.out.println(data.substring(cnt));
      }//end else
    }//end for loop
  }//end display48
  //-----------------------------------------------------//

}//end HuffmanDecoder class
//=======================================================//

//This class is the abstract superclass of the
// HuffNode and HuffLeaf classes.  Objects instantiated
// from HuffNode and HuffLeaf are populated and used to
// create a Huffman tree.
abstract class HuffTree implements Comparable{

  int frequency;

  public int getFrequency(){
    return frequency;
  }//end getFrequency

  //This method compares this object to an object whose
  // reference is received as an incoming parameter.
  // The method guarantees that sorting processes that
  // depend on this method, such as TreeSet objects, will
  // sort the objects into a definitive order.

  // If the frequency values of the two objects are
  // different, the sort is based on the frequency values.
  // If the frequency values are equal, the objects are
  // sorted based on their relative hashCode values.
  // Thus, if the same two objects with the same frequency
  // value are compared two or more times during the
  // execution of the program, those two objects will
  // always be sorted into the same order.  There is no
  // chance of an ambiguous tie as to which object
  // should be first except for the case where an object
  // is compared to itself using two references to the
  // same object.
  public int compareTo(Object obj){
    HuffTree theTree = (HuffTree)obj;
    if (frequency == theTree.frequency){
      //The objects are in a tie based on the frequency
      // value.  Return a tiebreaker value based on the
      // relative hashCode values of the two objects.
      return (hashCode() - theTree.hashCode());
    }else{
      //Return negative or positive as this frequency is
      // less than or greater than the frequency value of
      // the object referred to by the parameter.
      return frequency - theTree.frequency;
    }//end else
  }//end compareTo

}//end HuffTree class
//=======================================================//
```

Listing 45

About the author

Richard Baldwin is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in

DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

Keywords
java Huffman Lempel Ziv LZ77 lossless compression algorithm LZW LZSS DEFLATE tuple

-end-