pdf文档格式笔记

# Pdf优势

1. 概述

PDF文件是一种编程形式的文档格式，它所有显示的内容，都是通过相应的操作符进行绘制的。

PDF基本显示单元包括：文字，图片，矢量图，图片

PDF扩展单元包括：水印，电子署名，注释，表单，多媒体，3D

PDF动作单元：书签，超链接（初学者可以简单这么理解，对于研究比较深入的人员来说，拥有动作的单元有很多个，包括电子署名，多媒体，3D等等）

1. 优势

一致性：在所有可以打开PDF的机器上，展示的效果是完全一致，不会出现段落错乱、文字乱码这些排版问题，可以把辛苦排好版的文档原汁原味的发送给他人查看，完美的展示自己的劳动成果，尤其是文档中，本身可以嵌入字体，避免了客户端没有对应字体，而导致文字显示不一致的问题。所以，在印刷行业，绝大多数用的都是PDF格式。

不易修改：用过PDF文件的人，都会知道，对已经保存之后的PDF文件，想要进行重新排版，基本上就不可能的，这就保证了从资料源发往外界的资料，不容易被篡改。

安全性： PDF文档可以进行加密，包括以下几种加密形式：文档打开密码，文档权限密码，文档证书密码，加密的方法包括：RC4，AES，通过加密这种形式，可以达到资料防扩散等目的。

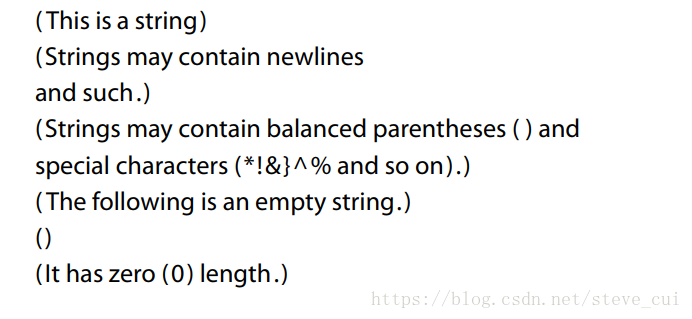
不失真： PDF文件中，使用了矢量图，在文件浏览时，无论放大多少倍，都不会导致使用矢量图绘制的文字，图案的失真。

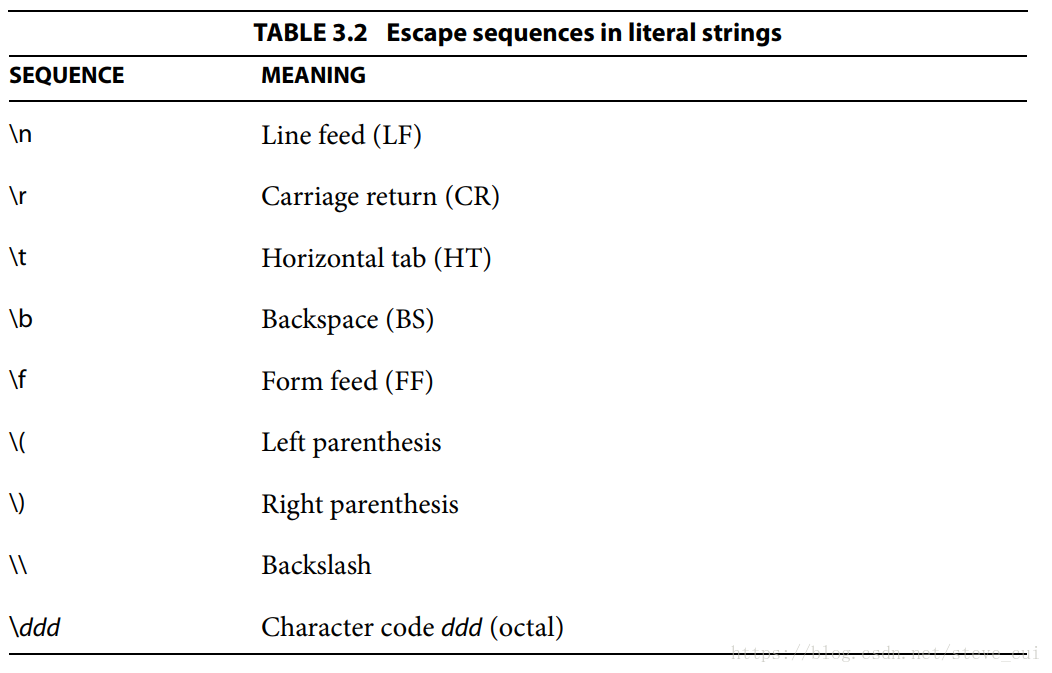
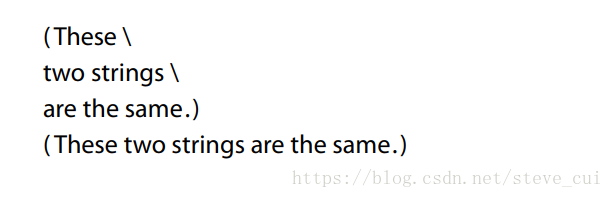
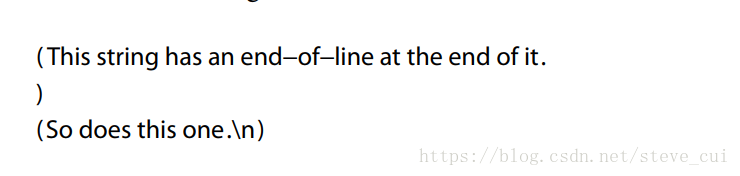
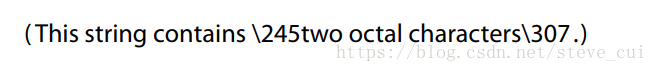
压缩：为了减少PDF文件的size，PDF格式支持各种压缩方式： asciihex，ascii85，lzw，runlength，ccitt，jbig2，jpeg（DCT），jpeg2000（jpx）

支持多种印刷标准：支持PDF-A，PDF-X

# PDF 基本格式规范

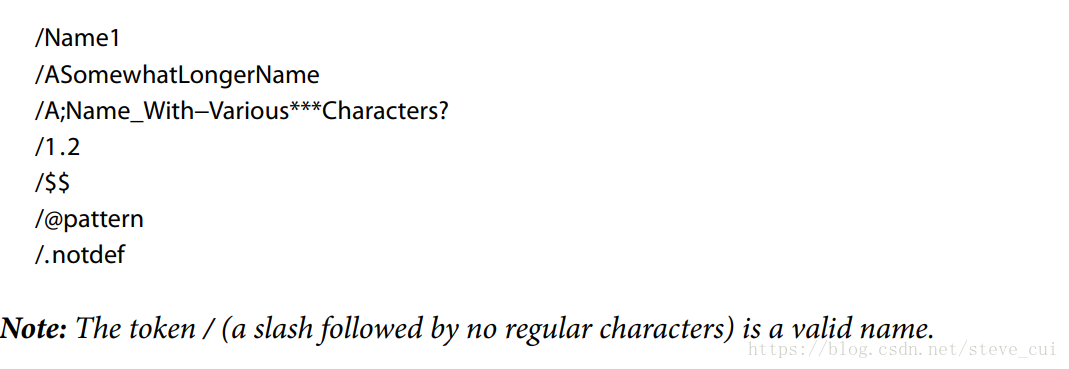
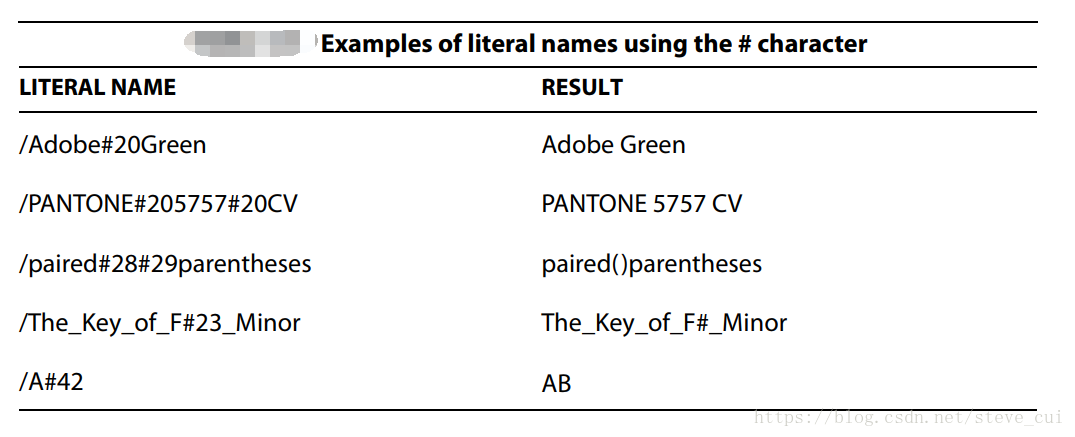
PDF文件是由对象集合组成的，包括：boolean（布尔型），numberic（数值型），string（字符串型），name（名字型），array（数组型），dictionary（字典型），stream（数据流型），null（空类型）， indirect（间接型）。

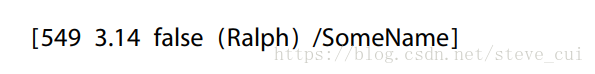
1. **boolean** PDF支持布尔型对象，对应的关键字为“true”和“false”，该对象可以存在于数组型对象、字典型对象。
2. **numberic** PDF支持两种类型的数值型对象：integer（整数）和real（浮点型）
   1. integer（整数）：123，-123，+456，0，234234
   2. real（浮点型）：1.34，-1.23，+123.4，2.，-.01，0.0
3. **string** 字符串对象可以写成以下两种形式：
   1. **literal string（文字串）**：该对象由“（”和“）”作为对象的开始和结束，内容为ascii字符组成的序列，以下的文字串都是有效对象： 

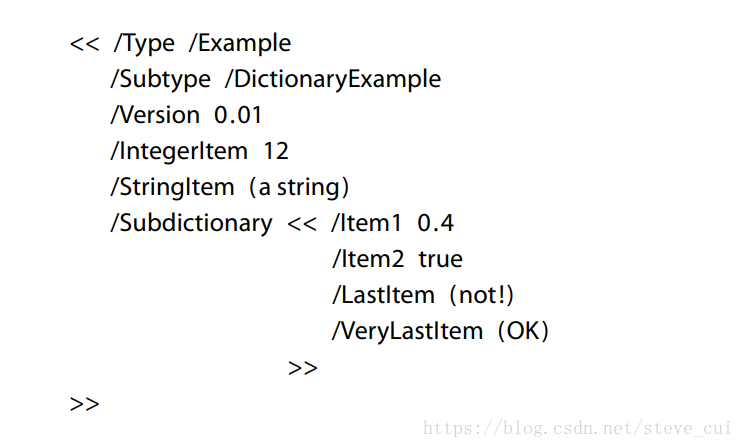
在文字串中，可以使用“\”反斜线，用于转义字符，下面列出PDF支持的转义字符，如果不在该表格中的字符，则忽略反斜线“\”：  如果一个文字串很长时，我们可以使用反斜线“\”，把这个字符串分成多行。  如果文字串的结尾是换行符，则等同于“\n”  “\ddd”这种表现形式，用来不能打印的ascii字符，“d”为八进制数值，“ddd”可以是一个，两个，三个，当高位是“0”，则忽略，当“ddd”后一个字符是数字，则前面的“0”不可忽略，如“\0053”，等价于“\005”+“3”。 

1. **hexadecimal（十六进制字符串）**：该对象由“<”和“>”作为对象的开始和结束，内容为十六进制数组成的序列（0-9，a-f或A-F）。 十六进制字符串

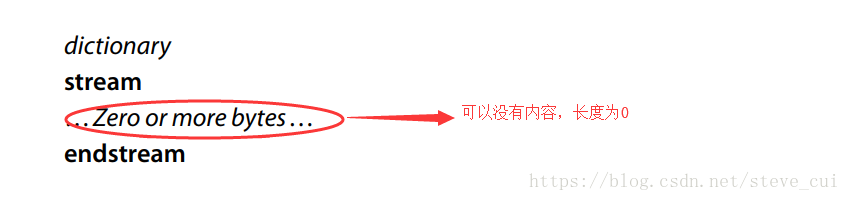
十六进制字符串的长度一定是偶数的，每两个数字表示一个字符，当出现为奇数的情况，PDF则在末尾补“0”，如：错误的十六进制数<980AC>，则被识别为<980AC0>。 当十六进制字符串中存在空白字符（空格，tab，回车，换行，换页符），则忽略空白字符。

1. **name** 名字型对象，是由“/”作为首字符的字符串组成，“/”只是用来标识名字对象，并不属于名字对象的内容，而且名字对象字符串中不允许出现空白符号。当两个名字型对象内容完全一致（区分大小写），则PDF认为这两个对象相同。  从PDF1.2以后版本，名字对象可以使用”#”+2个16进制数字，来表示那些不能显示的ascii码，值得注意的是，名字对象“/A#42”，它的长度是3，而不是5 

**6. array** 数组对象是一个容器集合，和java中的数组对象很相似。它的元素可以是PDF支持的任意对象类型，该对象由“[”和“]”作为对象的起始符号。  PDF中的数组对象只支持一维数组，但是可以通过数组嵌套来表示多维数组。

**7. dictionary** 字典对象是键值对的集合，和java中的map类似。key是唯一的，key的类型是name对象，value的类型是任意PDF支持的对象类型，当value为“null”，则表示该键值对不存在。字典对象以“<<”和“>>”作为对象的开始和结束。 

字典对象是构建PDF文档的主要结构，通常它们都是一些有特定意义的属性组成的复杂对象集合，一般每个字典中都包含“Type”名字对象，该对象的值表示字典对象描述的具体对象，如“Page”，表示该字典是页对象，“Outline”，表示该字典对象是书签对象。

**8. stream** 数据流对象，和java中的字符流对象类似，没有长度限制。 数据流对象由一个字典对象和关键字“stream”，“endstream”之间的数据块组成。   
所有的stream必须是indirect对象，而stream中的字典必须是direct对象。

关键字“stream”在stream对象字典内容之后，再下来是stream数据块，后面是“endstream”，stream数据块的前面和后面分别使用换行符，将“stream”和“endstream”分隔开，注意：分隔符没有计算在stream的长度内。

从PDF1.2开始，数据块可以被保存在一个外部文件（通过stream字典指定），当数据块被存储到外部文件时，“stream”与“endstream”中间的数据块将被忽略。

每个stream对象的字典都有“Length”属性，该属性表示数据块的长度（如果stream被压缩了，则表示的是压缩后的长度）。

通常stream对象的字典都包含以下几个属性：

**Length**

类型：整型

描述：（必须）数据块长度

**Filter**

类型：名字或数组

描述：（可选）对数据块进行压缩，名字类型，表示按照该名字标识的压缩方式去压缩，数组型，表示采用多重压缩，根据数组中名字的先后顺序进行压缩

**DecodeParms**

类型：字典或数组

描述：（可选）“Filter”指定压缩方式的一个参数字典，或一组参数字典（数组）。该参数必须被设置，除非使用默认值。如果“Filter”是数组对象时，DecodeParms也必须是数组对象。

**F**

类型：文件描述对象

描述：（可选，PDF1.2以上）stream数据块的内容将被保存在一个文件中。如果该属性存在，“stream”和“endstream”中的数据块将被忽略，“Filter”和“DecodeParms”也将被忽略，而使用“FFilter”和“FDecodeParms”替代，这时，“Length”的值通常为“0”

**FFilter**

类型：名字或数组

描述：（可选，PDF1.2以上）与“Filter”的规则相同，它用来描述外部文件数据

**FDecodeParms**

类型：字典或数组

描述：（可选，PDF1.2以上）与“DecodeParms”的规则相同，它用来描述外部文件数据

**DL**

类型：整型

描述：（可选，PDF1.5以上）非负整数，它表示解压缩之后的数据块长度。

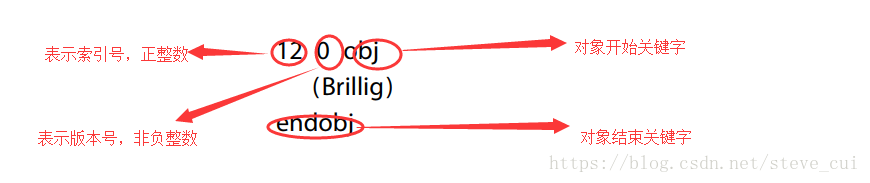
**Null**

空对象，类似与java中“null“，表示空值。

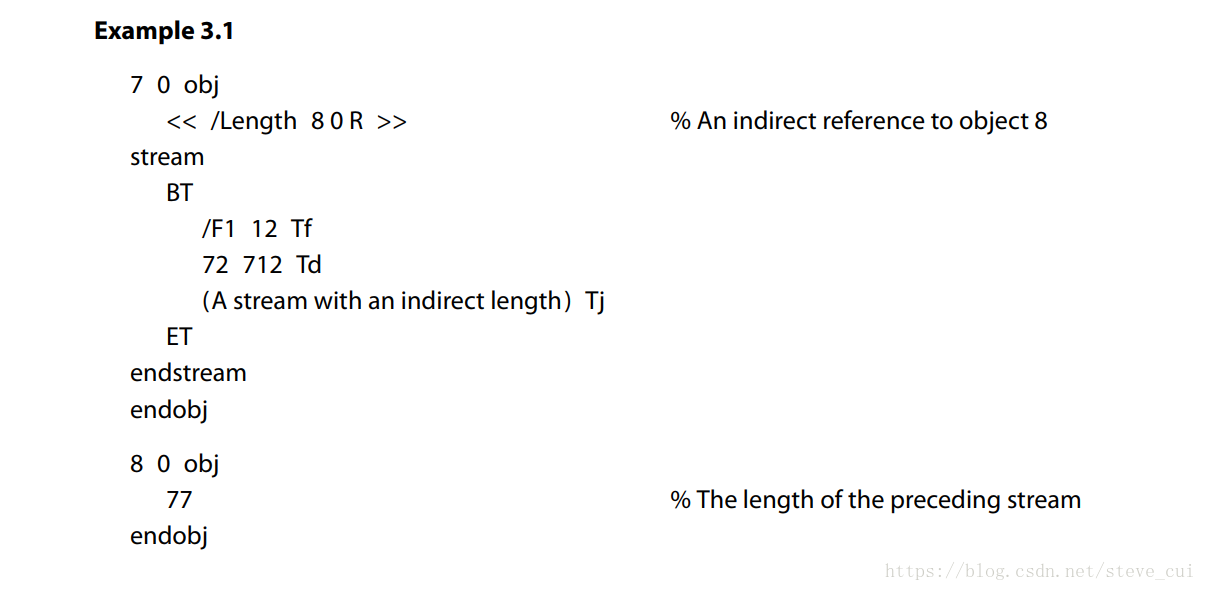
**indirect**

间接对象，PDF中的任何对象都可以封装成一个间接引用对象。

该类型对象，由一个对象号（索引号），一个版本号，”obj“关键字，”endobj“关键字组成。



一个间接对象，可以通过间接引用（如：12 0 R），引用到任何位置。



本章对PDF中的各种数据对象进行了简单的描述，都是一些通用的规则，在后面的章节当中，会更深入的对一些拥有特殊意义对象进行分析解答。

注：本系列博文会用到很多《PDF reference sixth edition》的例子和表格，大家可以结合之看会更有效果。

# filter的语法规则

在PDF中，为了让文件变的更小，通常的做法是，将stream对象进行压缩，因为stream对象的数据块比较大，所以，是重点关注的地方。

从PDF1.5开始往后的版本，支持了Object stream，可以把多个对象（非stream对象）放到同一个stream对象中，并进行压缩，达到减少文件大小的效果。

Filter大部分的类型是压缩，当然有一个是特殊的类型”Crypt“，这个类型是对该stream进行单独加密。

一个stream对象，可以单次压缩，也可以多次压缩。/Filter [/ASCII85Decode /LZWDecode]，表示被描述的这个stream对象进行了ASCII85Decode和LZWDecode两次压缩，因此，对该stream进行解压缩的时候，也要按照反顺序解，分别压缩。

PDF支持的filter可以分为三大类：

一、ASCII filters（ASCIIHexDecode、ASCII85Decode），这种编码类型可以将8位二进制数据编码成ASCII文本。注意：这种类型的filter不能用在被加密的PDF文档中。

二、加密filters（LZWDecode、FlateDecode、RunLengthDecode、CCITTFaxDecode、JBIG2Decode、DCTDecode，JPXDecode），这些压缩类型包括无损压缩和有损压缩。

三、加密filter（Crypt）

下面咱们来详细介绍每一种filter的用法和作用：

|  |
| --- |
| ***ASCIIHexDecode*** 将数据编码成ASCII十六进制形式，通常就是将二进制数据（如图片数据），编码成7位ASCII字符串。  算法如下（itext）:  /\*\* Decodes a stream that has the ASCIIHexDecode filter.  \* @param in the input data  \* @return the decoded data  \*/  public static byte[] ASCIIHexDecode(final byte in[]) {  ByteArrayOutputStream out = new ByteArrayOutputStream();  boolean first = true;  int n1 = 0;  for (int k = 0; k < in.length; ++k) {  int ch = in[k] & 0xff;  if (ch == '>')  break;  if (PRTokeniser.isWhitespace(ch))  continue;  int n = PRTokeniser.getHex(ch);  if (n == -1)  throw new RuntimeException(MessageLocalization.getComposedMessage("illegal.character.in.asciihexdecode"));  if (first)  n1 = n;  else  out.write((byte)((n1 << 4) + n));  first = !first;  }  if (!first)  out.write((byte)(n1 << 4));  return out.toByteArray();  } |

|  |
| --- |
| ***ASCII85Decode*** 作用与ASCIIHexDecode差不多，通常，ASCII85Decode建议优先使用，因为ASCII85Decode拥有更高的压缩比。ASCIIHexDecode的压缩比为4:5，ASCII85Decode的压缩比为1:2。  算法如下（itext）:  /\*\* Decodes a stream that has the ASCII85Decode filter.  \* @param in the input data  \* @return the decoded data  \*/  public static byte[] ASCII85Decode(final byte in[]) {  ByteArrayOutputStream out = new ByteArrayOutputStream();  int state = 0;  int chn[] = new int[5];  for (int k = 0; k < in.length; ++k) {  int ch = in[k] & 0xff;  if (ch == '~')  break;  if (PRTokeniser.isWhitespace(ch))  continue;  if (ch == 'z' && state == 0) {  out.write(0);  out.write(0);  out.write(0);  out.write(0);  continue;  }  if (ch < '!' || ch > 'u')  throw new RuntimeException(MessageLocalization.getComposedMessage("illegal.character.in.ascii85decode"));  chn[state] = ch - '!';  ++state;  if (state == 5) {  state = 0;  int r = 0;  for (int j = 0; j < 5; ++j)  r = r \* 85 + chn[j];  out.write((byte)(r >> 24));  out.write((byte)(r >> 16));  out.write((byte)(r >> 8));  out.write((byte)r);  }  }  int r = 0;  // We'll ignore the next two lines for the sake of perpetuating broken PDFs  // if (state == 1)  // throw new RuntimeException(MessageLocalization.getComposedMessage("illegal.length.in.ascii85decode"));  if (state == 2) {  r = chn[0] \* 85 \* 85 \* 85 \* 85 + chn[1] \* 85 \* 85 \* 85 + 85 \* 85 \* 85 + 85 \* 85 + 85;  out.write((byte)(r >> 24));  }  else if (state == 3) {  r = chn[0] \* 85 \* 85 \* 85 \* 85 + chn[1] \* 85 \* 85 \* 85 + chn[2] \* 85 \* 85 + 85 \* 85 + 85;  out.write((byte)(r >> 24));  out.write((byte)(r >> 16));  }  else if (state == 4) {  r = chn[0] \* 85 \* 85 \* 85 \* 85 + chn[1] \* 85 \* 85 \* 85 + chn[2] \* 85 \* 85 + chn[3] \* 85 + 85;  out.write((byte)(r >> 24));  out.write((byte)(r >> 16));  out.write((byte)(r >> 8));  }  return out.toByteArray();  } |
|  |
| ***LZWDecode***  LZW（Lempel-Ziv-Welch）是一种可变长度的自适应压缩方法，作为TIFF（Tag Image File Format）标准的压缩方式之一。该压缩方式可以压缩二进制数据和ASCII文本，最终生成的数据是二进制的。LZW具体压缩原理就不在这里赘述了，百度很容易找到。  算法如下（itext）:  /\*\*  \* Handles LZWDECODE filter  \*/  private static class Filter\_LZWDECODE implements FilterHandler{  public byte[] decode(byte[] b, PdfName filterName, PdfObject decodeParams, PdfDictionary streamDictionary) throws IOException {  b = PdfReader.LZWDecode(b);  b = PdfReader.decodePredictor(b, decodeParams);  return b;  }  }  /\*\*  \* A class for performing LZW decoding.  \*  \*  \*/  public class LZWDecoder {  byte stringTable[][];  byte data[] = null;  OutputStream uncompData;  int tableIndex, bitsToGet = 9;  int bytePointer, bitPointer;  int nextData = 0;  int nextBits = 0;  int andTable[] = {  511,  1023,  2047,  4095  };  public LZWDecoder() {  }  /\*\*  \* Method to decode LZW compressed data.  \*  \* @param data The compressed data.  \* @param uncompData Array to return the uncompressed data in.  \*/  public void decode(byte data[], OutputStream uncompData) {  if(data[0] == (byte)0x00 && data[1] == (byte)0x01) {  throw new RuntimeException(MessageLocalization.getComposedMessage("lzw.flavour.not.supported"));  }  initializeStringTable();  this.data = data;  this.uncompData = uncompData;  // Initialize pointers  bytePointer = 0;  bitPointer = 0;  nextData = 0;  nextBits = 0;  int code, oldCode = 0;  byte string[];  while ((code = getNextCode()) != 257) {  if (code == 256) {  initializeStringTable();  code = getNextCode();  if (code == 257) {  break;  }  writeString(stringTable[code]);  oldCode = code;  } else {  if (code < tableIndex) {  string = stringTable[code];  writeString(string);  addStringToTable(stringTable[oldCode], string[0]);  oldCode = code;  } else {  string = stringTable[oldCode];  string = composeString(string, string[0]);  writeString(string);  addStringToTable(string);  oldCode = code;  }  }  }  }  /\*\*  \* Initialize the string table.  \*/  public void initializeStringTable() {  stringTable = new byte[8192][];  for (int i=0; i<256; i++) {  stringTable[i] = new byte[1];  stringTable[i][0] = (byte)i;  }  tableIndex = 258;  bitsToGet = 9;  }  /\*\*  \* Write out the string just uncompressed.  \*/  public void writeString(byte string[]) {  try {  uncompData.write(string);  }  catch (IOException e) {  throw new ExceptionConverter(e);  }  }  /\*\*  \* Add a new string to the string table.  \*/  public void addStringToTable(byte oldString[], byte newString) {  int length = oldString.length;  byte string[] = new byte[length + 1];  System.arraycopy(oldString, 0, string, 0, length);  string[length] = newString;  // Add this new String to the table  stringTable[tableIndex++] = string;  if (tableIndex == 511) {  bitsToGet = 10;  } else if (tableIndex == 1023) {  bitsToGet = 11;  } else if (tableIndex == 2047) {  bitsToGet = 12;  }  }  /\*\*  \* Add a new string to the string table.  \*/  public void addStringToTable(byte string[]) {  // Add this new String to the table  stringTable[tableIndex++] = string;  if (tableIndex == 511) {  bitsToGet = 10;  } else if (tableIndex == 1023) {  bitsToGet = 11;  } else if (tableIndex == 2047) {  bitsToGet = 12;  }  }  /\*\*  \* Append <code>newString</code> to the end of <code>oldString</code>.  \*/  public byte[] composeString(byte oldString[], byte newString) {  int length = oldString.length;  byte string[] = new byte[length + 1];  System.arraycopy(oldString, 0, string, 0, length);  string[length] = newString;  return string;  }  // Returns the next 9, 10, 11 or 12 bits  public int getNextCode() {  // Attempt to get the next code. The exception is caught to make  // this robust to cases wherein the EndOfInformation code has been  // omitted from a strip. Examples of such cases have been observed  // in practice.  try {  nextData = (nextData << 8) | (data[bytePointer++] & 0xff);  nextBits += 8;  if (nextBits < bitsToGet) {  nextData = (nextData << 8) | (data[bytePointer++] & 0xff);  nextBits += 8;  }  int code =  (nextData >> (nextBits - bitsToGet)) & andTable[bitsToGet-9];  nextBits -= bitsToGet;  return code;  } catch(ArrayIndexOutOfBoundsException e) {  // Strip not terminated as expected: return EndOfInformation code.  return 257;  }  }  } |
|  |
| ***FlateDecode***  该方法是基于 zlib/deflate压缩方法，是一种可变长度Lopel-ZIV自适应压缩方法，与自适应霍夫曼编码级联，被定义在Internet RFCS 1950、ZLIB压缩数据格式规范和1951，DEFLATE压缩数据格式规范中。  该压缩方式可以压缩二进制数据和ASCII文本，最终生成的数据是二进制的。  算法如下（itext）:  private static class Filter\_FLATEDECODE implements FilterHandler{  public byte[] decode(byte[] b, PdfName filterName, PdfObject decodeParams, PdfDictionary streamDictionary) throws IOException {  b = PdfReader.FlateDecode(b);  b = PdfReader.decodePredictor(b, decodeParams);  return b;  }  }  /\*\* Decodes a stream that has the FlateDecode filter.  \* @param in the input data  \* @return the decoded data  \*/  public static byte[] FlateDecode(final byte in[]) {  byte b[] = FlateDecode(in, true);  if (b == null)  return FlateDecode(in, false);  return b;  }  /\*\* A helper to FlateDecode.  \* @param in the input data  \* @param strict <CODE>true</CODE> to read a correct stream. <CODE>false</CODE>  \* to try to read a corrupted stream  \* @return the decoded data  \*/  public static byte[] FlateDecode(final byte in[], final boolean strict) {  ByteArrayInputStream stream = new ByteArrayInputStream(in);  InflaterInputStream zip = new InflaterInputStream(stream);  ByteArrayOutputStream out = new ByteArrayOutputStream();  byte b[] = new byte[strict ? 4092 : 1];  try {  int n;  while ((n = zip.read(b)) >= 0) {  out.write(b, 0, n);  }  zip.close();  out.close();  return out.toByteArray();  }  catch (Exception e) {  if (strict)  return null;  return out.toByteArray();  }  finally {  try {  zip.close();  } catch (IOException ex) {  }  try {  out.close();  } catch (IOException ex) {  }  }  } |
|  |
| RunLengthDecode  行程长度压缩法即根据字符串的连续重复字符进行编码的一种方法.  算法如下（itext）：  /\*  \* Handles RUNLENGTHDECODE filter  \*/  private static class Filter\_RUNLENGTHDECODE implements FilterHandler{  public byte[] decode(byte[] b, PdfName filterName, PdfObject decodeParams, PdfDictionary streamDictionary) throws IOException {  // allocate the output buffer  ByteArrayOutputStream baos = new ByteArrayOutputStream();  byte dupCount = -1;  for(int i = 0; i < b.length; i++){  dupCount = b[i];  if (dupCount == -128) break; // this is implicit end of data  if (dupCount >= 0 && dupCount <= 127){  int bytesToCopy = dupCount+1;  baos.write(b, i, bytesToCopy);  i+=bytesToCopy;  } else {  // make dupcount copies of the next byte  i++;  for(int j = 0; j < 1-(int)(dupCount);j++){  baos.write(b[i]);  }  }  }  return baos.toByteArray();  }  } |
|  |
| ***CCITTFaxDecode***  该压缩方式主要用于对图片数据进行压缩，采用group3或group4。CCITT压缩是为了对单色图（每个像素点用1bit表示）进行有效压缩而设计的。  算法如下（itext）：  /\*\*  \* Class that can decompress TIFF files.  \* @since 5.0.3  \*/  public class TIFFFaxDecompressor {  /\*\*  \* The logical order of bits within a byte.  \* <pre>  \* 1 = MSB-to-LSB  \* 2 = LSB-to-MSB (flipped)  \* </pre>  \*/  protected int fillOrder;  protected int compression;  private int t4Options;  private int t6Options;  public int fails;  // Variables set by T4Options  /\*\*  \* Uncompressed mode flag: 1 if uncompressed, 0 if not.  \*/  protected int uncompressedMode = 0;  /\*\*  \* EOL padding flag: 1 if fill bits have been added before an EOL such  \* that the EOL ends on a byte boundary, 0 otherwise.  \*/  protected int fillBits = 0;  /\*\*  \* Coding dimensionality: 1 for 2-dimensional, 0 for 1-dimensional.  \*/  protected int oneD;  private byte[] data;  private int bitPointer, bytePointer;  // Output image buffer  private byte[] buffer;  private int w, h, bitsPerScanline;  private int lineBitNum;  // Data structures needed to store changing elements for the previous  // and the current scanline  private int changingElemSize = 0;  private int prevChangingElems[];  private int currChangingElems[];  // Element at which to start search in getNextChangingElement  private int lastChangingElement = 0;  static int table1[] = {  0x00, // 0 bits are left in first byte - SHOULD NOT HAPPEN  0x01, // 1 bits are left in first byte  0x03, // 2 bits are left in first byte  0x07, // 3 bits are left in first byte  0x0f, // 4 bits are left in first byte  0x1f, // 5 bits are left in first byte  0x3f, // 6 bits are left in first byte  0x7f, // 7 bits are left in first byte  0xff // 8 bits are left in first byte  };  static int table2[] = {  0x00, // 0  0x80, // 1  0xc0, // 2  0xe0, // 3  0xf0, // 4  0xf8, // 5  0xfc, // 6  0xfe, // 7  0xff // 8  };  // Table to be used when fillOrder = 2, for flipping bytes.  static byte flipTable[] = {  0, -128, 64, -64, 32, -96, 96, -32,  16, -112, 80, -48, 48, -80, 112, -16,  8, -120, 72, -56, 40, -88, 104, -24,  24, -104, 88, -40, 56, -72, 120, -8,  4, -124, 68, -60, 36, -92, 100, -28,  20, -108, 84, -44, 52, -76, 116, -12,  12, -116, 76, -52, 44, -84, 108, -20,  28, -100, 92, -36, 60, -68, 124, -4,  2, -126, 66, -62, 34, -94, 98, -30,  18, -110, 82, -46, 50, -78, 114, -14,  10, -118, 74, -54, 42, -86, 106, -22,  26, -102, 90, -38, 58, -70, 122, -6,  6, -122, 70, -58, 38, -90, 102, -26,  22, -106, 86, -42, 54, -74, 118, -10,  14, -114, 78, -50, 46, -82, 110, -18,  30, -98, 94, -34, 62, -66, 126, -2,  1, -127, 65, -63, 33, -95, 97, -31,  17, -111, 81, -47, 49, -79, 113, -15,  9, -119, 73, -55, 41, -87, 105, -23,  25, -103, 89, -39, 57, -71, 121, -7,  5, -123, 69, -59, 37, -91, 101, -27,  21, -107, 85, -43, 53, -75, 117, -11,  13, -115, 77, -51, 45, -83, 109, -19,  29, -99, 93, -35, 61, -67, 125, -3,  3, -125, 67, -61, 35, -93, 99, -29,  19, -109, 83, -45, 51, -77, 115, -13,  11, -117, 75, -53, 43, -85, 107, -21,  27, -101, 91, -37, 59, -69, 123, -5,  7, -121, 71, -57, 39, -89, 103, -25,  23, -105, 87, -41, 55, -73, 119, -9,  15, -113, 79, -49, 47, -81, 111, -17,  31, -97, 95, -33, 63, -65, 127, -1,};  // The main 10 bit white runs lookup table  static short white[] = {  // 0 - 7  6430, 6400, 6400, 6400, 3225, 3225, 3225, 3225,  // 8 - 15  944, 944, 944, 944, 976, 976, 976, 976,  // 16 - 23  1456, 1456, 1456, 1456, 1488, 1488, 1488, 1488,  // 24 - 31  718, 718, 718, 718, 718, 718, 718, 718,  // 32 - 39  750, 750, 750, 750, 750, 750, 750, 750,  // 40 - 47  1520, 1520, 1520, 1520, 1552, 1552, 1552, 1552,  // 48 - 55  428, 428, 428, 428, 428, 428, 428, 428,  // 56 - 63  428, 428, 428, 428, 428, 428, 428, 428,  // 64 - 71  654, 654, 654, 654, 654, 654, 654, 654,  // 72 - 79  1072, 1072, 1072, 1072, 1104, 1104, 1104, 1104,  // 80 - 87  1136, 1136, 1136, 1136, 1168, 1168, 1168, 1168,  // 88 - 95  1200, 1200, 1200, 1200, 1232, 1232, 1232, 1232,  // 96 - 103  622, 622, 622, 622, 622, 622, 622, 622,  // 104 - 111  1008, 1008, 1008, 1008, 1040, 1040, 1040, 1040,  // 112 - 119  44, 44, 44, 44, 44, 44, 44, 44,  // 120 - 127  44, 44, 44, 44, 44, 44, 44, 44,  // 128 - 135  396, 396, 396, 396, 396, 396, 396, 396,  // 136 - 143  396, 396, 396, 396, 396, 396, 396, 396,  // 144 - 151  1712, 1712, 1712, 1712, 1744, 1744, 1744, 1744,  // 152 - 159  846, 846, 846, 846, 846, 846, 846, 846,  // 160 - 167  1264, 1264, 1264, 1264, 1296, 1296, 1296, 1296,  // 168 - 175  1328, 1328, 1328, 1328, 1360, 1360, 1360, 1360,  // 176 - 183  1392, 1392, 1392, 1392, 1424, 1424, 1424, 1424,  // 184 - 191  686, 686, 686, 686, 686, 686, 686, 686,  // 192 - 199  910, 910, 910, 910, 910, 910, 910, 910,  // 200 - 207  1968, 1968, 1968, 1968, 2000, 2000, 2000, 2000,  // 208 - 215  2032, 2032, 2032, 2032, 16, 16, 16, 16,  // 216 - 223  10257, 10257, 10257, 10257, 12305, 12305, 12305, 12305,  // 224 - 231  330, 330, 330, 330, 330, 330, 330, 330,  // 232 - 239  330, 330, 330, 330, 330, 330, 330, 330,  // 240 - 247  330, 330, 330, 330, 330, 330, 330, 330,  // 248 - 255  330, 330, 330, 330, 330, 330, 330, 330,  // 256 - 263  362, 362, 362, 362, 362, 362, 362, 362,  // 264 - 271  362, 362, 362, 362, 362, 362, 362, 362,  // 272 - 279  362, 362, 362, 362, 362, 362, 362, 362,  // 280 - 287  362, 362, 362, 362, 362, 362, 362, 362,  // 288 - 295  878, 878, 878, 878, 878, 878, 878, 878,  // 296 - 303  1904, 1904, 1904, 1904, 1936, 1936, 1936, 1936,  // 304 - 311  -18413, -18413, -16365, -16365, -14317, -14317, -10221, -10221,  // 312 - 319  590, 590, 590, 590, 590, 590, 590, 590,  // 320 - 327  782, 782, 782, 782, 782, 782, 782, 782,  // 328 - 335  1584, 1584, 1584, 1584, 1616, 1616, 1616, 1616,  // 336 - 343  1648, 1648, 1648, 1648, 1680, 1680, 1680, 1680,  // 344 - 351  814, 814, 814, 814, 814, 814, 814, 814,  // 352 - 359  1776, 1776, 1776, 1776, 1808, 1808, 1808, 1808,  // 360 - 367  1840, 1840, 1840, 1840, 1872, 1872, 1872, 1872,  // 368 - 375  6157, 6157, 6157, 6157, 6157, 6157, 6157, 6157,  // 376 - 383  6157, 6157, 6157, 6157, 6157, 6157, 6157, 6157,  // 384 - 391  -12275, -12275, -12275, -12275, -12275, -12275, -12275, -12275,  // 392 - 399  -12275, -12275, -12275, -12275, -12275, -12275, -12275, -12275,  // 400 - 407  14353, 14353, 14353, 14353, 16401, 16401, 16401, 16401,  // 408 - 415  22547, 22547, 24595, 24595, 20497, 20497, 20497, 20497,  // 416 - 423  18449, 18449, 18449, 18449, 26643, 26643, 28691, 28691,  // 424 - 431  30739, 30739, -32749, -32749, -30701, -30701, -28653, -28653,  // 432 - 439  -26605, -26605, -24557, -24557, -22509, -22509, -20461, -20461,  // 440 - 447  8207, 8207, 8207, 8207, 8207, 8207, 8207, 8207,  // 448 - 455  72, 72, 72, 72, 72, 72, 72, 72,  // 456 - 463  72, 72, 72, 72, 72, 72, 72, 72,  // 464 - 471  72, 72, 72, 72, 72, 72, 72, 72,  // 472 - 479  72, 72, 72, 72, 72, 72, 72, 72,  // 480 - 487  72, 72, 72, 72, 72, 72, 72, 72,  // 488 - 495  72, 72, 72, 72, 72, 72, 72, 72,  // 496 - 503  72, 72, 72, 72, 72, 72, 72, 72,  // 504 - 511  72, 72, 72, 72, 72, 72, 72, 72,  // 512 - 519  104, 104, 104, 104, 104, 104, 104, 104,  // 520 - 527  104, 104, 104, 104, 104, 104, 104, 104,  // 528 - 535  104, 104, 104, 104, 104, 104, 104, 104,  // 536 - 543  104, 104, 104, 104, 104, 104, 104, 104,  // 544 - 551  104, 104, 104, 104, 104, 104, 104, 104,  // 552 - 559  104, 104, 104, 104, 104, 104, 104, 104,  // 560 - 567  104, 104, 104, 104, 104, 104, 104, 104,  // 568 - 575  104, 104, 104, 104, 104, 104, 104, 104,  // 576 - 583  4107, 4107, 4107, 4107, 4107, 4107, 4107, 4107,  // 584 - 591  4107, 4107, 4107, 4107, 4107, 4107, 4107, 4107,  // 592 - 599  4107, 4107, 4107, 4107, 4107, 4107, 4107, 4107,  // 600 - 607  4107, 4107, 4107, 4107, 4107, 4107, 4107, 4107,  // 608 - 615  266, 266, 266, 266, 266, 266, 266, 266,  // 616 - 623  266, 266, 266, 266, 266, 266, 266, 266,  // 624 - 631  266, 266, 266, 266, 266, 266, 266, 266,  // 632 - 639  266, 266, 266, 266, 266, 266, 266, 266,  // 640 - 647  298, 298, 298, 298, 298, 298, 298, 298,  // 648 - 655  298, 298, 298, 298, 298, 298, 298, 298,  // 656 - 663  298, 298, 298, 298, 298, 298, 298, 298,  // 664 - 671  298, 298, 298, 298, 298, 298, 298, 298,  // 672 - 679  524, 524, 524, 524, 524, 524, 524, 524,  // 680 - 687  524, 524, 524, 524, 524, 524, 524, 524,  // 688 - 695  556, 556, 556, 556, 556, 556, 556, 556,  // 696 - 703  556, 556, 556, 556, 556, 556, 556, 556,  // 704 - 711  136, 136, 136, 136, 136, 136, 136, 136,  // 712 - 719  136, 136, 136, 136, 136, 136, 136, 136,  // 720 - 727  136, 136, 136, 136, 136, 136, 136, 136,  // 728 - 735  136, 136, 136, 136, 136, 136, 136, 136,  // 736 - 743  136, 136, 136, 136, 136, 136, 136, 136,  // 744 - 751  136, 136, 136, 136, 136, 136, 136, 136,  // 752 - 759  136, 136, 136, 136, 136, 136, 136, 136,  // 760 - 767  136, 136, 136, 136, 136, 136, 136, 136,  // 768 - 775  168, 168, 168, 168, 168, 168, 168, 168,  // 776 - 783  168, 168, 168, 168, 168, 168, 168, 168,  // 784 - 791  168, 168, 168, 168, 168, 168, 168, 168,  // 792 - 799  168, 168, 168, 168, 168, 168, 168, 168,  // 800 - 807  168, 168, 168, 168, 168, 168, 168, 168,  // 808 - 815  168, 168, 168, 168, 168, 168, 168, 168,  // 816 - 823  168, 168, 168, 168, 168, 168, 168, 168,  // 824 - 831  168, 168, 168, 168, 168, 168, 168, 168,  // 832 - 839  460, 460, 460, 460, 460, 460, 460, 460,  // 840 - 847  460, 460, 460, 460, 460, 460, 460, 460,  // 848 - 855  492, 492, 492, 492, 492, 492, 492, 492,  // 856 - 863  492, 492, 492, 492, 492, 492, 492, 492,  // 864 - 871  2059, 2059, 2059, 2059, 2059, 2059, 2059, 2059,  // 872 - 879  2059, 2059, 2059, 2059, 2059, 2059, 2059, 2059,  // 880 - 887  2059, 2059, 2059, 2059, 2059, 2059, 2059, 2059,  // 888 - 895  2059, 2059, 2059, 2059, 2059, 2059, 2059, 2059,  // 896 - 903  200, 200, 200, 200, 200, 200, 200, 200,  // 904 - 911  200, 200, 200, 200, 200, 200, 200, 200,  // 912 - 919  200, 200, 200, 200, 200, 200, 200, 200,  // 920 - 927  200, 200, 200, 200, 200, 200, 200, 200,  // 928 - 935  200, 200, 200, 200, 200, 200, 200, 200,  // 936 - 943  200, 200, 200, 200, 200, 200, 200, 200,  // 944 - 951  200, 200, 200, 200, 200, 200, 200, 200,  // 952 - 959  200, 200, 200, 200, 200, 200, 200, 200,  // 960 - 967  232, 232, 232, 232, 232, 232, 232, 232,  // 968 - 975  232, 232, 232, 232, 232, 232, 232, 232,  // 976 - 983  232, 232, 232, 232, 232, 232, 232, 232,  // 984 - 991  232, 232, 232, 232, 232, 232, 232, 232,  // 992 - 999  232, 232, 232, 232, 232, 232, 232, 232,  // 1000 - 1007  232, 232, 232, 232, 232, 232, 232, 232,  // 1008 - 1015  232, 232, 232, 232, 232, 232, 232, 232,  // 1016 - 1023  232, 232, 232, 232, 232, 232, 232, 232,};  // Additional make up codes for both White and Black runs  static short additionalMakeup[] = {  28679, 28679, 31752, (short) 32777,  (short) 33801, (short) 34825, (short) 35849, (short) 36873,  (short) 29703, (short) 29703, (short) 30727, (short) 30727,  (short) 37897, (short) 38921, (short) 39945, (short) 40969  };  // Initial black run look up table, uses the first 4 bits of a code  static short initBlack[] = {  // 0 - 7  3226, 6412, 200, 168, 38, 38, 134, 134,  // 8 - 15  100, 100, 100, 100, 68, 68, 68, 68  };  //  static short twoBitBlack[] = {292, 260, 226, 226}; // 0 - 3  // Main black run table, using the last 9 bits of possible 13 bit code  static short black[] = {  // 0 - 7  62, 62, 30, 30, 0, 0, 0, 0,  // 8 - 15  0, 0, 0, 0, 0, 0, 0, 0,  // 16 - 23  0, 0, 0, 0, 0, 0, 0, 0,  // 24 - 31  0, 0, 0, 0, 0, 0, 0, 0,  // 32 - 39  3225, 3225, 3225, 3225, 3225, 3225, 3225, 3225,  // 40 - 47  3225, 3225, 3225, 3225, 3225, 3225, 3225, 3225,  // 48 - 55  3225, 3225, 3225, 3225, 3225, 3225, 3225, 3225,  // 56 - 63  3225, 3225, 3225, 3225, 3225, 3225, 3225, 3225,  // 64 - 71  588, 588, 588, 588, 588, 588, 588, 588,  // 72 - 79  1680, 1680, 20499, 22547, 24595, 26643, 1776, 1776,  // 80 - 87  1808, 1808, -24557, -22509, -20461, -18413, 1904, 1904,  // 88 - 95  1936, 1936, -16365, -14317, 782, 782, 782, 782,  // 96 - 103  814, 814, 814, 814, -12269, -10221, 10257, 10257,  // 104 - 111  12305, 12305, 14353, 14353, 16403, 18451, 1712, 1712,  // 112 - 119  1744, 1744, 28691, 30739, -32749, -30701, -28653, -26605,  // 120 - 127  2061, 2061, 2061, 2061, 2061, 2061, 2061, 2061,  // 128 - 135  424, 424, 424, 424, 424, 424, 424, 424,  // 136 - 143  424, 424, 424, 424, 424, 424, 424, 424,  // 144 - 151  424, 424, 424, 424, 424, 424, 424, 424,  // 152 - 159  424, 424, 424, 424, 424, 424, 424, 424,  // 160 - 167  750, 750, 750, 750, 1616, 1616, 1648, 1648,  // 168 - 175  1424, 1424, 1456, 1456, 1488, 1488, 1520, 1520,  // 176 - 183  1840, 1840, 1872, 1872, 1968, 1968, 8209, 8209,  // 184 - 191  524, 524, 524, 524, 524, 524, 524, 524,  // 192 - 199  556, 556, 556, 556, 556, 556, 556, 556,  // 200 - 207  1552, 1552, 1584, 1584, 2000, 2000, 2032, 2032,  // 208 - 215  976, 976, 1008, 1008, 1040, 1040, 1072, 1072,  // 216 - 223  1296, 1296, 1328, 1328, 718, 718, 718, 718,  // 224 - 231  456, 456, 456, 456, 456, 456, 456, 456,  // 232 - 239  456, 456, 456, 456, 456, 456, 456, 456,  // 240 - 247  456, 456, 456, 456, 456, 456, 456, 456,  // 248 - 255  456, 456, 456, 456, 456, 456, 456, 456,  // 256 - 263  326, 326, 326, 326, 326, 326, 326, 326,  // 264 - 271  326, 326, 326, 326, 326, 326, 326, 326,  // 272 - 279  326, 326, 326, 326, 326, 326, 326, 326,  // 280 - 287  326, 326, 326, 326, 326, 326, 326, 326,  // 288 - 295  326, 326, 326, 326, 326, 326, 326, 326,  // 296 - 303  326, 326, 326, 326, 326, 326, 326, 326,  // 304 - 311  326, 326, 326, 326, 326, 326, 326, 326,  // 312 - 319  326, 326, 326, 326, 326, 326, 326, 326,  // 320 - 327  358, 358, 358, 358, 358, 358, 358, 358,  // 328 - 335  358, 358, 358, 358, 358, 358, 358, 358,  // 336 - 343  358, 358, 358, 358, 358, 358, 358, 358,  // 344 - 351  358, 358, 358, 358, 358, 358, 358, 358,  // 352 - 359  358, 358, 358, 358, 358, 358, 358, 358,  // 360 - 367  358, 358, 358, 358, 358, 358, 358, 358,  // 368 - 375  358, 358, 358, 358, 358, 358, 358, 358,  // 376 - 383  358, 358, 358, 358, 358, 358, 358, 358,  // 384 - 391  490, 490, 490, 490, 490, 490, 490, 490,  // 392 - 399  490, 490, 490, 490, 490, 490, 490, 490,  // 400 - 407  4113, 4113, 6161, 6161, 848, 848, 880, 880,  // 408 - 415  912, 912, 944, 944, 622, 622, 622, 622,  // 416 - 423  654, 654, 654, 654, 1104, 1104, 1136, 1136,  // 424 - 431  1168, 1168, 1200, 1200, 1232, 1232, 1264, 1264,  // 432 - 439  686, 686, 686, 686, 1360, 1360, 1392, 1392,  // 440 - 447  12, 12, 12, 12, 12, 12, 12, 12,  // 448 - 455  390, 390, 390, 390, 390, 390, 390, 390,  // 456 - 463  390, 390, 390, 390, 390, 390, 390, 390,  // 464 - 471  390, 390, 390, 390, 390, 390, 390, 390,  // 472 - 479  390, 390, 390, 390, 390, 390, 390, 390,  // 480 - 487  390, 390, 390, 390, 390, 390, 390, 390,  // 488 - 495  390, 390, 390, 390, 390, 390, 390, 390,  // 496 - 503  390, 390, 390, 390, 390, 390, 390, 390,  // 504 - 511  390, 390, 390, 390, 390, 390, 390, 390,};  static byte twoDCodes[] = {  // 0 - 7  80, 88, 23, 71, 30, 30, 62, 62,  // 8 - 15  4, 4, 4, 4, 4, 4, 4, 4,  // 16 - 23  11, 11, 11, 11, 11, 11, 11, 11,  // 24 - 31  11, 11, 11, 11, 11, 11, 11, 11,  // 32 - 39  35, 35, 35, 35, 35, 35, 35, 35,  // 40 - 47  35, 35, 35, 35, 35, 35, 35, 35,  // 48 - 55  51, 51, 51, 51, 51, 51, 51, 51,  // 56 - 63  51, 51, 51, 51, 51, 51, 51, 51,  // 64 - 71  41, 41, 41, 41, 41, 41, 41, 41,  // 72 - 79  41, 41, 41, 41, 41, 41, 41, 41,  // 80 - 87  41, 41, 41, 41, 41, 41, 41, 41,  // 88 - 95  41, 41, 41, 41, 41, 41, 41, 41,  // 96 - 103  41, 41, 41, 41, 41, 41, 41, 41,  // 104 - 111  41, 41, 41, 41, 41, 41, 41, 41,  // 112 - 119  41, 41, 41, 41, 41, 41, 41, 41,  // 120 - 127  41, 41, 41, 41, 41, 41, 41, 41,};  public TIFFFaxDecompressor() {  }  /\*\*  \* Invokes the superclass method and then sets instance variables on  \* the basis of the metadata set on this decompressor.  \*/  public void SetOptions(int fillOrder, int compression, int t4Options, int t6Options) {  this.fillOrder = fillOrder;  this.compression = compression;  this.t4Options = t4Options;  this.t6Options = t6Options;  this.oneD = (int) (t4Options & 0x01);  this.uncompressedMode = (int) ((t4Options & 0x02) >> 1);  this.fillBits = (int) ((t4Options & 0x04) >> 2);  }  public void decodeRaw(byte[] buffer, byte[] compData, int w, int h) {  this.buffer = buffer;  this.data = compData;  this.w = w;  this.h = h;  this.bitsPerScanline = w;  this.lineBitNum = 0;  this.bitPointer = 0;  this.bytePointer = 0;  this.prevChangingElems = new int[w + 1];  this.currChangingElems = new int[w + 1];  fails = 0;  try {  if (compression == TIFFConstants.COMPRESSION\_CCITTRLE) {  decodeRLE();  } else if (compression == TIFFConstants.COMPRESSION\_CCITTFAX3) {  decodeT4();  } else if (compression == TIFFConstants.COMPRESSION\_CCITTFAX4) {  this.uncompressedMode = (int) ((t6Options & 0x02) >> 1);  decodeT6();  } else {  throw new RuntimeException("Unknown compression type " + compression);  }  } catch (ArrayIndexOutOfBoundsException e) {  //ignore  }  }  public void decodeRLE() {  for (int i = 0; i < h; i++) {  // Decode the line.  decodeNextScanline();  // Advance to the next byte boundary if not already there.  if (bitPointer != 0) {  bytePointer++;  bitPointer = 0;  }  // Update the total number of bits.  lineBitNum += bitsPerScanline;  }  }  public void decodeNextScanline() {  int bits = 0, code = 0, isT = 0;  int current, entry, twoBits;  boolean isWhite = true;  int bitOffset = 0;  // Initialize starting of the changing elements array  changingElemSize = 0;  // While scanline not complete  while (bitOffset < w) {  // Mark start of white run.  int runOffset = bitOffset;  while (isWhite && bitOffset < w) {  // White run  current = nextNBits(10);  entry = white[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x0f;  if (bits == 12) { // Additional Make up code  // Get the next 2 bits  twoBits = nextLesserThan8Bits(2);  // Consolidate the 2 new bits and last 2 bits into 4 bits  current = ((current << 2) & 0x000c) | twoBits;  entry = additionalMakeup[current];  bits = (entry >>> 1) & 0x07; // 3 bits 0000 0111  code = (entry >>> 4) & 0x0fff; // 12 bits  bitOffset += code; // Skip white run  updatePointer(4 - bits);  } else if (bits == 0) { // ERROR  ++fails;  // XXX return?  } else if (bits == 15) { // EOL  //  // Instead of throwing an exception, assume that the  // EOL was premature; emit a warning and return.  //  ++fails;  return;  } else {  // 11 bits - 0000 0111 1111 1111 = 0x07ff  code = (entry >>> 5) & 0x07ff;  bitOffset += code;  updatePointer(10 - bits);  if (isT == 0) {  isWhite = false;  currChangingElems[changingElemSize++] = bitOffset;  }  }  }  // Check whether this run completed one width  if (bitOffset == w) {  // If the white run has not been terminated then ensure that  // the next code word is a terminating code for a white run  // of length zero.  int runLength = bitOffset - runOffset;  if (isWhite  && runLength != 0 && runLength % 64 == 0  && nextNBits(8) != 0x35) {  ++fails;  updatePointer(8);  }  break;  }  // Mark start of black run.  runOffset = bitOffset;  while (isWhite == false && bitOffset < w) {  // Black run  current = nextLesserThan8Bits(4);  entry = initBlack[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x000f;  code = (entry >>> 5) & 0x07ff;  if (code == 100) {  current = nextNBits(9);  entry = black[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x000f;  code = (entry >>> 5) & 0x07ff;  if (bits == 12) {  // Additional makeup codes  updatePointer(5);  current = nextLesserThan8Bits(4);  entry = additionalMakeup[current];  bits = (entry >>> 1) & 0x07; // 3 bits 0000 0111  code = (entry >>> 4) & 0x0fff; // 12 bits  setToBlack(bitOffset, code);  bitOffset += code;  updatePointer(4 - bits);  } else if (bits == 15) {  //  // Instead of throwing an exception, assume that the  // EOL was premature; emit a warning and return.  //  ++fails;  return;  } else {  setToBlack(bitOffset, code);  bitOffset += code;  updatePointer(9 - bits);  if (isT == 0) {  isWhite = true;  currChangingElems[changingElemSize++] = bitOffset;  }  }  } else if (code == 200) {  // Is a Terminating code  current = nextLesserThan8Bits(2);  entry = twoBitBlack[current];  code = (entry >>> 5) & 0x07ff;  bits = (entry >>> 1) & 0x0f;  setToBlack(bitOffset, code);  bitOffset += code;  updatePointer(2 - bits);  isWhite = true;  currChangingElems[changingElemSize++] = bitOffset;  } else {  // Is a Terminating code  setToBlack(bitOffset, code);  bitOffset += code;  updatePointer(4 - bits);  isWhite = true;  currChangingElems[changingElemSize++] = bitOffset;  }  }  // Check whether this run completed one width  if (bitOffset == w) {  // If the black run has not been terminated then ensure that  // the next code word is a terminating code for a black run  // of length zero.  int runLength = bitOffset - runOffset;  if (!isWhite  && runLength != 0 && runLength % 64 == 0  && nextNBits(10) != 0x37) {  ++fails;  updatePointer(10);  }  break;  }  }  currChangingElems[changingElemSize++] = bitOffset;  }  public void decodeT4() {  int height = h;  int a0, a1, b1, b2;  int[] b = new int[2];  int entry, code, bits, color;  boolean isWhite;  int currIndex = 0;  int temp[];  if (data.length < 2) {  throw new RuntimeException("Insufficient data to read initial EOL.");  }  // The data should start with an EOL code  int next12 = nextNBits(12);  if (next12 != 1) {  ++fails;  }  updatePointer(12);  // Find the first one-dimensionally encoded line.  int modeFlag = 0;  int lines = -1; // indicates imaginary line before first actual line.  while (modeFlag != 1) {  try {  modeFlag = findNextLine();  lines++; // Normally 'lines' will be 0 on exiting loop.  } catch (Exception eofe) {  throw new RuntimeException("No reference line present.");  }  }  int bitOffset;  // Then the 1D encoded scanline data will occur, changing elements  // array gets set.  decodeNextScanline();  lines++;  lineBitNum += bitsPerScanline;  while (lines < height) {  // Every line must begin with an EOL followed by a bit which  // indicates whether the following scanline is 1D or 2D encoded.  try {  modeFlag = findNextLine();  } catch (Exception eofe) {  ++fails;  break;  }  if (modeFlag == 0) {  // 2D encoded scanline follows  // Initialize previous scanlines changing elements, and  // initialize current scanline's changing elements array  temp = prevChangingElems;  prevChangingElems = currChangingElems;  currChangingElems = temp;  currIndex = 0;  // a0 has to be set just before the start of this scanline.  a0 = -1;  isWhite = true;  bitOffset = 0;  lastChangingElement = 0;  while (bitOffset < w) {  // Get the next changing element  getNextChangingElement(a0, isWhite, b);  b1 = b[0];  b2 = b[1];  // Get the next seven bits  entry = nextLesserThan8Bits(7);  // Run these through the 2DCodes table  entry = (int) (twoDCodes[entry] & 0xff);  // Get the code and the number of bits used up  code = (entry & 0x78) >>> 3;  bits = entry & 0x07;  if (code == 0) {  if (!isWhite) {  setToBlack(bitOffset, b2 - bitOffset);  }  bitOffset = a0 = b2;  // Set pointer to consume the correct number of bits.  updatePointer(7 - bits);  } else if (code == 1) {  // Horizontal  updatePointer(7 - bits);  // identify the next 2 codes.  int number;  if (isWhite) {  number = decodeWhiteCodeWord();  bitOffset += number;  currChangingElems[currIndex++] = bitOffset;  number = decodeBlackCodeWord();  setToBlack(bitOffset, number);  bitOffset += number;  currChangingElems[currIndex++] = bitOffset;  } else {  number = decodeBlackCodeWord();  setToBlack(bitOffset, number);  bitOffset += number;  currChangingElems[currIndex++] = bitOffset;  number = decodeWhiteCodeWord();  bitOffset += number;  currChangingElems[currIndex++] = bitOffset;  }  a0 = bitOffset;  } else if (code <= 8) {  // Vertical  a1 = b1 + (code - 5);  currChangingElems[currIndex++] = a1;  // We write the current color till a1 - 1 pos,  // since a1 is where the next color starts  if (!isWhite) {  setToBlack(bitOffset, a1 - bitOffset);  }  bitOffset = a0 = a1;  isWhite = !isWhite;  updatePointer(7 - bits);  } else {  ++fails;  // Find the next one-dimensionally encoded line.  int numLinesTested = 0;  while (modeFlag != 1) {  try {  modeFlag = findNextLine();  numLinesTested++;  } catch (Exception eofe) {  return;  }  }  lines += numLinesTested - 1;  updatePointer(13);  break;  }  }  // Add the changing element beyond the current scanline for the  // other color too  currChangingElems[currIndex++] = bitOffset;  changingElemSize = currIndex;  } else { // modeFlag == 1  // 1D encoded scanline follows  decodeNextScanline();  }  lineBitNum += bitsPerScanline;  lines++;  } // while(lines < height)  }  public synchronized void decodeT6() {  int height = h;  int a0, a1, b1, b2;  int entry, code, bits;  boolean isWhite;  int currIndex;  int temp[];  // Return values from getNextChangingElement  int[] b = new int[2];  // uncompressedMode - have written some code for this, but this  // has not been tested due to lack of test images using this optional  // extension. This code is when code == 11. aastha 03/03/1999  // Local cached reference  int[] cce = currChangingElems;  // Assume invisible preceding row of all white pixels and insert  // both black and white changing elements beyond the end of this  // imaginary scanline.  changingElemSize = 0;  cce[changingElemSize++] = w;  cce[changingElemSize++] = w;  int bitOffset;  for (int lines = 0; lines < height; lines++) {  // a0 has to be set just before the start of the scanline.  a0 = -1;  isWhite = true;  // Assign the changing elements of the previous scanline to  // prevChangingElems and start putting this new scanline's  // changing elements into the currChangingElems.  temp = prevChangingElems;  prevChangingElems = currChangingElems;  cce = currChangingElems = temp;  currIndex = 0;  // Start decoding the scanline  bitOffset = 0;  // Reset search start position for getNextChangingElement  lastChangingElement = 0;  // Till one whole scanline is decoded  while (bitOffset < w) {  // Get the next changing element  getNextChangingElement(a0, isWhite, b);  b1 = b[0];  b2 = b[1];  // Get the next seven bits  entry = nextLesserThan8Bits(7);  // Run these through the 2DCodes table  entry = (int) (twoDCodes[entry] & 0xff);  // Get the code and the number of bits used up  code = (entry & 0x78) >>> 3;  bits = entry & 0x07;  if (code == 0) { // Pass  // We always assume WhiteIsZero format for fax.  if (!isWhite) {  if (b2 > w) {  b2 = w;  }  setToBlack(bitOffset, b2 - bitOffset);  }  bitOffset = a0 = b2;  // Set pointer to only consume the correct number of bits.  updatePointer(7 - bits);  } else if (code == 1) { // Horizontal  // Set pointer to only consume the correct number of bits.  updatePointer(7 - bits);  // identify the next 2 alternating color codes.  int number;  if (isWhite) {  // Following are white and black runs  number = decodeWhiteCodeWord();  bitOffset += number;  cce[currIndex++] = bitOffset;  number = decodeBlackCodeWord();  if (number > w - bitOffset) {  number = w - bitOffset;  }  setToBlack(bitOffset, number);  bitOffset += number;  cce[currIndex++] = bitOffset;  } else {  // First a black run and then a white run follows  number = decodeBlackCodeWord();  if (number > w - bitOffset) {  number = w - bitOffset;  }  setToBlack(bitOffset, number);  bitOffset += number;  cce[currIndex++] = bitOffset;  number = decodeWhiteCodeWord();  bitOffset += number;  cce[currIndex++] = bitOffset;  }  a0 = bitOffset;  } else if (code <= 8) { // Vertical  a1 = b1 + (code - 5);  cce[currIndex++] = a1;  // We write the current color till a1 - 1 pos,  // since a1 is where the next color starts  if (!isWhite) {  if (a1 > w) {  a1 = w;  }  setToBlack(bitOffset, a1 - bitOffset);  }  bitOffset = a0 = a1;  isWhite = !isWhite;  updatePointer(7 - bits);  } else if (code == 11) {  int entranceCode = nextLesserThan8Bits(3);  int zeros = 0;  boolean exit = false;  while (!exit) {  while (nextLesserThan8Bits(1) != 1) {  zeros++;  }  if (zeros > 5) {  // Exit code  // Zeros before exit code  zeros = zeros - 6;  if (!isWhite && (zeros > 0)) {  cce[currIndex++] = bitOffset;  }  // Zeros before the exit code  bitOffset += zeros;  if (zeros > 0) {  // Some zeros have been written  isWhite = true;  }  // Read in the bit which specifies the color of  // the following run  if (nextLesserThan8Bits(1) == 0) {  if (!isWhite) {  cce[currIndex++] = bitOffset;  }  isWhite = true;  } else {  if (isWhite) {  cce[currIndex++] = bitOffset;  }  isWhite = false;  }  exit = true;  }  if (zeros == 5) {  if (!isWhite) {  cce[currIndex++] = bitOffset;  }  bitOffset += zeros;  // Last thing written was white  isWhite = true;  } else {  bitOffset += zeros;  cce[currIndex++] = bitOffset;  setToBlack(bitOffset, 1);  ++bitOffset;  // Last thing written was black  isWhite = false;  }  }  }  } // while bitOffset < w  // Add the changing element beyond the current scanline for the  // other color too, if not already added previously  if (currIndex <= w)  cce[currIndex++] = bitOffset;  // Number of changing elements in this scanline.  changingElemSize = currIndex;  lineBitNum += bitsPerScanline;  } // for lines < height  }  private void setToBlack(int bitNum, int numBits) {  // bitNum is relative to current scanline so bump it by lineBitNum  bitNum += lineBitNum;  int lastBit = bitNum + numBits;  int byteNum = bitNum >> 3;  // Handle bits in first byte  int shift = bitNum & 0x7;  if (shift > 0) {  int maskVal = 1 << (7 - shift);  byte val = buffer[byteNum];  while (maskVal > 0 && bitNum < lastBit) {  val |= maskVal;  maskVal >>= 1;  ++bitNum;  }  buffer[byteNum] = val;  }  // Fill in 8 bits at a time  byteNum = bitNum >> 3;  while (bitNum < lastBit - 7) {  buffer[byteNum++] = (byte) 255;  bitNum += 8;  }  // Fill in remaining bits  while (bitNum < lastBit) {  byteNum = bitNum >> 3;  buffer[byteNum] |= 1 << (7 - (bitNum & 0x7));  ++bitNum;  }  }  // Returns run length  private int decodeWhiteCodeWord() {  int current, entry, bits, isT, twoBits, code = -1;  int runLength = 0;  boolean isWhite = true;  while (isWhite) {  current = nextNBits(10);  entry = white[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x0f;  if (bits == 12) { // Additional Make up code  // Get the next 2 bits  twoBits = nextLesserThan8Bits(2);  // Consolidate the 2 new bits and last 2 bits into 4 bits  current = ((current << 2) & 0x000c) | twoBits;  entry = additionalMakeup[current];  bits = (entry >>> 1) & 0x07; // 3 bits 0000 0111  code = (entry >>> 4) & 0x0fff; // 12 bits  runLength += code;  updatePointer(4 - bits);  } else if (bits == 0) { // ERROR  throw new RuntimeException("Error 0");  } else if (bits == 15) { // EOL  throw new RuntimeException("Error 1");  } else {  // 11 bits - 0000 0111 1111 1111 = 0x07ff  code = (entry >>> 5) & 0x07ff;  runLength += code;  updatePointer(10 - bits);  if (isT == 0) {  isWhite = false;  }  }  }  return runLength;  }  // Returns run length  private int decodeBlackCodeWord() {  int current, entry, bits, isT, twoBits, code = -1;  int runLength = 0;  boolean isWhite = false;  while (!isWhite) {  current = nextLesserThan8Bits(4);  entry = initBlack[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x000f;  code = (entry >>> 5) & 0x07ff;  if (code == 100) {  current = nextNBits(9);  entry = black[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x000f;  code = (entry >>> 5) & 0x07ff;  if (bits == 12) {  // Additional makeup codes  updatePointer(5);  current = nextLesserThan8Bits(4);  entry = additionalMakeup[current];  bits = (entry >>> 1) & 0x07; // 3 bits 0000 0111  code = (entry >>> 4) & 0x0fff; // 12 bits  runLength += code;  updatePointer(4 - bits);  } else if (bits == 15) {  // EOL code  throw new RuntimeException("Error 2");  } else {  runLength += code;  updatePointer(9 - bits);  if (isT == 0) {  isWhite = true;  }  }  } else if (code == 200) {  // Is a Terminating code  current = nextLesserThan8Bits(2);  entry = twoBitBlack[current];  code = (entry >>> 5) & 0x07ff;  runLength += code;  bits = (entry >>> 1) & 0x0f;  updatePointer(2 - bits);  isWhite = true;  } else {  // Is a Terminating code  runLength += code;  updatePointer(4 - bits);  isWhite = true;  }  }  return runLength;  }  private int findNextLine() {  // Set maximum and current bit index into the compressed data.  int bitIndexMax = data.length \* 8 - 1;  int bitIndexMax12 = bitIndexMax - 12;  int bitIndex = bytePointer \* 8 + bitPointer;  // Loop while at least 12 bits are available.  while (bitIndex <= bitIndexMax12) {  // Get the next 12 bits.  int next12Bits = nextNBits(12);  bitIndex += 12;  // Loop while the 12 bits are not unity, i.e., while the EOL  // has not been reached, and there is at least one bit left.  while (next12Bits != 1 && bitIndex < bitIndexMax) {  next12Bits =  ((next12Bits & 0x000007ff) << 1)  | (nextLesserThan8Bits(1) & 0x00000001);  bitIndex++;  }  if (next12Bits == 1) { // now positioned just after EOL  if (oneD == 1) { // two-dimensional coding  if (bitIndex < bitIndexMax) {  // check next bit against type of line being sought  return nextLesserThan8Bits(1);  }  } else {  return 1;  }  }  }  // EOL not found.  throw new RuntimeException();  }  private void getNextChangingElement(int a0, boolean isWhite, int[] ret) {  // Local copies of instance variables  int[] pce = this.prevChangingElems;  int ces = this.changingElemSize;  // If the previous match was at an odd element, we still  // have to search the preceeding element.  // int start = lastChangingElement & ~0x1;  int start = lastChangingElement > 0 ? lastChangingElement - 1 : 0;  if (isWhite) {  start &= ~0x1; // Search even numbered elements  } else {  start |= 0x1; // Search odd numbered elements  }  int i = start;  for (; i < ces; i += 2) {  int temp = pce[i];  if (temp > a0) {  lastChangingElement = i;  ret[0] = temp;  break;  }  }  if (i + 1 < ces) {  ret[1] = pce[i + 1];  }  }  private int nextNBits(int bitsToGet) {  byte b, next, next2next;  int l = data.length - 1;  int bp = this.bytePointer;  if (fillOrder == 1) {  b = data[bp];  if (bp == l) {  next = 0x00;  next2next = 0x00;  } else if ((bp + 1) == l) {  next = data[bp + 1];  next2next = 0x00;  } else {  next = data[bp + 1];  next2next = data[bp + 2];  }  } else if (fillOrder == 2) {  b = flipTable[data[bp] & 0xff];  if (bp == l) {  next = 0x00;  next2next = 0x00;  } else if ((bp + 1) == l) {  next = flipTable[data[bp + 1] & 0xff];  next2next = 0x00;  } else {  next = flipTable[data[bp + 1] & 0xff];  next2next = flipTable[data[bp + 2] & 0xff];  }  } else {  throw new RuntimeException("Invalid FillOrder");  }  int bitsLeft = 8 - bitPointer;  int bitsFromNextByte = bitsToGet - bitsLeft;  int bitsFromNext2NextByte = 0;  if (bitsFromNextByte > 8) {  bitsFromNext2NextByte = bitsFromNextByte - 8;  bitsFromNextByte = 8;  }  bytePointer++;  int i1 = (b & table1[bitsLeft]) << (bitsToGet - bitsLeft);  int i2 = (next & table2[bitsFromNextByte]) >>> (8 - bitsFromNextByte);  int i3 = 0;  if (bitsFromNext2NextByte != 0) {  i2 <<= bitsFromNext2NextByte;  i3 = (next2next & table2[bitsFromNext2NextByte])  >>> (8 - bitsFromNext2NextByte);  i2 |= i3;  bytePointer++;  bitPointer = bitsFromNext2NextByte;  } else {  if (bitsFromNextByte == 8) {  bitPointer = 0;  bytePointer++;  } else {  bitPointer = bitsFromNextByte;  }  }  int i = i1 | i2;  return i;  }  private int nextLesserThan8Bits(int bitsToGet) {  byte b, next;  int l = data.length - 1;  int bp = this.bytePointer;  if (fillOrder == 1) {  b = data[bp];  if (bp == l) {  next = 0x00;  } else {  next = data[bp + 1];  }  } else if (fillOrder == 2) {  b = flipTable[data[bp] & 0xff];  if (bp == l) {  next = 0x00;  } else {  next = flipTable[data[bp + 1] & 0xff];  }  } else {  throw new RuntimeException("Invalid FillOrder");  }  int bitsLeft = 8 - bitPointer;  int bitsFromNextByte = bitsToGet - bitsLeft;  int shift = bitsLeft - bitsToGet;  int i1, i2;  if (shift >= 0) {  i1 = (b & table1[bitsLeft]) >>> shift;  bitPointer += bitsToGet;  if (bitPointer == 8) {  bitPointer = 0;  bytePointer++;  }  } else {  i1 = (b & table1[bitsLeft]) << (-shift);  i2 = (next & table2[bitsFromNextByte]) >>> (8 - bitsFromNextByte);  i1 |= i2;  bytePointer++;  bitPointer = bitsFromNextByte;  }  return i1;  }  // Move pointer backwards by given amount of bits  private void updatePointer(int bitsToMoveBack) {  if (bitsToMoveBack > 8) {  bytePointer -= bitsToMoveBack / 8;  bitsToMoveBack %= 8;  }  int i = bitPointer - bitsToMoveBack;  if (i < 0) {  bytePointer--;  bitPointer = 8 + i;  } else {  bitPointer = i;  }  }  } |
|  |
| /\*\*  \* Class that can decode TIFF files.  \*/  public class TIFFFaxDecoder {  private int bitPointer, bytePointer;  private byte[] data;  private int w, h;  private long fillOrder;  // Data structures needed to store changing elements for the previous  // and the current scanline  private int changingElemSize = 0;  private int prevChangingElems[];  private int currChangingElems[];  // Element at which to start search in getNextChangingElement  private int lastChangingElement = 0;  private int compression = 2;  // Variables set by T4Options  private int uncompressedMode = 0;  private int fillBits = 0;  private int oneD;  // should iText try to recover from images it can't read?  private boolean recoverFromImageError;  static int table1[] = {  0x00, // 0 bits are left in first byte - SHOULD NOT HAPPEN  0x01, // 1 bits are left in first byte  0x03, // 2 bits are left in first byte  0x07, // 3 bits are left in first byte  0x0f, // 4 bits are left in first byte  0x1f, // 5 bits are left in first byte  0x3f, // 6 bits are left in first byte  0x7f, // 7 bits are left in first byte  0xff // 8 bits are left in first byte  };  static int table2[] = {  0x00, // 0  0x80, // 1  0xc0, // 2  0xe0, // 3  0xf0, // 4  0xf8, // 5  0xfc, // 6  0xfe, // 7  0xff // 8  };  // Table to be used when fillOrder = 2, for flipping bytes.  static byte flipTable[] = {  0, -128, 64, -64, 32, -96, 96, -32,  16, -112, 80, -48, 48, -80, 112, -16,  8, -120, 72, -56, 40, -88, 104, -24,  24, -104, 88, -40, 56, -72, 120, -8,  4, -124, 68, -60, 36, -92, 100, -28,  20, -108, 84, -44, 52, -76, 116, -12,  12, -116, 76, -52, 44, -84, 108, -20,  28, -100, 92, -36, 60, -68, 124, -4,  2, -126, 66, -62, 34, -94, 98, -30,  18, -110, 82, -46, 50, -78, 114, -14,  10, -118, 74, -54, 42, -86, 106, -22,  26, -102, 90, -38, 58, -70, 122, -6,  6, -122, 70, -58, 38, -90, 102, -26,  22, -106, 86, -42, 54, -74, 118, -10,  14, -114, 78, -50, 46, -82, 110, -18,  30, -98, 94, -34, 62, -66, 126, -2,  1, -127, 65, -63, 33, -95, 97, -31,  17, -111, 81, -47, 49, -79, 113, -15,  9, -119, 73, -55, 41, -87, 105, -23,  25, -103, 89, -39, 57, -71, 121, -7,  5, -123, 69, -59, 37, -91, 101, -27,  21, -107, 85, -43, 53, -75, 117, -11,  13, -115, 77, -51, 45, -83, 109, -19,  29, -99, 93, -35, 61, -67, 125, -3,  3, -125, 67, -61, 35, -93, 99, -29,  19, -109, 83, -45, 51, -77, 115, -13,  11, -117, 75, -53, 43, -85, 107, -21,  27, -101, 91, -37, 59, -69, 123, -5,  7, -121, 71, -57, 39, -89, 103, -25,  23, -105, 87, -41, 55, -73, 119, -9,  15, -113, 79, -49, 47, -81, 111, -17,  31, -97, 95, -33, 63, -65, 127, -1,  };  // The main 10 bit white runs lookup table  static short white[] = {  // 0 - 7  6430, 6400, 6400, 6400, 3225, 3225, 3225, 3225,  // 8 - 15  944, 944, 944, 944, 976, 976, 976, 976,  // 16 - 23  1456, 1456, 1456, 1456, 1488, 1488, 1488, 1488,  // 24 - 31  718, 718, 718, 718, 718, 718, 718, 718,  // 32 - 39  750, 750, 750, 750, 750, 750, 750, 750,  // 40 - 47  1520, 1520, 1520, 1520, 1552, 1552, 1552, 1552,  // 48 - 55  428, 428, 428, 428, 428, 428, 428, 428,  // 56 - 63  428, 428, 428, 428, 428, 428, 428, 428,  // 64 - 71  654, 654, 654, 654, 654, 654, 654, 654,  // 72 - 79  1072, 1072, 1072, 1072, 1104, 1104, 1104, 1104,  // 80 - 87  1136, 1136, 1136, 1136, 1168, 1168, 1168, 1168,  // 88 - 95  1200, 1200, 1200, 1200, 1232, 1232, 1232, 1232,  // 96 - 103  622, 622, 622, 622, 622, 622, 622, 622,  // 104 - 111  1008, 1008, 1008, 1008, 1040, 1040, 1040, 1040,  // 112 - 119  44, 44, 44, 44, 44, 44, 44, 44,  // 120 - 127  44, 44, 44, 44, 44, 44, 44, 44,  // 128 - 135  396, 396, 396, 396, 396, 396, 396, 396,  // 136 - 143  396, 396, 396, 396, 396, 396, 396, 396,  // 144 - 151  1712, 1712, 1712, 1712, 1744, 1744, 1744, 1744,  // 152 - 159  846, 846, 846, 846, 846, 846, 846, 846,  // 160 - 167  1264, 1264, 1264, 1264, 1296, 1296, 1296, 1296,  // 168 - 175  1328, 1328, 1328, 1328, 1360, 1360, 1360, 1360,  // 176 - 183  1392, 1392, 1392, 1392, 1424, 1424, 1424, 1424,  // 184 - 191  686, 686, 686, 686, 686, 686, 686, 686,  // 192 - 199  910, 910, 910, 910, 910, 910, 910, 910,  // 200 - 207  1968, 1968, 1968, 1968, 2000, 2000, 2000, 2000,  // 208 - 215  2032, 2032, 2032, 2032, 16, 16, 16, 16,  // 216 - 223  10257, 10257, 10257, 10257, 12305, 12305, 12305, 12305,  // 224 - 231  330, 330, 330, 330, 330, 330, 330, 330,  // 232 - 239  330, 330, 330, 330, 330, 330, 330, 330,  // 240 - 247  330, 330, 330, 330, 330, 330, 330, 330,  // 248 - 255  330, 330, 330, 330, 330, 330, 330, 330,  // 256 - 263  362, 362, 362, 362, 362, 362, 362, 362,  // 264 - 271  362, 362, 362, 362, 362, 362, 362, 362,  // 272 - 279  362, 362, 362, 362, 362, 362, 362, 362,  // 280 - 287  362, 362, 362, 362, 362, 362, 362, 362,  // 288 - 295  878, 878, 878, 878, 878, 878, 878, 878,  // 296 - 303  1904, 1904, 1904, 1904, 1936, 1936, 1936, 1936,  // 304 - 311  -18413, -18413, -16365, -16365, -14317, -14317, -10221, -10221,  // 312 - 319  590, 590, 590, 590, 590, 590, 590, 590,  // 320 - 327  782, 782, 782, 782, 782, 782, 782, 782,  // 328 - 335  1584, 1584, 1584, 1584, 1616, 1616, 1616, 1616,  // 336 - 343  1648, 1648, 1648, 1648, 1680, 1680, 1680, 1680,  // 344 - 351  814, 814, 814, 814, 814, 814, 814, 814,  // 352 - 359  1776, 1776, 1776, 1776, 1808, 1808, 1808, 1808,  // 360 - 367  1840, 1840, 1840, 1840, 1872, 1872, 1872, 1872,  // 368 - 375  6157, 6157, 6157, 6157, 6157, 6157, 6157, 6157,  // 376 - 383  6157, 6157, 6157, 6157, 6157, 6157, 6157, 6157,  // 384 - 391  -12275, -12275, -12275, -12275, -12275, -12275, -12275, -12275,  // 392 - 399  -12275, -12275, -12275, -12275, -12275, -12275, -12275, -12275,  // 400 - 407  14353, 14353, 14353, 14353, 16401, 16401, 16401, 16401,  // 408 - 415  22547, 22547, 24595, 24595, 20497, 20497, 20497, 20497,  // 416 - 423  18449, 18449, 18449, 18449, 26643, 26643, 28691, 28691,  // 424 - 431  30739, 30739, -32749, -32749, -30701, -30701, -28653, -28653,  // 432 - 439  -26605, -26605, -24557, -24557, -22509, -22509, -20461, -20461,  // 440 - 447  8207, 8207, 8207, 8207, 8207, 8207, 8207, 8207,  // 448 - 455  72, 72, 72, 72, 72, 72, 72, 72,  // 456 - 463  72, 72, 72, 72, 72, 72, 72, 72,  // 464 - 471  72, 72, 72, 72, 72, 72, 72, 72,  // 472 - 479  72, 72, 72, 72, 72, 72, 72, 72,  // 480 - 487  72, 72, 72, 72, 72, 72, 72, 72,  // 488 - 495  72, 72, 72, 72, 72, 72, 72, 72,  // 496 - 503  72, 72, 72, 72, 72, 72, 72, 72,  // 504 - 511  72, 72, 72, 72, 72, 72, 72, 72,  // 512 - 519  104, 104, 104, 104, 104, 104, 104, 104,  // 520 - 527  104, 104, 104, 104, 104, 104, 104, 104,  // 528 - 535  104, 104, 104, 104, 104, 104, 104, 104,  // 536 - 543  104, 104, 104, 104, 104, 104, 104, 104,  // 544 - 551  104, 104, 104, 104, 104, 104, 104, 104,  // 552 - 559  104, 104, 104, 104, 104, 104, 104, 104,  // 560 - 567  104, 104, 104, 104, 104, 104, 104, 104,  // 568 - 575  104, 104, 104, 104, 104, 104, 104, 104,  // 576 - 583  4107, 4107, 4107, 4107, 4107, 4107, 4107, 4107,  // 584 - 591  4107, 4107, 4107, 4107, 4107, 4107, 4107, 4107,  // 592 - 599  4107, 4107, 4107, 4107, 4107, 4107, 4107, 4107,  // 600 - 607  4107, 4107, 4107, 4107, 4107, 4107, 4107, 4107,  // 608 - 615  266, 266, 266, 266, 266, 266, 266, 266,  // 616 - 623  266, 266, 266, 266, 266, 266, 266, 266,  // 624 - 631  266, 266, 266, 266, 266, 266, 266, 266,  // 632 - 639  266, 266, 266, 266, 266, 266, 266, 266,  // 640 - 647  298, 298, 298, 298, 298, 298, 298, 298,  // 648 - 655  298, 298, 298, 298, 298, 298, 298, 298,  // 656 - 663  298, 298, 298, 298, 298, 298, 298, 298,  // 664 - 671  298, 298, 298, 298, 298, 298, 298, 298,  // 672 - 679  524, 524, 524, 524, 524, 524, 524, 524,  // 680 - 687  524, 524, 524, 524, 524, 524, 524, 524,  // 688 - 695  556, 556, 556, 556, 556, 556, 556, 556,  // 696 - 703  556, 556, 556, 556, 556, 556, 556, 556,  // 704 - 711  136, 136, 136, 136, 136, 136, 136, 136,  // 712 - 719  136, 136, 136, 136, 136, 136, 136, 136,  // 720 - 727  136, 136, 136, 136, 136, 136, 136, 136,  // 728 - 735  136, 136, 136, 136, 136, 136, 136, 136,  // 736 - 743  136, 136, 136, 136, 136, 136, 136, 136,  // 744 - 751  136, 136, 136, 136, 136, 136, 136, 136,  // 752 - 759  136, 136, 136, 136, 136, 136, 136, 136,  // 760 - 767  136, 136, 136, 136, 136, 136, 136, 136,  // 768 - 775  168, 168, 168, 168, 168, 168, 168, 168,  // 776 - 783  168, 168, 168, 168, 168, 168, 168, 168,  // 784 - 791  168, 168, 168, 168, 168, 168, 168, 168,  // 792 - 799  168, 168, 168, 168, 168, 168, 168, 168,  // 800 - 807  168, 168, 168, 168, 168, 168, 168, 168,  // 808 - 815  168, 168, 168, 168, 168, 168, 168, 168,  // 816 - 823  168, 168, 168, 168, 168, 168, 168, 168,  // 824 - 831  168, 168, 168, 168, 168, 168, 168, 168,  // 832 - 839  460, 460, 460, 460, 460, 460, 460, 460,  // 840 - 847  460, 460, 460, 460, 460, 460, 460, 460,  // 848 - 855  492, 492, 492, 492, 492, 492, 492, 492,  // 856 - 863  492, 492, 492, 492, 492, 492, 492, 492,  // 864 - 871  2059, 2059, 2059, 2059, 2059, 2059, 2059, 2059,  // 872 - 879  2059, 2059, 2059, 2059, 2059, 2059, 2059, 2059,  // 880 - 887  2059, 2059, 2059, 2059, 2059, 2059, 2059, 2059,  // 888 - 895  2059, 2059, 2059, 2059, 2059, 2059, 2059, 2059,  // 896 - 903  200, 200, 200, 200, 200, 200, 200, 200,  // 904 - 911  200, 200, 200, 200, 200, 200, 200, 200,  // 912 - 919  200, 200, 200, 200, 200, 200, 200, 200,  // 920 - 927  200, 200, 200, 200, 200, 200, 200, 200,  // 928 - 935  200, 200, 200, 200, 200, 200, 200, 200,  // 936 - 943  200, 200, 200, 200, 200, 200, 200, 200,  // 944 - 951  200, 200, 200, 200, 200, 200, 200, 200,  // 952 - 959  200, 200, 200, 200, 200, 200, 200, 200,  // 960 - 967  232, 232, 232, 232, 232, 232, 232, 232,  // 968 - 975  232, 232, 232, 232, 232, 232, 232, 232,  // 976 - 983  232, 232, 232, 232, 232, 232, 232, 232,  // 984 - 991  232, 232, 232, 232, 232, 232, 232, 232,  // 992 - 999  232, 232, 232, 232, 232, 232, 232, 232,  // 1000 - 1007  232, 232, 232, 232, 232, 232, 232, 232,  // 1008 - 1015  232, 232, 232, 232, 232, 232, 232, 232,  // 1016 - 1023  232, 232, 232, 232, 232, 232, 232, 232,  };  // Additional make up codes for both White and Black runs  static short additionalMakeup[] = {  28679, 28679, 31752, (short)32777,  (short)33801, (short)34825, (short)35849, (short)36873,  (short)29703, (short)29703, (short)30727, (short)30727,  (short)37897, (short)38921, (short)39945, (short)40969  };  // Initial black run look up table, uses the first 4 bits of a code  static short initBlack[] = {  // 0 - 7  3226, 6412, 200, 168, 38, 38, 134, 134,  // 8 - 15  100, 100, 100, 100, 68, 68, 68, 68  };  //  static short twoBitBlack[] = {292, 260, 226, 226}; // 0 - 3  // Main black run table, using the last 9 bits of possible 13 bit code  static short black[] = {  // 0 - 7  62, 62, 30, 30, 0, 0, 0, 0,  // 8 - 15  0, 0, 0, 0, 0, 0, 0, 0,  // 16 - 23  0, 0, 0, 0, 0, 0, 0, 0,  // 24 - 31  0, 0, 0, 0, 0, 0, 0, 0,  // 32 - 39  3225, 3225, 3225, 3225, 3225, 3225, 3225, 3225,  // 40 - 47  3225, 3225, 3225, 3225, 3225, 3225, 3225, 3225,  // 48 - 55  3225, 3225, 3225, 3225, 3225, 3225, 3225, 3225,  // 56 - 63  3225, 3225, 3225, 3225, 3225, 3225, 3225, 3225,  // 64 - 71  588, 588, 588, 588, 588, 588, 588, 588,  // 72 - 79  1680, 1680, 20499, 22547, 24595, 26643, 1776, 1776,  // 80 - 87  1808, 1808, -24557, -22509, -20461, -18413, 1904, 1904,  // 88 - 95  1936, 1936, -16365, -14317, 782, 782, 782, 782,  // 96 - 103  814, 814, 814, 814, -12269, -10221, 10257, 10257,  // 104 - 111  12305, 12305, 14353, 14353, 16403, 18451, 1712, 1712,  // 112 - 119  1744, 1744, 28691, 30739, -32749, -30701, -28653, -26605,  // 120 - 127  2061, 2061, 2061, 2061, 2061, 2061, 2061, 2061,  // 128 - 135  424, 424, 424, 424, 424, 424, 424, 424,  // 136 - 143  424, 424, 424, 424, 424, 424, 424, 424,  // 144 - 151  424, 424, 424, 424, 424, 424, 424, 424,  // 152 - 159  424, 424, 424, 424, 424, 424, 424, 424,  // 160 - 167  750, 750, 750, 750, 1616, 1616, 1648, 1648,  // 168 - 175  1424, 1424, 1456, 1456, 1488, 1488, 1520, 1520,  // 176 - 183  1840, 1840, 1872, 1872, 1968, 1968, 8209, 8209,  // 184 - 191  524, 524, 524, 524, 524, 524, 524, 524,  // 192 - 199  556, 556, 556, 556, 556, 556, 556, 556,  // 200 - 207  1552, 1552, 1584, 1584, 2000, 2000, 2032, 2032,  // 208 - 215  976, 976, 1008, 1008, 1040, 1040, 1072, 1072,  // 216 - 223  1296, 1296, 1328, 1328, 718, 718, 718, 718,  // 224 - 231  456, 456, 456, 456, 456, 456, 456, 456,  // 232 - 239  456, 456, 456, 456, 456, 456, 456, 456,  // 240 - 247  456, 456, 456, 456, 456, 456, 456, 456,  // 248 - 255  456, 456, 456, 456, 456, 456, 456, 456,  // 256 - 263  326, 326, 326, 326, 326, 326, 326, 326,  // 264 - 271  326, 326, 326, 326, 326, 326, 326, 326,  // 272 - 279  326, 326, 326, 326, 326, 326, 326, 326,  // 280 - 287  326, 326, 326, 326, 326, 326, 326, 326,  // 288 - 295  326, 326, 326, 326, 326, 326, 326, 326,  // 296 - 303  326, 326, 326, 326, 326, 326, 326, 326,  // 304 - 311  326, 326, 326, 326, 326, 326, 326, 326,  // 312 - 319  326, 326, 326, 326, 326, 326, 326, 326,  // 320 - 327  358, 358, 358, 358, 358, 358, 358, 358,  // 328 - 335  358, 358, 358, 358, 358, 358, 358, 358,  // 336 - 343  358, 358, 358, 358, 358, 358, 358, 358,  // 344 - 351  358, 358, 358, 358, 358, 358, 358, 358,  // 352 - 359  358, 358, 358, 358, 358, 358, 358, 358,  // 360 - 367  358, 358, 358, 358, 358, 358, 358, 358,  // 368 - 375  358, 358, 358, 358, 358, 358, 358, 358,  // 376 - 383  358, 358, 358, 358, 358, 358, 358, 358,  // 384 - 391  490, 490, 490, 490, 490, 490, 490, 490,  // 392 - 399  490, 490, 490, 490, 490, 490, 490, 490,  // 400 - 407  4113, 4113, 6161, 6161, 848, 848, 880, 880,  // 408 - 415  912, 912, 944, 944, 622, 622, 622, 622,  // 416 - 423  654, 654, 654, 654, 1104, 1104, 1136, 1136,  // 424 - 431  1168, 1168, 1200, 1200, 1232, 1232, 1264, 1264,  // 432 - 439  686, 686, 686, 686, 1360, 1360, 1392, 1392,  // 440 - 447  12, 12, 12, 12, 12, 12, 12, 12,  // 448 - 455  390, 390, 390, 390, 390, 390, 390, 390,  // 456 - 463  390, 390, 390, 390, 390, 390, 390, 390,  // 464 - 471  390, 390, 390, 390, 390, 390, 390, 390,  // 472 - 479  390, 390, 390, 390, 390, 390, 390, 390,  // 480 - 487  390, 390, 390, 390, 390, 390, 390, 390,  // 488 - 495  390, 390, 390, 390, 390, 390, 390, 390,  // 496 - 503  390, 390, 390, 390, 390, 390, 390, 390,  // 504 - 511  390, 390, 390, 390, 390, 390, 390, 390,  };  static byte twoDCodes[] = {  // 0 - 7  80, 88, 23, 71, 30, 30, 62, 62,  // 8 - 15  4, 4, 4, 4, 4, 4, 4, 4,  // 16 - 23  11, 11, 11, 11, 11, 11, 11, 11,  // 24 - 31  11, 11, 11, 11, 11, 11, 11, 11,  // 32 - 39  35, 35, 35, 35, 35, 35, 35, 35,  // 40 - 47  35, 35, 35, 35, 35, 35, 35, 35,  // 48 - 55  51, 51, 51, 51, 51, 51, 51, 51,  // 56 - 63  51, 51, 51, 51, 51, 51, 51, 51,  // 64 - 71  41, 41, 41, 41, 41, 41, 41, 41,  // 72 - 79  41, 41, 41, 41, 41, 41, 41, 41,  // 80 - 87  41, 41, 41, 41, 41, 41, 41, 41,  // 88 - 95  41, 41, 41, 41, 41, 41, 41, 41,  // 96 - 103  41, 41, 41, 41, 41, 41, 41, 41,  // 104 - 111  41, 41, 41, 41, 41, 41, 41, 41,  // 112 - 119  41, 41, 41, 41, 41, 41, 41, 41,  // 120 - 127  41, 41, 41, 41, 41, 41, 41, 41,  };  /\*\*  \* @param fillOrder The fill order of the compressed data bytes.  \* @param w  \* @param h  \*/  public TIFFFaxDecoder(long fillOrder, int w, int h) {  this.fillOrder = fillOrder;  this.w = w;  this.h = h;  this.bitPointer = 0;  this.bytePointer = 0;  this.prevChangingElems = new int[2\*w];  this.currChangingElems = new int[2\*w];  }  /\*\*  \* Reverses the bits in the array  \* @param b the bits to reverse  \*  \* @since 2.0.7  \*/  public static void reverseBits(byte[] b) {  for (int k = 0; k < b.length; ++k)  b[k] = flipTable[b[k] & 0xff];  }  // One-dimensional decoding methods  public void decode1D(byte[] buffer, byte[] compData, int startX, int height) {  this.data = compData;  int lineOffset = 0;  int scanlineStride = (w + 7)/8;  bitPointer = 0;  bytePointer = 0;  for (int i = 0; i < height; i++) {  decodeNextScanline(buffer, lineOffset, startX);  lineOffset += scanlineStride;  }  }  public void decodeNextScanline(byte[] buffer, int lineOffset, int bitOffset) {  int bits = 0, code = 0, isT = 0;  int current, entry, twoBits;  boolean isWhite = true;  // Initialize starting of the changing elements array  changingElemSize = 0;  // While scanline not complete  while (bitOffset < w) {  while (isWhite) {  // White run  current = nextNBits(10);  entry = white[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x0f;  if (bits == 12) { // Additional Make up code  // Get the next 2 bits  twoBits = nextLesserThan8Bits(2);  // Consolidate the 2 new bits and last 2 bits into 4 bits  current = ((current << 2) & 0x000c) | twoBits;  entry = additionalMakeup[current];  bits = (entry >>> 1) & 0x07; // 3 bits 0000 0111  code = (entry >>> 4) & 0x0fff; // 12 bits  bitOffset += code; // Skip white run  updatePointer(4 - bits);  } else if (bits == 0) { // ERROR  throw new RuntimeException(MessageLocalization.getComposedMessage("invalid.code.encountered"));  } else if (bits == 15) { // EOL  throw new RuntimeException(MessageLocalization.getComposedMessage("eol.code.word.encountered.in.white.run"));  } else {  // 11 bits - 0000 0111 1111 1111 = 0x07ff  code = (entry >>> 5) & 0x07ff;  bitOffset += code;  updatePointer(10 - bits);  if (isT == 0) {  isWhite = false;  currChangingElems[changingElemSize++] = bitOffset;  }  }  }  // Check whether this run completed one width, if so  // advance to next byte boundary for compression = 2.  if (bitOffset == w) {  if (compression == 2) {  advancePointer();  }  break;  }  while (!isWhite) {  // Black run  current = nextLesserThan8Bits(4);  entry = initBlack[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x000f;  code = (entry >>> 5) & 0x07ff;  if (code == 100) {  current = nextNBits(9);  entry = black[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x000f;  code = (entry >>> 5) & 0x07ff;  if (bits == 12) {  // Additional makeup codes  updatePointer(5);  current = nextLesserThan8Bits(4);  entry = additionalMakeup[current];  bits = (entry >>> 1) & 0x07; // 3 bits 0000 0111  code = (entry >>> 4) & 0x0fff; // 12 bits  setToBlack(buffer, lineOffset, bitOffset, code);  bitOffset += code;  updatePointer(4 - bits);  } else if (bits == 15) {  // EOL code  throw new RuntimeException(MessageLocalization.getComposedMessage("eol.code.word.encountered.in.black.run"));  } else {  setToBlack(buffer, lineOffset, bitOffset, code);  bitOffset += code;  updatePointer(9 - bits);  if (isT == 0) {  isWhite = true;  currChangingElems[changingElemSize++] = bitOffset;  }  }  } else if (code == 200) {  // Is a Terminating code  current = nextLesserThan8Bits(2);  entry = twoBitBlack[current];  code = (entry >>> 5) & 0x07ff;  bits = (entry >>> 1) & 0x0f;  setToBlack(buffer, lineOffset, bitOffset, code);  bitOffset += code;  updatePointer(2 - bits);  isWhite = true;  currChangingElems[changingElemSize++] = bitOffset;  } else {  // Is a Terminating code  setToBlack(buffer, lineOffset, bitOffset, code);  bitOffset += code;  updatePointer(4 - bits);  isWhite = true;  currChangingElems[changingElemSize++] = bitOffset;  }  }  // Check whether this run completed one width  if (bitOffset == w) {  if (compression == 2) {  advancePointer();  }  break;  }  }  currChangingElems[changingElemSize++] = bitOffset;  }  // Two-dimensional decoding methods  public void decode2D(byte[] buffer, byte compData[], int startX, int height, long tiffT4Options) {  this.data = compData;  compression = 3;  bitPointer = 0;  bytePointer = 0;  int scanlineStride = (w + 7)/8;  int a0, a1, b1, b2;  int[] b = new int[2];  int entry, code, bits;  boolean isWhite;  int currIndex = 0;  int temp[];  // fillBits - dealt with this in readEOL  // 1D/2D encoding - dealt with this in readEOL  // uncompressedMode - haven't dealt with this yet.  oneD = (int)(tiffT4Options & 0x01);  uncompressedMode = (int)((tiffT4Options & 0x02) >> 1);  fillBits = (int)((tiffT4Options & 0x04) >> 2);  // The data must start with an EOL code  if (readEOL(true) != 1) {  throw new RuntimeException(MessageLocalization.getComposedMessage("first.scanline.must.be.1d.encoded"));  }  int lineOffset = 0;  int bitOffset;  // Then the 1D encoded scanline data will occur, changing elements  // array gets set.  decodeNextScanline(buffer, lineOffset, startX);  lineOffset += scanlineStride;  for (int lines = 1; lines < height; lines++) {  // Every line must begin with an EOL followed by a bit which  // indicates whether the following scanline is 1D or 2D encoded.  if (readEOL(false) == 0) {  // 2D encoded scanline follows  // Initialize previous scanlines changing elements, and  // initialize current scanline's changing elements array  temp = prevChangingElems;  prevChangingElems = currChangingElems;  currChangingElems = temp;  currIndex = 0;  // a0 has to be set just before the start of this scanline.  a0 = -1;  isWhite = true;  bitOffset = startX;  lastChangingElement = 0;  while (bitOffset < w) {  // Get the next changing element  getNextChangingElement(a0, isWhite, b);  b1 = b[0];  b2 = b[1];  // Get the next seven bits  entry = nextLesserThan8Bits(7);  // Run these through the 2DCodes table  entry = twoDCodes[entry] & 0xff;  // Get the code and the number of bits used up  code = (entry & 0x78) >>> 3;  bits = entry & 0x07;  if (code == 0) {  if (!isWhite) {  setToBlack(buffer, lineOffset, bitOffset,  b2 - bitOffset);  }  bitOffset = a0 = b2;  // Set pointer to consume the correct number of bits.  updatePointer(7 - bits);  } else if (code == 1) {  // Horizontal  updatePointer(7 - bits);  // identify the next 2 codes.  int number;  if (isWhite) {  number = decodeWhiteCodeWord();  bitOffset += number;  currChangingElems[currIndex++] = bitOffset;  number = decodeBlackCodeWord();  setToBlack(buffer, lineOffset, bitOffset, number);  bitOffset += number;  currChangingElems[currIndex++] = bitOffset;  } else {  number = decodeBlackCodeWord();  setToBlack(buffer, lineOffset, bitOffset, number);  bitOffset += number;  currChangingElems[currIndex++] = bitOffset;  number = decodeWhiteCodeWord();  bitOffset += number;  currChangingElems[currIndex++] = bitOffset;  }  a0 = bitOffset;  } else if (code <= 8) {  // Vertical  a1 = b1 + (code - 5);  currChangingElems[currIndex++] = a1;  // We write the current color till a1 - 1 pos,  // since a1 is where the next color starts  if (!isWhite) {  setToBlack(buffer, lineOffset, bitOffset,  a1 - bitOffset);  }  bitOffset = a0 = a1;  isWhite = !isWhite;  updatePointer(7 - bits);  } else {  throw new RuntimeException(MessageLocalization.getComposedMessage("invalid.code.encountered.while.decoding.2d.group.3.compressed.data"));  }  }  // Add the changing element beyond the current scanline for the  // other color too  currChangingElems[currIndex++] = bitOffset;  changingElemSize = currIndex;  } else {  // 1D encoded scanline follows  decodeNextScanline(buffer, lineOffset, startX);  }  lineOffset += scanlineStride;  }  }  public void decodeT6(byte[] buffer,  byte[] compData,  int startX,  int height,  long tiffT6Options) {  this.data = compData;  compression = 4;  bitPointer = 0;  bytePointer = 0;  int scanlineStride = (w + 7)/8;  int a0, a1, b1, b2;  int entry, code, bits;  boolean isWhite;  int currIndex;  int temp[];  // Return values from getNextChangingElement  int[] b = new int[2];  // uncompressedMode - have written some code for this, but this  // has not been tested due to lack of test images using this optional  uncompressedMode = (int)((tiffT6Options & 0x02) >> 1);  // Local cached reference  int[] cce = currChangingElems;  // Assume invisible preceding row of all white pixels and insert  // both black and white changing elements beyond the end of this  // imaginary scanline.  changingElemSize = 0;  cce[changingElemSize++] = w;  cce[changingElemSize++] = w;  int lineOffset = 0;  int bitOffset;  for (int lines = 0; lines < height; lines++) {  // a0 has to be set just before the start of the scanline.  a0 = -1;  isWhite = true;  // Assign the changing elements of the previous scanline to  // prevChangingElems and start putting this new scanline's  // changing elements into the currChangingElems.  temp = prevChangingElems;  prevChangingElems = currChangingElems;  cce = currChangingElems = temp;  currIndex = 0;  // Start decoding the scanline at startX in the raster  bitOffset = startX;  // Reset search start position for getNextChangingElement  lastChangingElement = 0;  // Till one whole scanline is decoded  escape:  while (bitOffset < w && bytePointer < data.length - 1) {  // Get the next changing element  getNextChangingElement(a0, isWhite, b);  b1 = b[0];  b2 = b[1];  // Get the next seven bits  entry = nextLesserThan8Bits(7);  // Run these through the 2DCodes table  entry = twoDCodes[entry] & 0xff;  // Get the code and the number of bits used up  code = (entry & 0x78) >>> 3;  bits = entry & 0x07;  if (code == 0) { // Pass  // We always assume WhiteIsZero format for fax.  if (!isWhite) {  setToBlack(buffer, lineOffset, bitOffset,  b2 - bitOffset);  }  bitOffset = a0 = b2;  // Set pointer to only consume the correct number of bits.  updatePointer(7 - bits);  } else if (code == 1) { // Horizontal  // Set pointer to only consume the correct number of bits.  updatePointer(7 - bits);  // identify the next 2 alternating color codes.  int number;  if (isWhite) {  // Following are white and black runs  number = decodeWhiteCodeWord();  bitOffset += number;  cce[currIndex++] = bitOffset;  number = decodeBlackCodeWord();  setToBlack(buffer, lineOffset, bitOffset, number);  bitOffset += number;  cce[currIndex++] = bitOffset;  } else {  // First a black run and then a white run follows  number = decodeBlackCodeWord();  setToBlack(buffer, lineOffset, bitOffset, number);  bitOffset += number;  cce[currIndex++] = bitOffset;  number = decodeWhiteCodeWord();  bitOffset += number;  cce[currIndex++] = bitOffset;  }  a0 = bitOffset;  } else if (code <= 8) { // Vertical  a1 = b1 + (code - 5);  cce[currIndex++] = a1;  // We write the current color till a1 - 1 pos,  // since a1 is where the next color starts  if (!isWhite) {  setToBlack(buffer, lineOffset, bitOffset,  a1 - bitOffset);  }  bitOffset = a0 = a1;  isWhite = !isWhite;  updatePointer(7 - bits);  } else if (code == 11) {  if (nextLesserThan8Bits(3) != 7) {  throw new InvalidImageException(MessageLocalization.getComposedMessage("invalid.code.encountered.while.decoding.2d.group.4.compressed.data"));  }  int zeros = 0;  boolean exit = false;  while (!exit) {  while (nextLesserThan8Bits(1) != 1) {  zeros++;  }  if (zeros > 5) {  // Exit code  // Zeros before exit code  zeros = zeros - 6;  if (!isWhite && (zeros > 0)) {  cce[currIndex++] = bitOffset;  }  // Zeros before the exit code  bitOffset += zeros;  if (zeros > 0) {  // Some zeros have been written  isWhite = true;  }  // Read in the bit which specifies the color of  // the following run  if (nextLesserThan8Bits(1) == 0) {  if (!isWhite) {  cce[currIndex++] = bitOffset;  }  isWhite = true;  } else {  if (isWhite) {  cce[currIndex++] = bitOffset;  }  isWhite = false;  }  exit = true;  }  if (zeros == 5) {  if (!isWhite) {  cce[currIndex++] = bitOffset;  }  bitOffset += zeros;  // Last thing written was white  isWhite = true;  } else {  bitOffset += zeros;  cce[currIndex++] = bitOffset;  setToBlack(buffer, lineOffset, bitOffset, 1);  ++bitOffset;  // Last thing written was black  isWhite = false;  }  }  } else {  //micah\_tessler@yahoo.com  //Microsoft TIFF renderers seem to treat unknown codes as line-breaks  //That is, they give up on the current line and move on to the next one  //set bitOffset to w to move on to the next scan line.  bitOffset = w;  updatePointer(7 - bits);  }  } // end loop  // Add the changing element beyond the current scanline for the  // other color too  //make sure that the index does not exceed the bounds of the array  if(currIndex < cce.length)  cce[currIndex++] = bitOffset;  // Number of changing elements in this scanline.  changingElemSize = currIndex;  lineOffset += scanlineStride;  }  }  private void setToBlack(byte[] buffer,  int lineOffset, int bitOffset,  int numBits) {  int bitNum = 8\*lineOffset + bitOffset;  int lastBit = bitNum + numBits;  int byteNum = bitNum >> 3;  // Handle bits in first byte  int shift = bitNum & 0x7;  if (shift > 0) {  int maskVal = 1 << (7 - shift);  byte val = buffer[byteNum];  while (maskVal > 0 && bitNum < lastBit) {  val |= maskVal;  maskVal >>= 1;  ++bitNum;  }  buffer[byteNum] = val;  }  // Fill in 8 bits at a time  byteNum = bitNum >> 3;  while (bitNum < lastBit - 7) {  buffer[byteNum++] = (byte)255;  bitNum += 8;  }  // Fill in remaining bits  while (bitNum < lastBit) {  byteNum = bitNum >> 3;  if ( recoverFromImageError && !(byteNum < buffer.length) ) {  // do nothing  } else {  buffer[byteNum] |= 1 << (7 - (bitNum & 0x7));  }  ++bitNum;  }  }  // Returns run length  private int decodeWhiteCodeWord() {  int current, entry, bits, isT, twoBits, code = -1;  int runLength = 0;  boolean isWhite = true;  while (isWhite) {  current = nextNBits(10);  entry = white[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x0f;  if (bits == 12) { // Additional Make up code  // Get the next 2 bits  twoBits = nextLesserThan8Bits(2);  // Consolidate the 2 new bits and last 2 bits into 4 bits  current = ((current << 2) & 0x000c) | twoBits;  entry = additionalMakeup[current];  bits = (entry >>> 1) & 0x07; // 3 bits 0000 0111  code = (entry >>> 4) & 0x0fff; // 12 bits  runLength += code;  updatePointer(4 - bits);  } else if (bits == 0) { // ERROR  throw new InvalidImageException(MessageLocalization.getComposedMessage("invalid.code.encountered"));  } else if (bits == 15) { // EOL  if ( runLength == 0 ) {  isWhite = false;  } else {  throw new RuntimeException(MessageLocalization.getComposedMessage("eol.code.word.encountered.in.white.run"));  }  } else {  // 11 bits - 0000 0111 1111 1111 = 0x07ff  code = (entry >>> 5) & 0x07ff;  runLength += code;  updatePointer(10 - bits);  if (isT == 0) {  isWhite = false;  }  }  }  return runLength;  }  // Returns run length  private int decodeBlackCodeWord() {  int current, entry, bits, isT, code = -1;  int runLength = 0;  boolean isWhite = false;  while (!isWhite) {  current = nextLesserThan8Bits(4);  entry = initBlack[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x000f;  code = (entry >>> 5) & 0x07ff;  if (code == 100) {  current = nextNBits(9);  entry = black[current];  // Get the 3 fields from the entry  isT = entry & 0x0001;  bits = (entry >>> 1) & 0x000f;  code = (entry >>> 5) & 0x07ff;  if (bits == 12) {  // Additional makeup codes  updatePointer(5);  current = nextLesserThan8Bits(4);  entry = additionalMakeup[current];  bits = (entry >>> 1) & 0x07; // 3 bits 0000 0111  code = (entry >>> 4) & 0x0fff; // 12 bits  runLength += code;  updatePointer(4 - bits);  } else if (bits == 15) {  // EOL code  throw new RuntimeException(MessageLocalization.getComposedMessage("eol.code.word.encountered.in.black.run"));  } else {  runLength += code;  updatePointer(9 - bits);  if (isT == 0) {  isWhite = true;  }  }  } else if (code == 200) {  // Is a Terminating code  current = nextLesserThan8Bits(2);  entry = twoBitBlack[current];  code = (entry >>> 5) & 0x07ff;  runLength += code;  bits = (entry >>> 1) & 0x0f;  updatePointer(2 - bits);  isWhite = true;  } else {  // Is a Terminating code  runLength += code;  updatePointer(4 - bits);  isWhite = true;  }  }  return runLength;  }  private int readEOL(boolean isFirstEOL) {  if (fillBits == 0) {  int next12Bits = nextNBits(12);  if (isFirstEOL && next12Bits == 0) {  // Might have the case of EOL padding being used even  // though it was not flagged in the T4Options field.  // This was observed to be the case in TIFFs produced  // by a well known vendor who shall remain nameless.  if(nextNBits(4) == 1) {  // EOL must be padded: reset the fillBits flag.  fillBits = 1;  return 1;  }  }  if(next12Bits != 1) {  throw new RuntimeException(MessageLocalization.getComposedMessage("scanline.must.begin.with.eol.code.word"));  }  } else if (fillBits == 1) {  // First EOL code word xxxx 0000 0000 0001 will occur  // As many fill bits will be present as required to make  // the EOL code of 12 bits end on a byte boundary.  int bitsLeft = 8 - bitPointer;  if (nextNBits(bitsLeft) != 0) {  throw new RuntimeException(MessageLocalization.getComposedMessage("all.fill.bits.preceding.eol.code.must.be.0"));  }  // If the number of bitsLeft is less than 8, then to have a 12  // bit EOL sequence, two more bytes are certainly going to be  // required. The first of them has to be all zeros, so ensure  // that.  if (bitsLeft < 4) {  if (nextNBits(8) != 0) {  throw new RuntimeException(MessageLocalization.getComposedMessage("all.fill.bits.preceding.eol.code.must.be.0"));  }  }  // There might be a random number of fill bytes with 0s, so  // loop till the EOL of 0000 0001 is found, as long as all  // the bytes preceding it are 0's.  int n;  while ((n = nextNBits(8)) != 1) {  // If not all zeros  if (n != 0) {  throw new RuntimeException(MessageLocalization.getComposedMessage("all.fill.bits.preceding.eol.code.must.be.0"));  }  }  }  // If one dimensional encoding mode, then always return 1  if (oneD == 0) {  return 1;  } else {  // Otherwise for 2D encoding mode,  // The next one bit signifies 1D/2D encoding of next line.  return nextLesserThan8Bits(1);  }  }  private void getNextChangingElement(int a0, boolean isWhite, int[] ret) {  // Local copies of instance variables  int[] pce = this.prevChangingElems;  int ces = this.changingElemSize;  // If the previous match was at an odd element, we still  // have to search the preceeding element.  // int start = lastChangingElement & ~0x1;  int start = lastChangingElement > 0 ? lastChangingElement - 1 : 0;  if (isWhite) {  start &= ~0x1; // Search even numbered elements  } else {  start |= 0x1; // Search odd numbered elements  }  int i = start;  for (; i < ces; i += 2) {  int temp = pce[i];  if (temp > a0) {  lastChangingElement = i;  ret[0] = temp;  break;  }  }  if (i + 1 < ces) {  ret[1] = pce[i + 1];  }  }  private int nextNBits(int bitsToGet) {  byte b, next, next2next;  int l = data.length - 1;  int bp = this.bytePointer;  if (fillOrder == 1) {  b = data[bp];  if (bp == l) {  next = 0x00;  next2next = 0x00;  } else if ((bp + 1) == l) {  next = data[bp + 1];  next2next = 0x00;  } else {  next = data[bp + 1];  next2next = data[bp + 2];  }  } else if (fillOrder == 2) {  b = flipTable[data[bp] & 0xff];  if (bp == l) {  next = 0x00;  next2next = 0x00;  } else if ((bp + 1) == l) {  next = flipTable[data[bp + 1] & 0xff];  next2next = 0x00;  } else {  next = flipTable[data[bp + 1] & 0xff];  next2next = flipTable[data[bp + 2] & 0xff];  }  } else {  throw new RuntimeException(MessageLocalization.getComposedMessage("tiff.fill.order.tag.must.be.either.1.or.2"));  }  int bitsLeft = 8 - bitPointer;  int bitsFromNextByte = bitsToGet - bitsLeft;  int bitsFromNext2NextByte = 0;  if (bitsFromNextByte > 8) {  bitsFromNext2NextByte = bitsFromNextByte - 8;  bitsFromNextByte = 8;  }  bytePointer++;  int i1 = (b & table1[bitsLeft]) << (bitsToGet - bitsLeft);  int i2 = (next & table2[bitsFromNextByte]) >>> (8 - bitsFromNextByte);  int i3 = 0;  if (bitsFromNext2NextByte != 0) {  i2 <<= bitsFromNext2NextByte;  i3 = (next2next & table2[bitsFromNext2NextByte]) >>>  (8 - bitsFromNext2NextByte);  i2 |= i3;  bytePointer++;  bitPointer = bitsFromNext2NextByte;  } else {  if (bitsFromNextByte == 8) {  bitPointer = 0;  bytePointer++;  } else {  bitPointer = bitsFromNextByte;  }  }  int i = i1 | i2;  return i;  }  private int nextLesserThan8Bits(int bitsToGet) {  byte b = 0, next = 0;  int l = data.length - 1;  int bp = this.bytePointer;  if (fillOrder == 1) {  b = data[bp];  if (bp == l) {  next = 0x00;  } else {  next = data[bp + 1];  }  } else if (fillOrder == 2) {  if ( recoverFromImageError && !(bp < data.length) ) {  // do nothing  } else {  b = flipTable[data[bp] & 0xff];  if (bp == l) {  next = 0x00;  } else {  next = flipTable[data[bp + 1] & 0xff];  }  }  } else {  throw new RuntimeException(MessageLocalization.getComposedMessage("tiff.fill.order.tag.must.be.either.1.or.2"));  }  int bitsLeft = 8 - bitPointer;  int bitsFromNextByte = bitsToGet - bitsLeft;  int shift = bitsLeft - bitsToGet;  int i1, i2;  if (shift >= 0) {  i1 = (b & table1[bitsLeft]) >>> shift;  bitPointer += bitsToGet;  if (bitPointer == 8) {  bitPointer = 0;  bytePointer++;  }  } else {  i1 = (b & table1[bitsLeft]) << (-shift);  i2 = (next & table2[bitsFromNextByte]) >>> (8 - bitsFromNextByte);  i1 |= i2;  bytePointer++;  bitPointer = bitsFromNextByte;  }  return i1;  }  // Move pointer backwards by given amount of bits  private void updatePointer(int bitsToMoveBack) {  int i = bitPointer - bitsToMoveBack;  if (i < 0) {  bytePointer--;  bitPointer = 8 + i;  } else {  bitPointer = i;  }  }  // Move to the next byte boundary  private boolean advancePointer() {  if (bitPointer != 0) {  bytePointer++;  bitPointer = 0;  }  return true;  }  public void setRecoverFromImageError(boolean recoverFromImageError) {  this.recoverFromImageError = recoverFromImageError;  }  } |
|  |
| /\*\*  \* Handles CCITTFAXDECODE filter  \*/  private static class Filter\_CCITTFAXDECODE implements FilterHandler{  public byte[] decode(byte[] b, PdfName filterName, PdfObject decodeParams, PdfDictionary streamDictionary) throws IOException {  PdfNumber wn = (PdfNumber)PdfReader.getPdfObjectRelease(streamDictionary.get(PdfName.WIDTH));  PdfNumber hn = (PdfNumber)PdfReader.getPdfObjectRelease(streamDictionary.get(PdfName.HEIGHT));  if (wn == null || hn == null)  throw new UnsupportedPdfException(MessageLocalization.getComposedMessage("filter.ccittfaxdecode.is.only.supported.for.images"));  int width = wn.intValue();  int height = hn.intValue();  PdfDictionary param = decodeParams instanceof PdfDictionary ? (PdfDictionary)decodeParams : null;  int k = 0;  boolean blackIs1 = false;  boolean byteAlign = false;  if (param != null) {  PdfNumber kn = param.getAsNumber(PdfName.K);  if (kn != null)  k = kn.intValue();  PdfBoolean bo = param.getAsBoolean(PdfName.BLACKIS1);  if (bo != null)  blackIs1 = bo.booleanValue();  bo = param.getAsBoolean(PdfName.ENCODEDBYTEALIGN);  if (bo != null)  byteAlign = bo.booleanValue();  }  byte[] outBuf = new byte[(width + 7) / 8 \* height];  TIFFFaxDecompressor decoder = new TIFFFaxDecompressor();  if (k == 0 || k > 0) {  int tiffT4Options = k > 0 ? TIFFConstants.GROUP3OPT\_2DENCODING : 0;  tiffT4Options |= byteAlign ? TIFFConstants.GROUP3OPT\_FILLBITS : 0;  decoder.SetOptions(1, TIFFConstants.COMPRESSION\_CCITTFAX3, tiffT4Options, 0);  decoder.decodeRaw(outBuf, b, width, height);  if (decoder.fails > 0) {  byte[] outBuf2 = new byte[(width + 7) / 8 \* height];  int oldFails = decoder.fails;  decoder.SetOptions(1, TIFFConstants.COMPRESSION\_CCITTRLE, tiffT4Options, 0);  decoder.decodeRaw(outBuf2, b, width, height);  if (decoder.fails < oldFails) {  outBuf = outBuf2;  }  }  }  else {  TIFFFaxDecoder deca = new TIFFFaxDecoder(1, width, height);  deca.decodeT6(outBuf, b, 0, height, 0);  }  if (!blackIs1) {  int len = outBuf.length;  for (int t = 0; t < len; ++t) {  outBuf[t] ^= 0xff;  }  }  b = outBuf;  return b;  }  } |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

***JBIG2Decode***   
JBIG2是JBIG的改进版本。   
JBIG(Joint Bi-level Image Experts Group，联合二值图像专家组)是发布二值图像编码标准的专家组。在官方来说，JBIG是ISO/IEC JTC1 SC29工作组1，这个工作组也负责JPEG标准，它是一套压缩算法，用来产生Web浏览器支持的以及典型地用于复杂图像(例如照片)的图像文件。　　   
JBIG已经发布了一个二值图像压缩的标准，二值图像是一种只用1bit来表达每个像素的颜色值的图像。这个标准也可用于对灰度图像以及每个像素使用有限个比特的彩色图像进行编码。JBIG专门用于使用传真编码发送的图像，它提供了比组3和组4的传真编码好的多的压缩。   
算法如下（pdfbox）：

/\*\*

\* Decompresses data encoded using the JBIG2 standard, reproducing the original

\* monochrome (1 bit per pixel) image data (or an approximation of that data).

\*

\* Requires a JBIG2 plugin for Java Image I/O to be installed. A known working

\* plug-in is the Apache PDFBox JBIG2 plugin.

\*

\* @author Timo Boehme

\*/

final class JBIG2Filter extends Filter

{

private static final Log LOG = LogFactory.getLog(JBIG2Filter.class);

private static boolean levigoLogged = false;

private static synchronized void logLevigoDonated()

{

if (!levigoLogged)

{

LOG.info("The Levigo JBIG2 plugin has been donated to the Apache Foundation");

LOG.info("and an improved version is available for download at "

+ "https://pdfbox.apache.org/download.cgi");

levigoLogged = true;

}

}

@Override

public DecodeResult decode(InputStream encoded, OutputStream decoded, COSDictionary

parameters, int index, DecodeOptions options) throws IOException

{

ImageReader reader = findImageReader("JBIG2", "jbig2-imageio is not installed");

if (reader.getClass().getName().contains("levigo"))

{

logLevigoDonated();

}

int bits = parameters.getInt(COSName.BITS\_PER\_COMPONENT, 1);

COSDictionary params = getDecodeParams(parameters, index);

ImageReadParam irp = reader.getDefaultReadParam();

irp.setSourceSubsampling(options.getSubsamplingX(), options.getSubsamplingY(),

options.getSubsamplingOffsetX(), options.getSubsamplingOffsetY());

irp.setSourceRegion(options.getSourceRegion());

options.setFilterSubsampled(true);

COSStream globals = null;

if (params != null)

{

globals = (COSStream) params.getDictionaryObject(COSName.JBIG2\_GLOBALS);

}

ImageInputStream iis = null;

try

{

if (globals != null)

{

iis = ImageIO.createImageInputStream(

new SequenceInputStream(globals.createInputStream(), encoded));

reader.setInput(iis);

}

else

{

iis = ImageIO.createImageInputStream(encoded);

reader.setInput(iis);

}

BufferedImage image;

try

{

image = reader.read(0, irp);

}

catch (Exception e)

{

// wrap and rethrow any exceptions

throw new IOException("Could not read JBIG2 image", e);

}

// I am assuming since JBIG2 is always black and white

// depending on your renderer this might or might be needed

if (image.getColorModel().getPixelSize() != bits)

{

if (bits != 1)

{

LOG.warn("Attempting to handle a JBIG2 with more than 1-bit depth");

}

BufferedImage packedImage = new BufferedImage(image.getWidth(), image.getHeight(),

BufferedImage.TYPE\_BYTE\_BINARY);

Graphics graphics = packedImage.getGraphics();

graphics.drawImage(image, 0, 0, null);

graphics.dispose();

image = packedImage;

}

DataBuffer dBuf = image.getData().getDataBuffer();

if (dBuf.getDataType() == DataBuffer.TYPE\_BYTE)

{

decoded.write(((DataBufferByte) dBuf).getData());

}

else

{

throw new IOException("Unexpected image buffer type");

}

}

finally

{

if (iis != null)

{

iis.close();

}

reader.dispose();

}

return new DecodeResult(parameters);

}

@Override

public DecodeResult decode(InputStream encoded, OutputStream decoded,

COSDictionary parameters, int index) throws IOException

{

return decode(encoded, decoded, parameters, index, DecodeOptions.DEFAULT);

}

@Override

protected void encode(InputStream input, OutputStream encoded, COSDictionary parameters)

throws IOException

{

throw new UnsupportedOperationException("JBIG2 encoding not implemented");

}

}

***DCTDecode***   
该压缩方式（JPEG）应用于灰度图和彩色图的压缩。该压缩方式有损压缩。   
算法如下（pdfbox）：

/\*\*

\* Decompresses data encoded using a DCT (discrete cosine transform)

\* technique based on the JPEG standard.

\*

\* @author John Hewson

\*/

final class DCTFilter extends Filter

{

private static final Log LOG = LogFactory.getLog(DCTFilter.class);

private static final int POS\_TRANSFORM = 11;

private static final String ADOBE = "Adobe";

@Override

public DecodeResult decode(InputStream encoded, OutputStream decoded, COSDictionary

parameters, int index, DecodeOptions options) throws IOException

{

ImageReader reader = findImageReader("JPEG", "a suitable JAI I/O image filter is not installed");

ImageInputStream iis = null;

try

{

iis = ImageIO.createImageInputStream(encoded);

// skip one LF if there

if (iis.read() != 0x0A)

{

iis.seek(0);

}

reader.setInput(iis);

ImageReadParam irp = reader.getDefaultReadParam();

irp.setSourceSubsampling(options.getSubsamplingX(), options.getSubsamplingY(),

options.getSubsamplingOffsetX(), options.getSubsamplingOffsetY());

irp.setSourceRegion(options.getSourceRegion());

options.setFilterSubsampled(true);

String numChannels = getNumChannels(reader);

// get the raster using horrible JAI workarounds

ImageIO.setUseCache(false);

Raster raster;

// Strategy: use read() for RGB or "can't get metadata"

// use readRaster() for CMYK and gray and as fallback if read() fails

// after "can't get metadata" because "no meta" file was CMYK

if ("3".equals(numChannels) || numChannels.isEmpty())

{

try

{

// I'd like to use ImageReader#readRaster but it is buggy and can't read RGB correctly

BufferedImage image = reader.read(0, irp);

raster = image.getRaster();

}

catch (IIOException e)

{

// JAI can't read CMYK JPEGs using ImageReader#read or ImageIO.read but

// fortunately ImageReader#readRaster isn't buggy when reading 4-channel files

raster = reader.readRaster(0, irp);

}

}

else

{

// JAI can't read CMYK JPEGs using ImageReader#read or ImageIO.read but

// fortunately ImageReader#readRaster isn't buggy when reading 4-channel files

raster = reader.readRaster(0, irp);

}

// special handling for 4-component images

if (raster.getNumBands() == 4)

{

// get APP14 marker

Integer transform;

try

{

transform = getAdobeTransform(reader.getImageMetadata(0));

}

catch (IIOException e)

{

// we really tried asking nicely, now we're using brute force.

transform = getAdobeTransformByBruteForce(iis);

}

catch (NegativeArraySizeException e)

{

// we really tried asking nicely, now we're using brute force.

transform = getAdobeTransformByBruteForce(iis);

}

int colorTransform = transform != null ? transform : 0;

// 0 = Unknown (RGB or CMYK), 1 = YCbCr, 2 = YCCK

switch (colorTransform)

{

case 0:

// already CMYK

break;

case 1:

raster = fromYCbCrtoCMYK(raster);

break;

case 2:

raster = fromYCCKtoCMYK(raster);

break;

default:

throw new IllegalArgumentException("Unknown colorTransform");

}

}

else if (raster.getNumBands() == 3)

{

// BGR to RGB

raster = fromBGRtoRGB(raster);

}

DataBufferByte dataBuffer = (DataBufferByte)raster.getDataBuffer();

decoded.write(dataBuffer.getData());

}

finally

{

if (iis != null)

{

iis.close();

}

reader.dispose();

}

return new DecodeResult(parameters);

}

@Override

public DecodeResult decode(InputStream encoded, OutputStream decoded,

COSDictionary parameters, int index) throws IOException

{

return decode(encoded, decoded, parameters, index, DecodeOptions.DEFAULT);

}

// reads the APP14 Adobe transform tag and returns its value, or 0 if unknown

private Integer getAdobeTransform(IIOMetadata metadata)

{

Element tree = (Element)metadata.getAsTree("javax\_imageio\_jpeg\_image\_1.0");

Element markerSequence = (Element)tree.getElementsByTagName("markerSequence").item(0);

NodeList app14AdobeNodeList = markerSequence.getElementsByTagName("app14Adobe");

if (app14AdobeNodeList != null && app14AdobeNodeList.getLength() > 0)

{

Element adobe = (Element) app14AdobeNodeList.item(0);

return Integer.parseInt(adobe.getAttribute("transform"));

}

return 0;

}

// See in https://github.com/haraldk/TwelveMonkeys

// com.twelvemonkeys.imageio.plugins.jpeg.AdobeDCT class for structure of APP14 segment

private int getAdobeTransformByBruteForce(ImageInputStream iis) throws IOException

{

int a = 0;

iis.seek(0);

int by;

while ((by = iis.read()) != -1)

{

if (ADOBE.charAt(a) == by)

{

++a;

if (a != ADOBE.length())

{

continue;

}

// match

a = 0;

long afterAdobePos = iis.getStreamPosition();

iis.seek(iis.getStreamPosition() - 9);

int tag = iis.readUnsignedShort();

if (tag != 0xFFEE)

{

iis.seek(afterAdobePos);

continue;

}

int len = iis.readUnsignedShort();

if (len >= POS\_TRANSFORM + 1)

{

byte[] app14 = new byte[Math.max(len, POS\_TRANSFORM + 1)];

if (iis.read(app14) >= POS\_TRANSFORM + 1)

{

return app14[POS\_TRANSFORM];

}

}

}

else

{

a = 0;

}

}

return 0;

}

// converts YCCK image to CMYK. YCCK is an equivalent encoding for

// CMYK data, so no color management code is needed here, nor does the

// PDF color space have to be consulted

private WritableRaster fromYCCKtoCMYK(Raster raster)

{

WritableRaster writableRaster = raster.createCompatibleWritableRaster();

int[] value = new int[4];

for (int y = 0, height = raster.getHeight(); y < height; y++)

{

for (int x = 0, width = raster.getWidth(); x < width; x++)

{

raster.getPixel(x, y, value);

// 4-channels 0..255

float Y = value[0];

float Cb = value[1];

float Cr = value[2];

float K = value[3];

// YCCK to RGB, see http://software.intel.com/en-us/node/442744

int r = clamp(Y + 1.402f \* Cr - 179.456f);

int g = clamp(Y - 0.34414f \* Cb - 0.71414f \* Cr + 135.45984f);

int b = clamp(Y + 1.772f \* Cb - 226.816f);

// naive RGB to CMYK

int cyan = 255 - r;

int magenta = 255 - g;

int yellow = 255 - b;

// update new raster

value[0] = cyan;

value[1] = magenta;

value[2] = yellow;

value[3] = (int)K;

writableRaster.setPixel(x, y, value);

}

}

return writableRaster;

}

private WritableRaster fromYCbCrtoCMYK(Raster raster)

{

WritableRaster writableRaster = raster.createCompatibleWritableRaster();

int[] value = new int[4];

for (int y = 0, height = raster.getHeight(); y < height; y++)

{

for (int x = 0, width = raster.getWidth(); x < width; x++)

{

raster.getPixel(x, y, value);

// 4-channels 0..255

float Y = value[0];

float Cb = value[1];

float Cr = value[2];

float K = value[3];

// YCbCr to RGB, see http://www.equasys.de/colorconversion.html

int r = clamp( (1.164f \* (Y-16)) + (1.596f \* (Cr - 128)) );

int g = clamp( (1.164f \* (Y-16)) + (-0.392f \* (Cb-128)) + (-0.813f \* (Cr-128)));

int b = clamp( (1.164f \* (Y-16)) + (2.017f \* (Cb-128)));

// naive RGB to CMYK

int cyan = 255 - r;

int magenta = 255 - g;

int yellow = 255 - b;

// update new raster

value[0] = cyan;

value[1] = magenta;

value[2] = yellow;

value[3] = (int)K;

writableRaster.setPixel(x, y, value);

}

}

return writableRaster;

}

// converts from BGR to RGB

private WritableRaster fromBGRtoRGB(Raster raster)

{

WritableRaster writableRaster = raster.createCompatibleWritableRaster();

int width = raster.getWidth();

int height = raster.getHeight();

int w3 = width \* 3;

int[] tab = new int[w3];

//BEWARE: handling the full image at a time is slower than one line at a time

for (int y = 0; y < height; y++)

{

raster.getPixels(0, y, width, 1, tab);

for (int off = 0; off < w3; off += 3)

{

int tmp = tab[off];

tab[off] = tab[off + 2];

tab[off + 2] = tmp;

}

writableRaster.setPixels(0, y, width, 1, tab);

}

return writableRaster;

}

// returns the number of channels as a string, or an empty string if there is an error getting the meta data

private String getNumChannels(ImageReader reader)

{

try

{

IIOMetadata imageMetadata = reader.getImageMetadata(0);

if (imageMetadata == null)

{

return "";

}

IIOMetadataNode metaTree = (IIOMetadataNode) imageMetadata.getAsTree("javax\_imageio\_1.0");

Element numChannelsItem = (Element) metaTree.getElementsByTagName("NumChannels").item(0);

if (numChannelsItem == null)

{

return "";

}

return numChannelsItem.getAttribute("value");

}

catch (IOException e)

{

return "";

}

catch (NegativeArraySizeException e)

{

return "";

}

}

// clamps value to 0-255 range

private int clamp(float value)

{

return (int)((value < 0) ? 0 : ((value > 255) ? 255 : value));

}

@Override

protected void encode(InputStream input, OutputStream encoded, COSDictionary parameters)

throws IOException

{

throw new UnsupportedOperationException("DCTFilter encoding not implemented, use the JPEGFactory methods instead");

}

}

***JPXDecode***   
该压缩方式是JPEG2000，JPEG 2000是基于小波变换的图像压缩标准，由Joint Photographic Experts Group组织创建和维护。JPEG 2000通常被认为是未来取代JPEG（基于离散余弦变换）的下一代图像压缩标准。JPEG 2000文件的副档名通常为.jp2，MIME类型是image/jp2。   
JPEG2000的压缩比更高，而且不会产生原先的基于离散余弦变换的JPEG标准产生的块状模糊瑕疵。JPEG2000同时支持有损压缩和无损压缩。另外，JPEG2000也支持更复杂的渐进式显示和下载。   
JPEG2000是国际标准化组织（ISO）发布的标准，文档代码为ISO/IEC 15444-1:2000。虽然JPEG2000在技术上有一定的优势，但是到目前为止（2006年），网络上采用JPEG2000技术制作的图像文件数量仍然很少，并且大多数的浏览器仍然没有内置支持JPEG2000图像文件的显示。但是，由于JPEG2000在无损压缩下仍然能有比较好的压缩率，所以JPEG2000在图像品质要求比较高的医学图像的分析和处理中已经有了一定程度的广泛应用。   
算法如下（pdfbox）：

/\*\*

\* Decompress data encoded using the wavelet-based JPEG 2000 standard,

\* reproducing the original data.

\*

\* Requires the Java Advanced Imaging (JAI) Image I/O Tools to be installed from java.net, see

\* <a href="http://download.java.net/media/jai-imageio/builds/release/1.1/">jai-imageio</a>.

\* Alternatively you can build from the source available in the

\* <a href="https://java.net/projects/jai-imageio-core/">jai-imageio-core svn repo</a>.

\*

\* Mac OS X users should download the tar.gz file for linux and unpack it to obtain the

\* required jar files. The .so file can be safely ignored.

\*

\* @author John Hewson

\* @author Timo Boehme

\*/

public final class JPXFilter extends Filter

{

@Override

public DecodeResult decode(InputStream encoded, OutputStream decoded, COSDictionary

parameters, int index, DecodeOptions options) throws IOException

{

DecodeResult result = new DecodeResult(new COSDictionary());

result.getParameters().addAll(parameters);

BufferedImage image = readJPX(encoded, options, result);

WritableRaster raster = image.getRaster();

switch (raster.getDataBuffer().getDataType())

{

case DataBuffer.TYPE\_BYTE:

DataBufferByte byteBuffer = (DataBufferByte) raster.getDataBuffer();

decoded.write(byteBuffer.getData());

return result;

case DataBuffer.TYPE\_USHORT:

DataBufferUShort wordBuffer = (DataBufferUShort) raster.getDataBuffer();

for (short w : wordBuffer.getData())

{

decoded.write(w >> 8);

decoded.write(w);

}

return result;

default:

throw new IOException("Data type " + raster.getDataBuffer().getDataType() + " not implemented");

}

}

@Override

public DecodeResult decode(InputStream encoded, OutputStream decoded,

COSDictionary parameters, int index) throws IOException

{

return decode(encoded, decoded, parameters, index, DecodeOptions.DEFAULT);

}

// try to read using JAI Image I/O

private BufferedImage readJPX(InputStream input, DecodeOptions options, DecodeResult result) throws IOException

{

ImageReader reader = findImageReader("JPEG2000", "Java Advanced Imaging (JAI) Image I/O Tools are not installed");

ImageInputStream iis = null;

try

{

// PDFBOX-4121: ImageIO.createImageInputStream() is much slower

iis = new MemoryCacheImageInputStream(input);

reader.setInput(iis, true, true);

ImageReadParam irp = reader.getDefaultReadParam();

irp.setSourceRegion(options.getSourceRegion());

irp.setSourceSubsampling(options.getSubsamplingX(), options.getSubsamplingY(),

options.getSubsamplingOffsetX(), options.getSubsamplingOffsetY());

options.setFilterSubsampled(true);

BufferedImage image;

try

{

image = reader.read(0, irp);

}

catch (Exception e)

{

// wrap and rethrow any exceptions

throw new IOException("Could not read JPEG 2000 (JPX) image", e);

}

COSDictionary parameters = result.getParameters();

// "If the image stream uses the JPXDecode filter, this entry is optional

// and shall be ignored if present"

//

// note that indexed color spaces make the BPC logic tricky, see PDFBOX-2204

int bpc = image.getColorModel().getPixelSize() / image.getRaster().getNumBands();

parameters.setInt(COSName.BITS\_PER\_COMPONENT, bpc);

// "Decode shall be ignored, except in the case where the image is treated as a mask"

if (!parameters.getBoolean(COSName.IMAGE\_MASK, false))

{

parameters.setItem(COSName.DECODE, null);

}

// override dimensions, see PDFBOX-1735

parameters.setInt(COSName.WIDTH, reader.getWidth(0));

parameters.setInt(COSName.HEIGHT, reader.getHeight(0));

// extract embedded color space

if (!parameters.containsKey(COSName.COLORSPACE))

{

result.setColorSpace(new PDJPXColorSpace(image.getColorModel().getColorSpace()));

}

return image;

}

finally

{

if (iis != null)

{

iis.close();

}

reader.dispose();

}

}

@Override

protected void encode(InputStream input, OutputStream encoded, COSDictionary parameters)

throws IOException

{

throw new UnsupportedOperationException("JPX encoding not implemented");

}

}

***Crypt***   
该Filter是文档级加密，单独对某个stream进行加密。加密方法将在后面的章节中描述。

***总结：***   
本章主要描述了PDF中各类压缩方式，并贴出了对应的加解压缩算法，参考的来自itext，pdfbox，这两个开源的source，能支持绝大部分PDF，但是对于极个别特殊文件，它们加解密算法并不能达到100%正确，请各位注意。

**PDF格式分析（五）文档结构之文件头**

一般情况下，文件头，即，PDF文件的第一行，它用来定义PDF的版本，从而确定该PDF遵循的哪个版本的PDF规范。   
PDF版本是向下兼容的，即高版本的规范，兼容低版本的规范。   
目前我见过的版本有：   
%PDF-1.0   
%PDF-1.1   
%PDF-1.2   
%PDF-1.3   
%PDF-1.4   
%PDF-1.5   
%PDF-1.6   
%PDF-1.7（最常见）   
%PDF-1.8（在标准文档里没有，只是遇到一些客户特殊的文件中会出现，非常少见）   
以上都是常见的版本号，还有一个特殊的版本，如PDF/X、PDF/E 和 PDF/A，PDF/VT等，它们在PDF文件中表现为文件头版本+OutputIntent相结合来标识版本。   
在Catalog字典中，存在Version属性，它的值也是PDF版本号，该版本号如果高与文件头版本号，则采用这里的版本号，否则忽略。这里的Version属性是可选的，通常是在PDF增量写的情况下用到。因为原PDF文件进过编辑后，添加或修改的内容需要遵循的版本比原本更高，而增量写的方式要求原PDF文件内容不修改，所以，通过修改catalog对象中Version，只要增量部分写入catalog对象，就可以改变PDF的版本号。

PDF文件头有些情况下，并不只保存了版本号，还可能在版本号前面加一下用户自定义的内容，用来满足用户的特殊需求。当遇到这种情况时，对文件头的读取算法就要写的灵活一点，不然取值会发生错误，同时还会影响到间接对象读取的问题，这个问题，在后面介绍间接对象读取方法的时候，会介绍。

关于PDF版本，会单独起一章专门讨论。

# PDF版本

PDF规范从1993年到现在，已经有过7个版本，六次版本升级，从最初的pdf1.0，版本到现在的PDF1.7, 每次的版本升级都会加入一些新的特性，PDF参考说明书也是从最初的100多页到现在的1000多页，但是PDF文件格式的主要特性还是没有改变，主要是随著Adobe Acrobat的新版本而更新的。以下为PDF文件的版本号，公布时间及Acrobat 版本号对应列表。   
1993 – PDF 1.0 / Acrobat 1.0   
1994 – PDF 1.1 / Acrobat 2.0 加入了文档加密（40字节），线索树，名字树，链接，设备独立色彩资源。   
1996 – PDF 1.2 / Acrobat 3.0 表单, 半色调屏幕，和其他的一些高级色彩特性, 对中文，日文和韩文的支持   
2000 – PDF 1.3 / Acrobat 4.0 数字签名, 逻辑结构， JavaScript, 嵌入式文件，Masked Images, 平滑阴影, 支持 CID字体的附加色彩。   
2001 – PDF 1.4 / Acrobat 5.0 文件加密 (128 字节), 标签式 PDF, 访问控制，透明，元数据流   
2003 – PDF 1.5 / Acrobat 6.0 文档加密 (公钥), JPEG 2000 压缩, 可选的内容组，附加的注解类型   
2005 – PDF 1.6 / Acrobat 7.0 文档加密 (AES),增加最大文件支持，加入3D支持，额外的注解类型   
2006 – PDF 1.7 / Acrobat 8.0 PDF包，自动识别表单域，管理共享审阅，存档 Microsoft Outlook 、Lotus Notes电子邮件，Microsoft Word格式保存，对 AutoCAD 的支持   
2006 – PDF 1.7 / Acrobat 8.2   
2008 – PDF 1.7, Adobe Extension Level 3 / Acrobat 9.0   
2009 – PDF 1.7, Adobe Extension Level 5 / Acrobat 9.1   
ISO标准ISO 32000-1:2008及Adobe PDF 1.7格式技术上是并存的。Adobe公司宣布以后不会再发布PDF 1.8格式。未来的PDF版本将由ISO技术协会提供。

在转换文件的过程中经常在另存为选项中看到PDF/X、PDF/E和PDF/A，这是PDF转换设置时比较重要的选项，如果需要转换出比较专业的PDF文档，还是需要了解的。那么PDF/X、PDF/E和PDF/A具体有什么区别呢？下面就介绍PDF/X、PDF/E和PDF/A标准。

PDF/X、PDF/E和PDF/A标准是由国际标准化组织(ISO)定义的。

PDF/A   
2005年9月，国际标准化组织（ISO）批准了针对存档电子文档的PDF/A新标准。根据ISO19005-1标准，PDF/A是PDF的一种变型。它“提供了一种工具，使电子文件在长时间之后依然以一种保留其外观的方式重现，而不管该文件是用什么工具和系统创建、储存或制作的。”   
这种保留方式使PDF文件可自我持续。PDF/A通过嵌入在文档自身内部显示该文档的信息（内容、颜色、字体、图像，等等），存储这种自我持续。换句话说，PDF/A文件不要求任何附加的外部信息才能适当地显示。但是，此种格式要存储其自身的持续性，就必须排除某些在诸如电影、音响和透明纸等标准PDF文件中的功能。   
PDF/A可分为：PDF/A-1a，PDF/A-1b，PDF/A-2a，PDF/A-2b，PDF/A-2u，PDF/A-3a，PDF/A-3b，PDF/A-3u。

PDF/X   
PDF/X标准应用于图形内容交换；PDF/E标准应用于工程文档的交互式交换；PDF/A标准应用于电子文档的长期归档，基本上就是屏蔽了一些不适合的功能，如Javascript，音频、视频等等。   
在PDF转换过程中，将对要处理的文件对照指定标准进行检查。如果PDF不符合选定的ISO标准，系统会提示您取消转换或创建不符合标准的文件。   
在印刷出版工作流程中广泛使用的标准有以下几种PDF/X格式：PDF/X-1a、PDF/X-3和PDF/X-4(2008)。在PDF归档中广泛使用的标准为PDF/A和PDF/A（要求较低）。

PDF/X，与PDF/A一样，是PDF的一个子集。PDF/X的目的在于为设计员、绘图员、工程师和图像艺术家提供一种可为任何服务提供者正确打印的电子文件格式。PDF/X使保持完全一致性成为可能，即使文件被人们在多处位置，用不同的机器处理，也一样。这种格式对大多数网络公司的打印就绪文件传输很理想；这里，打印就绪信息的输送者和接收者并无很强的互相关联。除了为打印任务提供坚实的传送格式之外，PDF/X还具备其他好处，包括有一个文件查看器，更佳的压缩效果（文件大小更小），支持专色印刷色彩这种识别打印条件（比如哪个文件已就绪）的技术手段，以及更多。不过，也与PDF/A一样，PDF/X的好处也伴随着一些妥协。比如透明、加密和JBIG2压缩等功能在DF/X中就是被禁止的。

目前，PDF/E的唯一版本是PDF/E-1。如果您想要了解一下PDF/X、PDF/E和PDF/A的详细信息，可以访问ISO和AIIM网站。

PDF/VT特别适用于可变数据印刷的作业交付。名称中的”VT”指的是”可变数据与票据”(Variable and Transactional)。目的是解决所有可变数据印刷在直邮、电话账单和信用卡账单等领域个性化的特殊要求。

后面章节会介绍这些版本的判断方法。