JPEG Metadata Format Specification and Usage Notes

[JPEG Metadata](#_gjdgxs)

[Abbreviated Streams](#_30j0zll)

[Sources of Tables](#_1fob9te)

[Colorspace Transformations and Conventional Markers](#_3znysh7)

[Thumbnail Images](#_2et92p0)

[Progressive Encoding](#_tyjcwt)

[Native Metadata Format Tree Structure and Editing](#_3dy6vkm)

[Image Metadata DTD](#_1t3h5sf)

[Stream Metadata DTD](#_4d34og8)

NOTE: It is important to call dispose() on the JPEG reader and writer objects when they are no longer needed, as they consume significant native resources which are not adequately recovered by garbage collection. Both reader and writer call dispose() in their finalizers, but those finalizers may not be called before the native code has exhausted native memory.

The JPEG writer does not support replacing pixels.

## JPEG Metadata

JPEG metadata consists of the data contained in marker segments in a JPEG stream. The image metadata object returned from a read describes the contents of the marker segments between the SOI marker and the EOI marker for that image. The image metadata object passed into a write determines the contents of the stream between the SOI marker and the EOI marker for that image, subject to the controls in any ImageWriteParam.

Stream metadata is used only for tables-only images found (or to be placed) at the beginning of a stream containing abbreviated images. Tables-only images are not treated as images and do not consume an image index. The stream metadata object returned from a read describes the contents of the marker segments between the SOI marker and the EOI marker for the single tables-only image at the beginning of the stream, if one is present. If no tables-only image is present at the front of the stream, the getStreamMetadata method of ImageReader returns null. If stream metadata is provided to the writer, a single tables-only image containing the tables from the stream metadata object will be written at the beginning of the stream. If the stream metadata object contains no tables, default tables will be written. As the sole purpose of stream metadata is for specifying tables-only images at the front of abbreviated streams, the stream metadata argument is useful only on the ImageWriter.prepareWriteSequence method. It is ignored on all other methods.

The ImageWriter.getDefaultStreamMetadata method returns an object containing the tables from the ImageWriteParam argument, if it is a JPEGImageWriteParam and contains tables. Otherwise, the returned object will contain default tables.

The ImageWriter.getDefaultImageMetadata method returns a metadata object containing no tables if the ImageWriteParam argument contains tables. Otherwise the returned metadata object will contain default visually lossless tables. Of course, only a JPEGImageWriteParam may contain tables.

If ignoreMetadata is set to true when the input is set on the reader, stream metadata will not be available, but image metadata will.

## Abbreviated Streams

Both the reader and the writer retain their tables from one operation to the next, thus permitting the use of abbreviated streams quite naturally, with a few minor restrictions:

1. Abbreviated streams may contain only one tables-only image, which must come first in the stream. Subsequent tables-only images will cause undefined behavior.
2. Abbreviated streams must be read fully and in order. No random access is allowed, in either direction. The same image may be read multiple times. If a call is made with an image index that is not the same as or one greater than the most recent call (or 0 if no calls have been made), then an IllegalArgumentException is thrown.

These restrictions mean that streams may contain abbreviated images interspersed with images containing tables. As required by JPEG, any tables appearing in the stream override previous tables, regardless of the source of the previous tables.

Note that once a tables-only image has been read, it's contents is available as stream metadata from the reader until either another tables-only image is read from another stream or the reader is reset. Changing the input does not reset the stream metadata. This is useful for reading the tables from one file, then changing the input to read an abbreviated stream containing a sequence of images. The tables will be used automatically, and will remain available as "stream" metadata.

Abbreviated streams are written using the sequence methods of ImageWriter. Stream metadata is used to write a tables-only image at the beginning of the stream, and the tables are set up for use, using ImageWriter.prepareWriteSequence. If no stream metadata is supplied to ImageWriter.prepareWriteSequence, then no tables-only image is written. If stream metadata containing no tables is supplied to ImageWriter.prepareWriteSequence, then a tables-only image containing default visually lossless tables is written.

## Sources of Tables

Images are written with tables if tables are present in their metadata objects or without them if no tables are present in their metadata objects. If no metadata object is present then the tables are written. The tables used for compression are taken from one of the following sources, which are consulted in order:

1. If there is an ImageWriteParam and the compression mode is set to EXPLICIT, default tables constructed using the quality setting are used. They are written only if the metadata contains tables or if there is no metadata, but they replace the tables in the metadata.
2. If there is an ImageWriteParam and the compression mode is set to DEFAULT, default visually lossles tables are used. They are written only if the metadata contains tables or if there is no metadata, but they replace the tables in the metadata.
3. Otherwise the compression mode on the ImageWriteParam must be MODE\_COPY\_FROM\_METADATA, in which case the following are used:
   1. the tables in the image metadata, if present
   2. the tables in the stream metadata, if present
   3. the tables in the JPEGImageWriteParam, if present
   4. default visually lossless tables

Tables are written only if they are taken from image metadata.This ordering implements the design intention that tables should be included in JPEGImageWriteParams only as a means of specifying tables when no other source is available, and this can occur only when writing to an abbreviated stream without tables using known non-standard tables for compression.

When reading, tables in a JPEGImageReadParam are consulted only if tables have not been set by any previous read. Tables set from a JPEGImageReadParam are overridden by any tables present in the stream being read.

Note that if no image metadata object is specified for a particular image, a default object is used, which includes default tables.

## Colorspace Transformations and Conventional Markers

Colorspace transformations are controlled by the destination type for both reading and writing of images. When Rasters are read, no colorspace transformation is performed, and any destination type is ignored. A warning is sent to any listeners if a destination type is specified in this case. When Rasters are written, any destination type is used to interpret the bands. This might result in a JFIF or Adobe header being written, or different component ids being written to the frame and scan headers. If values present in a metadata object do not match the destination type, the destination type is used and a warning is sent to any listeners.

When reading, the contents of the stream are interpreted by the usual JPEG conventions, as follows:

* If a JFIF APP0 marker segment is present, the colorspace is known to be either grayscale or YCbCr. If an APP2 marker segment containing an embedded ICC profile is also present, then the YCbCr is converted to RGB according to the formulas given in the JFIF spec, and the ICC profile is assumed to refer to the resulting RGB space.
* If an Adobe APP14 marker segment is present, the colorspace is determined by consulting the transform flag. The transform flag takes one of three values:
  + 2 - The image is encoded as YCCK (implicitly converted from CMYK on encoding).
  + 1 - The image is encoded as YCbCr (implicitly converted from RGB on encoding).
  + 0 - Unknown. 3-channel images are assumed to be RGB, 4-channel images are assumed to be CMYK.
* If neither marker segment is present, the following procedure is followed: Single-channel images are assumed to be grayscale, and 2-channel images are assumed to be grayscale with an alpha channel. For 3- and 4-channel images, the component ids are consulted. If these values are 1-3 for a 3-channel image, then the image is assumed to be YCbCr. If these values are 1-4 for a 4-channel image, then the image is assumed to be YCbCrA. If these values are > 4, they are checked against the ASCII codes for 'R', 'G', 'B', 'A', 'C', 'c'. These can encode the following colorspaces:  
    
  RGB  
  RGBA  
  YCC (as 'Y','C','c'), assumed to be PhotoYCC  
  YCCA (as 'Y','C','c','A'), assumed to be PhotoYCCA  
  Otherwise, 3-channel subsampled images are assumed to be YCbCr, 3-channel non-subsampled images are assumed to be RGB, 4-channel subsampled images are assumed to be YCCK, and 4-channel, non-subsampled images are assumed to be CMYK.
* All other images are declared uninterpretable and an exception is thrown if an attempt is made to read one as a BufferedImage. Such an image may be read only as a Raster. If an image is interpretable but there is no Java ColorSpace available corresponding to the encoded colorspace (e.g. YCbCr), then ImageReader.getRawImageType will return null.

Once an encoded colorspace is determined, then the target colorspace is determined as follows:

* If a destination type is not set, then the following default transformations take place after upsampling: YCbCr (and YCbCrA) images are converted to RGB (and RGBA) using the conversion provided by the underlying IJG library and either the built-in sRGB ColorSpace or a custom RGB ColorSpace object based on an embedded ICC profile is used to create the output ColorModel. PhotoYCC and PhotoYCCA images are not converted. CMYK and YCCK images are currently not supported.
* If a destination image or type is set, it is used as follows: If the IJG library provides an appropriate conversion, it is used. Otherwise the default library conversion is followed by a colorspace conversion in Java.
* Bands are selected AFTER any library colorspace conversion. If a subset of either source or destination bands is used, then the default library conversions are used with no further conversion in Java, regardless of any destination type.
* An exception is thrown if an attempt is made to read an image in an unsupported jpeg colorspace as a BufferedImage (e.g. CMYK). Such images may be read as Rasters. If an image colorspace is unsupported or uninterpretable, then ImageReader.getImageTypes will return an empty Iterator. If a subset of the raw bands are required, a Raster must be obtained first and the bands obtained from that.

For writing, the color transformation to apply is determined as follows:

If a subset of the source bands is to be written, no color conversion is performed. Any destination, if set, must match the number of bands that will be written, and serves as an interpretation of the selected bands, rather than a conversion request. This behavior is identical to that for Rasters. If all the bands are to be written and an image (as opposed to a Raster) is being written, any destination type is ignored and a warning is sent to any listeners.

If a destination type is used and any aspect of the metadata object, if there is one, is not compatible with that type, the destination type is used, the metadata written is modified from that provided, and a warning is sent to listeners. This includes the app0JFIF and app14Adobe nodes. The component ids in the sof and sos nodes are not modified, however, as unless a app0JFIF node is present, any values may be used.

When a full image is written, a destination colorspace will be chosen based on the image contents and the metadata settings, according to the following algorithm:

If no metadata object is specified, then the following defaults apply:

* Grayscale images are written with a JFIF APP0 marker segment. Grayscale images with alpha are written with no special marker. As required by JFIF, the component ids in the frame and scan header is set to 1.
* RGB images are converted to YCbCr, subsampled in the chrominance channels by half both vertically and horizontally, and written with a JFIF APP0 marker segment. If the ColorSpace of the image is based on an ICCProfile (it is an instance of ICC\_ColorSpace, but is not one of the standard built-in ColorSpaces), then that profile is embedded in an APP2 marker segment. As required by JFIF, the component ids in the frame and scan headers are set to 1, 2, and 3.
* RGBA images are converted to YCbCrA, subsampled in the chrominance channels by half both vertically and horizontally, and written without any special marker segments. The component ids in the frame and scan headers are set to 1, 2, 3, and 4.
* PhotoYCC and YCCAimages are subsampled by half in the chrominance channels both vertically and horizontally and written with an Adobe APP14 marker segment and 'Y','C', and 'c' (and 'A' if an alpha channel is present) as component ids in the frame and scan headers.

Default metadata objects for these image types will reflect these settings.

If a metadata object is specified, then the number of channels in the frame and scan headers must always match the number of bands to be written, or an exception is thrown. app0JFIF and app14Adobe nodes may appear in the same metadata object only if the app14Adobe node indicates YCbCr, and the component ids are JFIF compatible (0-2). The various image types are processed in the following ways:

(All multi-channel images are subsampled according to the sampling factors in the frame header node of the metadata object, regardless of color space.)

* Grayscale Images:
  + If an app0JFIF node is present in the metadata object, a JFIF APP0 marker segment is written.
  + If an app14Adobe node is present in the metadata object, it is checked for validity (transform must be UNKNOWN) and written.
  + If neither node is present in the metadata object, no special marker segment is written.
* Grayscale Images with an Alpha Channel:
  + If an app0JFIF node is present in the metadata object, it is ignored and a warning is sent to listeners, as JFIF does not support 2-channel images.
  + If an app14Adobe node is present in the metadata object, it is checked for validity (transform must be UNKNOWN) and written. If transform is not UNKNOWN, a warning is sent to listeners and the correct transform is written.
  + If neither node is present in the metadata object, no special marker segment is written.
* RGB Images:
  + If an app0JFIF node is present in the metadata object, the image is converted to YCbCr and written with a JFIF APP0 marker segment. If the ColorSpace of the image is based on a non-standard ICC Profile, then that profile is embedded in an APP2 marker segment. If the ColorSpace is not based on a non-standard ICC Profile, but an app2ICC node appears in the metadata, then an APP2 marker segment is written with the appropriate standard profile. Note that the profile must specify an RGB color space, as the file must be JFIF compliant.
  + If an app14Adobe node is present in the metadata object, the image is converted according to the color transform setting and written with an Adobe APP14 marker segment. Component ids are written just as they appear in the frame and scan headers. The color transform must be either YCbCr or UNKNOWN. If it is UNKNOWN, the image is not color converted.
  + If neither node is present, the component ids in the frame header are consulted. If these indicate a colorspace as described above, then the image is converted to that colorspace if possible. If the component ids do not indicate a colorspace, then the sampling factors are consulted. If the image is to be subsampled, it is converted to YCbCr first. If the image is not to be subsampled, then no conversion is applied. No special marker segmentss are written.
* RGBA images:
  + If an app0JFIF node is present in the metadata object, it is ignored and a warning is sent to listeners, as JFIF does not support 4-channel images.
  + If an app14Adobe node is present in the metadata object, the image is written with an Adobe APP14 marker segment. No colorspace conversion is performed. Component ids are written just as they appear in the frame and scan headers. The color transform must be UNKNOWN. If it is not, a warning is sent to listeners.
  + If no app14Adobe node is present, the component ids in the frame header are consulted. If these indicate a colorspace as described above, then the image is converted to that colorspace if possible. If the component ids do not indicate a colorspace, then the sampling factors are consulted. If the image is to be subsampled, it is converted to YCbCrA. If the image is not to be subsampled, then no conversion is applied. No special marker segments are written.
* PhotoYCC Images:
  + If an app0JFIF node is present in the metadata object, the image is converted to sRGB, and then to YCbCr during encoding, and a JFIF APP0 marker segment is written.
  + If an app14Adobe node is present in the metadata object, no conversion is applied, and an Adobe APP14 marker segment is written. The color transform must be YCC. If it is not, a warning is sent to listeners.
  + If neither node is present in the metadata object, no conversion is applied, and no special marker segment is written.
* PhotoYCCA Images:
  + If an app0JFIF node is present in the metadata object, it is ignored and a warning is sent to listeners, as JFIF does not support 4-channel images.
  + If an app14Adobe node is present in the metadata object, no conversion is applied, and an Adobe APP14 marker segment is written. The color transform must be UNKNOWN. If it is not, a warning is sent to listeners.
  + If neither node is present in the metadata object, no conversion is applied, and no special marker segment is written.

## Thumbnail Images

Thumbnails are supported by the use of JFIF and JFIF extension marker segments. Thumbnails provided on the write methods determine the thumbnails that will be included. app0JFIF and app0JFXX nodes present in the metadata do not contain any thumbnail pixel data. However, the kinds of thumbnails written depend on the contents of the metadata object, as follows. Any thumbnail which is to be written as an indexed or RGB image and which is larger than 255 by 255 will be clipped, not scaled, to 255 by 255. Thumbnails written as JPEG images may be any size. A warning is sent to any listeners whenever a thumbnail is clipped.

* If there is a single thumbnail, it is processed as follows:
  + If the thumbnail image is an RGB palette image, it is processed as follows:
    - If no app0JFXX node is present in the metadata, or the first app0JFXX node present in the metadata contains a JFIFthumbPalette element, a palette thumbnail is written in a JFXX APP0 marker segment.
    - If the first app0JFXX node present in the metadata contains another thumbnail form (RGB or JPEG), the palette image is expanded to RGB and the indicated thumbnail form is written.
  + If the thumbnail image is an RGB image, it is processed as follows:
    - If no app0JFXX node is present in the metadata, the thumbnail is written as part of the JFIF APP0 marker segment.
    - If the first app0JFXX node present in the metadata contains a JFIFthumbRGB element, an RGB thumbnail is written in a JFXX APP0 marker segment.
    - If the first app0JFXX node present in the metadata contains a JFIFthumbJPEG element, a JPEG thumbnail is written in a JFXX APP0 marker segment.
    - If the first app0JFXX node present in the metadata contains a JFIFthumbPalette element, an RGB thumbnail is written in a JFXX APP0 marker segment and a warning is sent to any listeners.
  + If the thumbnail image is a grayscale image, it is processed as follows:
    - If no app0JFXX node is present in the metadata, the thumbnail is expanded to RGB and written as part of the JFIF APP0 marker segment.
    - If the first app0JFXX node present in the metadata contains a JFIFthumbRGB element, the thumbnail is expanded to RGB and written in a separate JFXX RGB marker segment.
    - If the first app0JFXX node present in the metadata contains a JFIFthumbJPEG element, a JPEG thumbnail is written in a JFXX APP0 marker segment.
    - If the first app0JFXX node present in the metadata contains a JFIFthumbPalette element, a JPEG thumbnail is written in a JFXX APP0 marker segment and a warning is sent to any listeners.
  + Any other thumbnail image types are ignored and a warning is sent to any listeners.
* If there are multiple thumbnails, each one is processed as above, except that no thumbnail is placed in the JFIF APP0 segment, and the app0JFXX node consulted for each thumbnail is the app0JFXX node from the metadata that occurs in the same sequence as the thumbnail. I.e. the first app0JFXX node applies to the first thumbnail, the second node to the second thumbnail, and so on. If there are fewer app0JFXX nodes in the metadata than thumbnails, then those thumbnails are considered to have no matching app0JFXX node. An RGB thumbnail with no matching app0JFXX node is written in a JFXX APP0 marker segment. A grayscale thumbnail with no matching app0JFXX node is written as a JPEG image to a JFXX APP0 marker segment.

Note that as the only mechanism for storing thumbnails is via the JFIF or JFIF extension marker segments, only grayscale or RGB images may have thumbnails. If thumbnails are present when writing any other type of image, the thumbnails are ignored and a warning is sent to any warning listeners.

## Progressive Encoding

Progressive encoding must be enabled on the ImageWriteParam passed in to a write operation, or the image will be written sequentially, regardless of the scan headers included in the metadata object. If progressive encoding is enabled and set to copy from metadata, then the sequence of scan headers from the metadata is used to write the image. If progressive encoding is enabled and set to use a default, then the scans in the metadata are ignored and a default set of scans is used. Progressive encoding always forces optimized Huffman tables to be used. Any Huffman tables present in the metadata will be ignored, and a warning will be sent to any warning listeners. If Huffman-table optimization is requested on the ImageWriteParam, all Huffman tables in the metadata or in the ImageWriteParam itself are ignored, and a warning will be sent to any warning listeners if any such tables are present.

## Native Metadata Format Tree Structure and Editing

The DTDs below describe just the trees of metadata objects actually returned by the IIOMetadata object. They do not include nodes corresponding to SOI, EOI, or RST markers, as these parsing delimiters do not carry any meaningful metadata.

The first node is always a JPEGvariety node. In the javax\_imageio\_jpeg\_image\_1.0 version of the JPEG metadata format, this node may have one child, an app0JFIF node, indicating that the JPEG stream contains a JFIF marker segment and related data, or no children, indicating that the stream contains no JFIF marker. In future versions of the JPEG metadata format, other varieties of JPEG metadata may be supported (e.g. Exif) by defining other types of nodes which may appear as a child of the JPEGvariety node.

(Note that an application wishing to interpret Exif metadata given a metadata tree structure in the javax\_imageio\_jpeg\_image\_1.0 format must check for an unknown marker segment with a tag indicating an APP1 marker and containing data identifying it as an Exif marker segment. Then it may use application-specific code to interpret the data in the marker segment. If such an application were to encounter a metadata tree formatted according to a future version of the JPEG metadata format, the Exif marker segment might not be unknown in that format - it might be structured as a child node of the JPEGvariety node. Thus, it is important for an application to specify which version to use by passing the string identifying the version to the method/constructor used to obtain an IIOMetadata object.)

On reading, JFXX and app2ICC nodes occur as children of an app0JFIF node. This is true regardless of where the JFXX APP0 and APP2 marker segments actually occur in the stream. The ordering of nodes within the markerSequence node corresponds to the ordering of marker segments found in the JPEG stream.

On writing, any JFXX and app2ICC nodes must occur as children of an app0JFIF node, itself a child of a JPEGvariety node, which must always be the first node. (If the stream is not to be JFIF compliant, no app0JFIF node should be provided, and the JPEGvariety node should have no children.) Any JFIF APP0, JFXX APP0, and APP2 marker segments are written first, followed by all Adobe APP14, APPn, COM and unknown segments in the order in which their corresponding nodes appear in the markerSequence node, followed by DQT (and DHT for non-progressive writes) marker segments, followed by the SOF and SOS marker segments. For progressive writes using metadata to control progression, the SOS segments are used in the order in which their corresponding nodes occur in the markerSequence node.

The reset, mergeTree and setFromTree operations have the following semantics for the JPEG plug-in metadata object:

reset - A call to reset will restore the metadata object to the same state it had immediately after creation, whether this came about from reading a stream or by obtaining a default object from the ImageWriter. This is true regardless of how many times the metadata object has been modified since creation.

mergeTree - Native Format

The mergeTree operation accepts valid trees conforming to the DTD below, and merges the nodes using the following ordering rules. In all cases, only data present in the new node is changed in a corresponding existing node, if any. This means that nodes cannot be removed using mergeTree. To remove nodes, use setFromTree. The tree must consist of IIOMetadataNodes.

* app0JFIF
  + If an app0JFIF node already exists, the contents of the new one modify the existing one.
  + If there is no such node, a new one is created and inserted in the appropriate position.
* dqt
  + If there already exist dqt nodes in the sequence, then each table in the node replaces the first table, in any dqt node, with the same table id.
  + If none of the existing dqt nodes contain a table with the same id, then the table is added to the last existing dqt node.
  + If there are no dqt nodes, then a new one is created and added as follows:
    - If there are dht nodes, the new dqt node is inserted before the first one.
    - If there are no dht nodes, the new dqt node is inserted before an sof node, if there is one.
    - If there is no sof node, the new dqt node is inserted before the first sos node, if there is one.
    - If there is no sos node, the new dqt node is added to the end of the sequence.
* dht
  + If there already exist dht nodes in the sequence, then each table in the node replaces the first table, in any dht node, with the same table class and table id.
  + If none of the existing dht nodes contain a table with the same class and id, then the table is added to the last existing dht node.
  + If there are no dht nodes, then a new one is created and added as follows:
    - If there are dqt nodes, the new dht node is inserted immediately following the last dqt node.
    - If there are no dqt nodes, the new dht node is inserted before an sof node, if there is one.
    - If there is no sof node, the new dht node is inserted before the first sos node, if there is one.
    - If there is no sos node, the new dht node is added to the end of the sequence.
* dri
  + If there already exists a dri node, the restart interval value is updated.
  + If there is no dri node, then a new one is created and added as follows:
    - If there is an sof node, the new dri node is inserted before it.
    - If there is no sof node, the new dri node is inserted before the first sos node, if there is one.
    - If there is no sos node, the new dri node is added to the end of the sequence.
* com  
  A new com node is created and inserted as follows:
  + If there already exist com nodes, the new one is inserted after the last one.
  + If there are no com nodes, the new com node is inserted after the app14Adobe node, if there is one.
  + If there is no app14Adobe node, the new com node is inserted at the beginning of the sequence.
* app14Adobe
  + If there already exists an app14Adobe node, then its attributes are updated from the node.
  + If there is no app14Adobe node, then a new one is created and added as follows:
    - The new app14Adobe node is inserted after the last unknown node, if there are any.
    - If there are no unknown nodes, the new app14Adobe node is inserted at the beginning of the sequence.
* unknown  
  A new unknown node is created and added to the sequence as follows:
  + If there already exist unknown marker nodes, the new one is inserted after the last one.
  + If there are no unknown nodes, the new unknown node is inserted before the app14Adobe node, if there is one.
  + If there is no app14Adobe node, the new unknown node is inserted at the beginning of the sequence.
* sof
  + If there already exists an sof node in the sequence, then its values are updated from the node.
  + If there is no sof node, then a new one is created and added as follows:
    - If there are any sos nodes, the new sof node is inserted before the first one.
    - If there is no sos node, the new sof node is added to the end of the sequence.
* sos
  + If there already exists a single sos node, then the values are updated from the node.
  + If there are more than one existing sos nodes, then an IIOInvalidTreeException is thrown, as sos nodes cannot be merged into a set of progressive scans.
  + If there are no sos nodes, a new one is created and added to the end of the sequence.

mergeTree - Standard Format

The mergeTree operation, when given a tree in the standard format, will modify the native tree in the following ways:

* Chroma - The ColorSpaceType subnode of a Chroma node may change the target colorspace of the compressed image. The selection of a new colorspace can cause a number of changes, in keeping with the algorithms described above: app0JFIF and app14Adobe nodes may be added or removed, subsampling may be added or removed, component ids may be changed, and sof and sos nodes will be updated accordingly. If necessary, additional quantization and huffman tables are added. In the case of quantization tables, the default will be scaled to match the quality level of any existing tables. No tables are added to metadata that does not already contain tables. If the existing metadata specifies progressive encoding, then the number of channels must not change. Any Transparency node is also taken into account, as an explicit value of none for the Alpha subnode can cause the removal of an alpha channel, and anything other than none can cause the addition of an alpha channel.
* Dimension - A PixelAspectRatio specification can cause the contents of an app0JFIF node to change, if there is one present, or the addition of an app0JFIF node containing appropriate values, if there can be one. An appropriate pair of integers is computed from the floating-point ratio for inclusion in the node.
* Text - Each uncompressed text item is converted to a com node and inserted according to the rules above for merging com nodes.

setFromTree - Native Format

The setFromTree operation, when given a tree in the native format described below, will simply replace the existing tree in its entirety with the new one. The tree must consist of IIOMetadataNodes.

setFromTree - Standard Format

The setFromTree operation, when given a tree in the standard format, performs a reset followed by a merge of the new tree.

## Image Metadata DTD

<!DOCTYPE "javax\_imageio\_jpeg\_image\_1.0" [  
  
 <!ELEMENT "javax\_imageio\_jpeg\_image\_1.0" (JPEGvariety, markerSequence)>  
  
 <!ELEMENT "JPEGvariety" (app0JFIF)>  
 <!-- A node grouping all marker segments specific to the variety of  
 stream being read/written (e.g. JFIF) - may be empty -->   
  
 <!ELEMENT "app0JFIF" (JFXX?, app2ICC?)>  
 <!ATTLIST "app0JFIF" "majorVersion" #CDATA "1">  
 <!-- The major JFIF version number -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "app0JFIF" "minorVersion" #CDATA "2">  
 <!-- The minor JFIF version number -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "app0JFIF" "resUnits" ("0" | "1" | "2") "0">  
 <!-- The resolution units for Xdensisty and Ydensity (0 = no   
 units, just aspect ratio; 1 = dots/inch; 2 = dots/cm) -->   
 <!ATTLIST "app0JFIF" "Xdensity" #CDATA "1">  
 <!-- The horizontal density or aspect ratio numerator -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 1 (inclusive) -->  
 <!-- Max value: 65535 (inclusive) -->  
 <!ATTLIST "app0JFIF" "Ydensity" #CDATA "1">  
 <!-- The vertical density or aspect ratio denominator -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 1 (inclusive) -->  
 <!-- Max value: 65535 (inclusive) -->  
 <!ATTLIST "app0JFIF" "thumbWidth" #CDATA "0">  
 <!-- The width of the thumbnail, or 0 if there isn't one -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "app0JFIF" "thumbHeight" #CDATA "0">  
 <!-- The height of the thumbnail, or 0 if there isn't one -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
  
 <!ELEMENT "JFXX" (app0JFXX)\*>  
 <!-- Min children: 1 -->  
  
 <!ELEMENT "app0JFXX" (JFIFthumbJPEG | JFIFthumbPalette |   
 JFIFthumbRGB)>  
 <!-- A JFIF extension marker segment -->   
 <!ATTLIST "app0JFXX" "extensionCode" ("16" | "17" | "19")  
 #IMPLIED>  
 <!-- The JFXX extension code identifying thumbnail type: (16 =   
 JPEG, 17 = indexed, 19 = RGB -->   
  
 <!ELEMENT "JFIFthumbJPEG" (markerSequence?)>  
 <!-- A JFIF thumbnail in JPEG format (no JFIF segments   
 permitted) -->   
  
 <!ELEMENT "JFIFthumbPalette" EMPTY>  
 <!-- A JFIF thumbnail as an RGB indexed image -->   
 <!ATTLIST "JFIFthumbPalette" "thumbWidth" #CDATA #IMPLIED>  
 <!-- The width of the thumbnail -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "JFIFthumbPalette" "thumbHeight" #CDATA #IMPLIED>  
 <!-- The height of the thumbnail -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
  
 <!ELEMENT "JFIFthumbRGB" EMPTY>  
 <!-- A JFIF thumbnail as an RGB image -->   
 <!ATTLIST "JFIFthumbRGB" "thumbWidth" #CDATA #IMPLIED>  
 <!-- The width of the thumbnail -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "JFIFthumbRGB" "thumbHeight" #CDATA #IMPLIED>  
 <!-- The height of the thumbnail -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
  
 <!ELEMENT "app2ICC" EMPTY>  
 <!-- An ICC profile APP2 marker segment -->   
 <!-- Optional User object: java.awt.color.ICC\_Profile -->  
  
 <!ELEMENT "markerSequence" (dqt | dht | dri | com | unknown |   
 app14Adobe | sof | sos)\*>  
 <!-- A node grouping all non-jfif marker segments -->   
  
 <!ELEMENT "dqt" (dqtable)\*>  
 <!-- A Define Quantization Table(s) marker segment -->   
 <!-- Min children: 1 -->  
 <!-- Max children: 4 -->  
  
 <!ELEMENT "dqtable" EMPTY>  
 <!-- A single quantization table -->   
 <!-- User object: javax.imageio.plugins.jpeg.JPEGQTable -->  
 <!ATTLIST "dqtable" "elementPrecision" #CDATA "0">  
 <!-- The number of bits in each table element (0 = 8, 1 = 16)   
 -->   
 <!-- Data type: Integer -->  
 <!ATTLIST "dqtable" "qtableId" ("0" | "1" | "2" | "3") #REQUIRED>  
  
 <!ELEMENT "dht" (dhtable)\*>  
 <!-- A Define Huffman Table(s) marker segment -->   
 <!-- Min children: 1 -->  
 <!-- Max children: 4 -->  
  
 <!ELEMENT "dhtable" EMPTY>  
 <!-- A single Huffman table -->   
 <!-- User object: javax.imageio.plugins.jpeg.JPEGHuffmanTable -->  
 <!ATTLIST "dhtable" "class" ("0" | "1") #REQUIRED>  
 <!-- Indicates whether this is a DC (0) or an AC (1) table -->   
 <!ATTLIST "dhtable" "htableId" ("0" | "1" | "2" | "3") #REQUIRED>  
 <!-- The table id -->   
  
 <!ELEMENT "dri" EMPTY>  
 <!-- A Define Restart Interval marker segment -->   
 <!ATTLIST "dri" "interval" #CDATA #REQUIRED>  
 <!-- The restart interval in MCUs -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 65535 (inclusive) -->  
  
 <!ELEMENT "com" EMPTY>  
 <!-- A Comment marker segment. The user object contains the actual   
 bytes. -->   
 <!-- User object: array of [B -->  
 <!-- Min length: 1 -->  
 <!-- Max length: 65533 -->  
 <!ATTLIST "com" "comment" #CDATA #IMPLIED>  
 <!-- The comment as a string (used only if user object is null)   
 -->   
 <!-- Data type: String -->  
  
 <!ELEMENT "unknown" EMPTY>  
 <!-- An unrecognized marker segment. The user object contains the   
 data not including length. -->   
 <!-- User object: array of [B -->  
 <!-- Min length: 1 -->  
 <!-- Max length: 65533 -->  
 <!ATTLIST "unknown" "MarkerTag" #CDATA #REQUIRED>  
 <!-- The tag identifying this marker segment -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
  
 <!ELEMENT "app14Adobe" EMPTY>  
 <!-- An Adobe APP14 marker segment -->   
 <!ATTLIST "app14Adobe" "version" #CDATA "100">  
 <!-- The version of Adobe APP14 marker segment -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 100 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "app14Adobe" "flags0" #CDATA "0">  
 <!-- The flags0 variable of an APP14 marker segment -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 65535 (inclusive) -->  
 <!ATTLIST "app14Adobe" "flags1" #CDATA "0">  
 <!-- The flags1 variable of an APP14 marker segment -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 65535 (inclusive) -->  
 <!ATTLIST "app14Adobe" "transform" ("0" | "1" | "2") #REQUIRED>  
 <!-- The color transform applied to the image (0 = Unknown, 1 =   
 YCbCr, 2 = YCCK) -->   
  
 <!ELEMENT "sof" (componentSpec)\*>  
 <!-- A Start Of Frame marker segment -->   
 <!-- Min children: 1 -->  
 <!-- Max children: 4 -->  
 <!ATTLIST "sof" "process" ("0" | "1" | "2") #IMPLIED>  
 <!-- The JPEG process (0 = Baseline sequential, 1 = Extended   
 sequential, 2 = Progressive) -->   
 <!ATTLIST "sof" "samplePrecision" #CDATA "8">  
 <!-- The number of bits per sample -->   
 <!-- Data type: Integer -->  
 <!ATTLIST "sof" "numLines" #CDATA #IMPLIED>  
 <!-- The number of lines in the image -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 65535 (inclusive) -->  
 <!ATTLIST "sof" "samplesPerLine" #CDATA #IMPLIED>  
 <!-- The number of samples per line -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 65535 (inclusive) -->  
 <!ATTLIST "sof" "numFrameComponents" ("1" | "2" | "3" | "4")  
 #IMPLIED>  
 <!-- The number of components in the image -->   
  
 <!ELEMENT "componentSpec" EMPTY>  
 <!-- A component specification for a frame -->   
 <!ATTLIST "componentSpec" "componentId" #CDATA #REQUIRED>  
 <!-- The id for this component -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "componentSpec" "HsamplingFactor" #CDATA #REQUIRED>  
 <!-- The horizontal sampling factor for this component -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 1 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "componentSpec" "VsamplingFactor" #CDATA #REQUIRED>  
 <!-- The vertical sampling factor for this component -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 1 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "componentSpec" "QtableSelector" ("0" | "1" | "2" |   
 "3") #REQUIRED>  
 <!-- The quantization table to use for this component -->   
  
 <!ELEMENT "sos" (scanComponentSpec)\*>  
 <!-- A Start Of Scan marker segment -->   
 <!-- Min children: 1 -->  
 <!-- Max children: 4 -->  
 <!ATTLIST "sos" "numScanComponents" ("1" | "2" | "3" | "4")  
 #REQUIRED>  
 <!-- The number of components in the scan -->   
 <!ATTLIST "sos" "startSpectralSelection" #CDATA "0">  
 <!-- The first spectral band included in this scan -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 63 (inclusive) -->  
 <!ATTLIST "sos" "endSpectralSelection" #CDATA "63">  
 <!-- The last spectral band included in this scan -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 63 (inclusive) -->  
 <!ATTLIST "sos" "approxHigh" #CDATA "0">  
 <!-- The highest bit position included in this scan -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 15 (inclusive) -->  
 <!ATTLIST "sos" "approxLow" #CDATA "0">  
 <!-- The lowest bit position included in this scan -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 15 (inclusive) -->  
  
 <!ELEMENT "scanComponentSpec" EMPTY>  
 <!-- A component specification for a scan -->   
 <!ATTLIST "scanComponentSpec" "componentSelector" #CDATA  
 #REQUIRED>  
 <!-- The id of this component -->   
 <!-- Data type: Integer -->  
 <!-- Min value: 0 (inclusive) -->  
 <!-- Max value: 255 (inclusive) -->  
 <!ATTLIST "scanComponentSpec" "dcHuffTable" ("0" | "1" | "2" |   
 "3") #REQUIRED>  
 <!-- The huffman table to use for encoding DC coefficients -->   
 <!ATTLIST "scanComponentSpec" "acHuffTable" ("0" | "1" | "2" |   
 "3") #REQUIRED>  
 <!-- The huffman table to use for encoding AC coefficients -->   
]>

## Stream Metadata DTD

<!DOCTYPE "javax\_imageio\_jpeg\_stream\_1.0" [  
 <!ELEMENT "javax\_imageio\_jpeg\_stream\_1.0" (dqt |  
 dht |   
 dri |   
 com |   
 unknown)\*>  
   
 <!-- All elements are as defined above for image metadata -->  
]>