Java™ Image I/O API Guide

Phil Race, Dan Rice, Raúl Vera

Beta Draft

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

April 2001

Send comments about this document to: jsr015-comments@eng.sun.com

Contents

CHAPTER 1

Introduction

1.1 The Java™ Image I/O API

The Java Image I/O API provides a pluggable architecture for working with images stored in files and accessed across the network. It offers substantially more flexibility and power than the previously available APIs for loading and saving images.

The API provides a framework for the addition of format-specific plug-ins. Plug-ins for several common formats will be included with the standard Java SDK; other plug-ins will be available from Sun and from third parties. A voting process has been established where Java developers may indicate their interest in particular formats. See:

`http://developer.java.sun.com/developer/bugParade/bugs/4339476.html`

1.2 About This Document

This document is a brief guide to the API and its goals, including some simple programming examples. However, this document is not intended to be a complete guide to the API. The definitive guide to the specification is the automatically-generated ("javadoc") API documentation. In all cases, the automatically-generated documentation takes precedence over the descriptive materal in this document.

1.3 Platforms

The API will be a standard part of the "Merlin" release of the Java™ Platform Standard Edition (J2SE version 1.4). Sun will provide an implementation of J2SE for the Solaris, Linux, and Microsoft Windows 95/98/NT/ME (Win32) platforms. The availibility of a particular J2SE release on other platforms is determined by the platform vendor.

1.4 Target Audience

The API is intended to be used in a variety of contexts. Client-side applications may use it to load image data from local disk files or across the network. Server-side applications may use it to generate images dynamically in response to client requests, for example by using the Servlet API. The Image I/O API may be used in conjunction with other APIs such as Java Advanced Imaging, Java3D, and the Java Media Framework in order to build sophisticated multimedia applications.

1.5 Acknowledgements

The authors would like to thank the following members of the expert group and others who provided valuable assistance with the API design:

Bharti Agrawal, Troy Chinen, Shoji Hara, Fuji FilmSoft

David Clunie, Quintiles Intelligent Imaging

Bob Deen, NASA Jet Propulsion Laboratories

Thomas DeWeese, Eastman Kodak Corp.

Steve Levoe, NASA Jet Propulsion Laboratories

Todd Rowell, Oracle

Jeannette Hung and Ihtisham Kabir provided excellent front-line management and made it possible for the API and its implementation to be devloped with a minimum of red tape. Aastha Bhardwaj, Brian Burkhalter, Jerry Evans, Ivan Wong, and John Zimmerman provided invaluable technical assistance.

CHAPTER 2

Goals

2.1 API Goals

The design of the Image I/O API was influenced by the desire to support a number of primary goals. Each goal provides the justification for a particular set of API features. These goals may be divided into those motivated mainly by the needs of client-side applications, and those motivated by the needs of server applications. Naturally, these primary goals represent only a fraction of the capabilities of the API. They are listed here in order to provide a sense of the motivations that went into the design of the API.

As a general rule of thumb, whenever a tradeoff between ease of use for application developers and plug-in developers was identified, the application developer was given a higher priority. Since relatively few users of the API are expected to write plug-ins, and even those developers who do write plug-ins will typically write no more than a handful in order to support formats of immediate use to them, it makes sense to push complexity into the plug-ins rather than into applications.

To ameliorate the complexity of plug-in development, some utility methods and implementation classes are provided that perform common functions. However, it is not possible to provide implementation classes to cover every situation, and providing an excessive number of utility classes and methods would increase the footprint for all API users even if they did not use non-standard plug-ins. We expect that plug-in developers will be able to incorporate source code from existing plug-ins and sample code into their plug-ins. While this might lead to some redundancy when using multiple, independently developed plug-ins, most users will probably use only a small number of plug-ins at a time in any case. Thus, we believe the cost of having to load code that might theoretically be shared but actually is not will tend to outweight the cost of loading some redundant code when many plug-ins are used.

2.1.1 Client-Side Application Goals

Pluggability

An application written to use the Image I/O API should be able to automatically take advantage of new plug-ins without any rewriting or recompilation. This requires that plug-ins follow, as much as possible, a set of format-neutral interfaces. However, every image format has unique properties and capabilities, which plug-ins must be able to expose to applications. This is accomplished by allowing plug-ins to extend a number of interfaces within the API. Applications that are unaware of the plug-in specific extensions may continue to make use of the normal plug-in capabilities, while aware applications may use the extended interfaces.

Both generic and plug-in specific access to metadata (non-image data) is supported by allowing plug-ins to provide access to metadata in multiple formats. These may include plug-in specific formats, a plug-in neutral format defined by the API, and industry standard formats.

Automatic and Manual Plug-in Installation and Selection

For simple applications that wish to load images without user intervention, it is important to allow a reader plug-in to be selected automatically based on file contents. However, it is desirable to avoid the loading and instantiation of complex classes unless they are needed. A plug-in must be able to determine whether it can handle a particular image without loading all of its code. In order to meet this goal, plug-ins are instantiated via a service provider interface mechanism that allows simple queries to be performed without loading the main body of plug-in code.

All of the bytecode (.class) files associated with a plug-in may be combined into a JAR file, which may be installed permanently into a Java™ Runtime Environment, or loaded dynamically using the application `CLASSPATH` mechanism.

Manual Plug-in Selection

Although automatic plug-in selection is convenient for many applications, it must be possible for more sophisticated applications to exercise greater control over the plug-in selection process. This is accomplished by means of a run-time registry of plug-ins that may be queried and manipulated by an application.

Network, Disk-based, and Direct I/O

Increasingly, applications must deal with both disk-based and network-based data. In many cases, even disk-based data must be dealt with in the form of an `InputStream` due to the needs of other APIs. The Image I/O API offers a set of interfaces that allow data from `Files`, `InputStreams`, or other sources to be handled in a unified manner while preserving the ability to seek backwards and forwards. The API is future-proofed to allow new I/O interfaces, including direct interfaces to image acquisition and output devices, to be used without rewriting application code.

Access to Metadata

The metadata stored alongside an image may be as important as the image data. The JavaTM Image I/O API provides thorough yet flexible access to metadata. Since metadata may take many forms, and contain very specialized information, it is difficult to provide direct access to such metadata in a general-purpose API. Instead, the API requires plug-ins to cast their metadata into the form of an XML Document structure, possibly enhanced with `Object` references in addition to textual data. Once this has been done, the metadata may be accessed and edited using standard XML DOM (Document Object Model) interfaces. The syntax of the `Documents` will vary from plug-in to plug-in, but the structure may be traversed, displayed, and edited without specific knowledge of the plug-in being used.

Support for Advanced Applications

In order to support advanced applications, the API must provide access to features such as “thumbnail” images, multiple images stored in a file, muli-resolution imagery, and tiled imagery. It must provide the capability to decode portions of a larger image, and to perform subsampling during decoding in order to allow panning over very large images. When writing images, it must be possible to write image data piece-by-piece, without storing the entire image in memory at once.

2.1.2 Server Use Cases

Dynamic image generation

Modern web servers typically generate a large portion of their content dynamically. The Java Servlet and Java Server Pages (JSP) APIs provide portable ways to generate HTML pages in response to requests from web browsers. Just as unique, custom HTML may be generated for each user, it is possible to customize image content.

In many cases, image content must be generated dynamically. For example, consider supplying stock price charts. While it might be possible to generate and store a limited set of charts for each listed stock in advance, if the user is allowed to add customizations such as the time interval over which the price should be displayed, or a set of indices or other stocks against which the stock price should be compared, the number of possible images grows exponentially. Only a dynamic approach can possibly allow such customization.

Image Customization

Web images are typically one-size-fits-all, with the same image data being delivered regardless of the display capabilities of the receiver. The increasing use of wireless and handheld devices will require images that are customized for their limited bandwidth and display capabilities. Desktop computers have seen increases in their display resolution, causing many web images to appear too small. The lack of scalable images also causes problems for visually impaired users. Server-side image customization can be used to provide optimal images for all users, with the image resolution and color characteristics being chosen based on user preferences.

2.2 Non-Goals

Thread Safety

A particular instance of an `ImageReader` or `ImageWriter` is not required to support re-entrant (simultaneous) calls to its methods (with the exception of the `abort` method that requests current read or writes to halt). However, it must be possible for multiple instances of the same plug-in class to operate simultaneously. For the sake of brevity, we will only discuss reader plug-ins below.

Supporting full re-entrancy would require that the reader bundle all of its state information (e.g., the current input source) into a separate state object, which would allow methods in progress to continue to work with the settings that were in effect at the time they began, while allowing a separate thread to modify the state to be used by the next operation.

Rather than forcing each `ImageReader` to keep track of its state in this way, it is simpler to require the application to instantiate multiple instances of the same `ImageReader` class if it wishes to perform multithreaded processing. This means that the state of an `ImageReader` must be maintained using non-static instance variables only, which should not be a burden for plug-in developers.

CHAPTER 3

Writing Image I/O Applications

3.1 Reading and Writing Images

The javax.imageio.ImageIO class provides a set of static convenience methods that perform most simple Image I/O operations.

Reading an image that is in a standard format (GIF, PNG, or JPEG) is simple:

```
File f = new File("c:\images\myimage.gif");
BufferedImage bi = ImageIO.read(f);
```

The format of the image will be auto-detected by the API based on the contents of the file. Most image files contain a "magic number" in their first few bytes that identifies the file format. For formats that do not have a magic number, auto-detection may fail and somewhat more sophisticated application code will be needed.

Additional formats may be handled by installing JAR files containing plug-ins; the details are described in the next chapter. Once a plug-in has been installed, a new format will be understood automatically without any changes to the application code.

The set of formats available for reading can be obtained by calling ImageIO.getReaderFormatNames. This will return an array of Strings containing the names of the available formats, such as "gif" and "png."

Writing an image in a supported format is equally simple:

```
BufferedImage bi;
File f = new File("c:\images\myimage.png");
ImageIO.write(im, "png", f);
```

The list of supported formats may be obtained by calling `ImageIO.getWriterFormatNames`.

3.2 Going Further

The methods described above are sufficient for simple applications. However, the Image I/O API offers much more power for writing sophisticated applications. In order to make use of the advanced features of the API, applications will work directly with the `ImageReader` and `ImageWriter` classes that are part of the API.

3.3 ImageReader

Rather than using the `ImageIO` class to perform the entire decoding operation, an application may use the `ImageIO` class to obtain an `ImageReader` object that may be used to perform the read:

```
Iterator readers = ImageIO.getImageReadersByFormatName("gif");
ImageReader reader = (ImageReader)readers.next();
```

Readers may also be retrieved based on file contents, file suffix, or MIME type. The mechanism for locating and instantiating the reader makes use of the `javax.imageio.spi.ImageReaderSpi` class, which allows information about a reader plug-in to be retrieved without actually instantiating the plug-in. The notion of "service provider interfaces" (SPIs) is described in detail in the following chapter.

Once a reader has been obtained, it must be supplied with an input source. Most readers are able to read from an `ImageInputStream`, which is a special input source that is defined by the Image I/O API. A special input source is used in order to make it simple for reader and writers to work with both file and streaming I/O.

Obtaining an `ImageInputStream` is straightforward. Given an input source in the form of a `File` or `InputStream`, an `ImageInputStream` is produced by the call:

```
Object source; // File or InputStream
ImageInputStream iis = ImageIO.createImageInputStream(source);
```

Once a source has been obtained, it may be attached to the reader by calling

```
reader.setInput(iis, true);
```

The second parameter should be set to false if the source file contains multiple images and the application is not guaranteed to read them in order. For file formats that allow only a single image to be stored in a file, it is always legal to pass in a value of true.

Once the reader has its input source set, we can use it to obtain information about the image without necessarily causing image data to be read into memory. For example, calling `reader.getImageWidth(0)` allows us to obtain the width of the first image stored in the file. A well-written plug-in will attempt to decode only as much of the file as is necessary to determine the image width, without reading any pixels.

To read the image, the application may call `reader.read(imageIndex)`, where `imageIndex` is the index of the image within the file. This produces the same result as if it had called `ImageIO.read`, as shown above.

3.3.1 ImageReadParam

More control may be obtained by supplying the `read` method with an additional parameter of type `ImageReadParam`. An `ImageReadParam` allows the application to specify a destination image in which the decoded image data should be stored, allowing better control over memory use. It also allows a region of interest to be specified, as well as subsampling factors that may be used to obtain a scaled-down version of the image.

When a source region is set, the reader plug-in will attempt to decode only the desired region, to the extent that the file format allows partial decoding. In any case, no pixels outside the region will appear in the output. This capability makes it possible to work with extremely large images in a limited amount of memory.

For example, to decode only the upper-left quadrant of the image, the application first obtains an `ImageReadParam` that is suitable for use with the reader:

```
ImageReadParam param = reader.getDefaultReadParam();
```

Next, the source region of interest is set on the `ImageReadParam`:

```
import java.awt.Rectangle;
int imageIndex = 0;
int half_width = reader.getImageWidth(imageIndex)/2;
int half_height = reader.getImageHeight(imageIndex)/2;
Rectangle rect = new Rectangle(0, 0, half_width, half_height);
param.setSourceRegion(rect);
```

Finally, the image is read using the `ImageReadParam`:

```
BufferedImage bi = reader.read(imageIndex, param);
```

The result is a new BufferedImage whose width and height are equal to half those of the original image (rounding down if the image had an odd width or height).

The lower-right quadrant of the image may then be read into the same BufferedImage that was created to hold the upper-left quadrant, overwriting the previous pixel data:

```
param.setDestination(bi);
rect = new Rectangle(half_width, half_height, half_width, half_height);
param.setSourceRegion(rect);
BufferedImage bi2 = reader.read(0, param);
if (bi == bi2) {
     System.out.println("The same BufferedImage was used!");
} else {
     System.out.println("This can't happen!");
}
```

In practice, the application could simply call reader.read(0, param) without assigning the result anywhere, knowing that the pixels will be written into the existing BufferedImage bi.

As another example, to read every third pixel of the image, resulting in an image one-ninth the size of the original, subsampling factors may be set in the ImageReadParam:

```
param = reader.getDefaultImageParam();
param.setSourceSubsampling(3, 3, 0, 0);
BufferedImage bi3 = reader.read(0, param);
```

3.3.2 IIOParamController

A plug-in may optionally supply an IIOParamController object that may be used to set up an IIOReadParam (or IIOWriteParam) using a graphical user interface (GUI), or any other interface. A reader plug-in may attach an IIOParamController to any ImageReadParam objects that it creates:

```
ImageReadParam param = reader.getDefaultImageParam();
IIOParamController controller = param.getController();
if (controller != null) {
     controller.activate(param);
}
```

When the controller's activate method is called, it displays the GUI and handles user events such as slider movements and button presses. Typically the interface will contain an "OK" or "Apply" button, which when pressed will cause the activate method to return. The controller is responsible for calling methods on its associated ImageReadParam to update its state, either in response to each GUI event, or all at once prior to returning from activate.

3.3.3 Reading From Multi-Image Files

All the methods in the `ImageReader` class that deal with images take an `imageIndex` parameter. This parameter allows access to any of the images in a multi-image file.

The `ImageReader.getNumImages` method returns the number of images that are stored in the input file. This method takes a boolean parameter, `allowSearch`. Some image formats, notably GIF, do not provide any way to determine the number of images without reading the entire file. Since this may be costly, setting `allowSearch` to `false` will allow the reader to return a value of `-1` instead of the actual number of images. If the parameter is `true`, the reader will always return the actual number of images.

Even if the number of images is not known by the application, it is still possible to call `read(imageIndex)`; if the index is too large, the method will throw an `IndexOutOfBoundsException`. Thus, the application can request images with increasing indices until it receives an exception.

3.3.4 Reading "Thumbnail" Images

Some image formats allow a small preview image (or multiple previews) to be stored alongside the main image. These "thumbnail" images are useful for identifying image files quickly, without the need to decode the entire image.

Applications can determine how many thumbnail images associated with a particular image are available by calling:

```
reader.getNumThumbnails(imageIndex);
```

If a thumbnail image is present, it can be retrieved by calling:

```
int thumbailIndex = 0;
BufferedImage bi;
bi = reader.readThumbnail(imageIndex, thumbnailIndex);
```

3.4 The `ImageWriter` Class

Just as we obtained a list of `ImageReaders` for a particular format using a method from the `ImageIO` class, we can obtain a list of `ImageWriters`:

```
Iterator writers = ImageIO.getImageWritersByFormatName("png");
ImageWriter writer = (ImageWriter)writers.next();
```

Once an ImageWriter has been obtained, its destination must be set to an ImageOutputStream:

```
File f = new File("c:\images\myimage.png");
ImageOutputStream ios = ImageIO.createImageOutputStream(f);
writer.setOutput(ios);
```

Finally, the image may be written to the output stream:

```
BufferedImage bi;
writer.write(bi);
```

3.4.1 Writing Multiple Images

The IIOImage class is use to store references to an image, one or more thumbnail images, and metadata (non-image data) associated with the image. Metadata will be discussed below; for now, we will simply pass null as the value for the metadata-related parameters.

The ImageWriter class contains a write method that creates a new file from an IIOImage, and a writeInsert method that adds an IIOImage to an existing file . By calling the two in sequence a multi-image file may be written:

```
BufferedImage first_bi, second_bi;
IIOImage first_IIOImage = new IIOImage(first_bi, null, null);
IIOImage second_IIOImage = new IIOImage(second_bi, null, null);
writer.write(null, first_IIOImage, null);
if (writer.canInsertImage(1)) {
     writer.writeInsert(1, second_IIOImage, null);
} else {
     System.err.println("Writer can't append a second image!");
}
```

3.5 Handling Metadata

Data stored in an image file that does not represent actual pixel values is referred to as *metadata*. The javax.imageio.metadata package contains classes and interfaces for accessing metadata.

Metadata may contain complex, hierarchical structures. The Java XML Document Object Model (DOM) API is used to represent these structures, allowing developers to leverage their knowledge of these interfaces.

The Image I/O API differentiates between *stream metadata*, which is associated with all images stored in an image file, and *image metadata* which is associated with a single image. For image formats that contain a single image, only image metadata is used.

Metadata may be obtained by calling the `ImageReader.getStreamMetadata` and `getImageMetadata(int imageIndex)` methods. These return an object that implements the `IIOMetadata` interface. The actual class type of the returned object is up to the `ImageReader`, and will usually be a unique class used only by that reader. This object should be designed to store as much of the metadata allowed by the format as possible, as losslessly as possible. However, this fidelity to the specifications of the image format comes at a cost, as access to the metadata becomes format-specific.

In order to allow access to the metadata without the need for format-specific application code, `IIOMetadata` objects expose their internal information in the form of an XML DOM structure, which is essentially a tree of nodes of various types that may contain a set of *attributes* (`String` values accessed by name), and which may reference a set of child nodes.

A single plug-in may support multiple document formats, which are distinguished by a format name. Typically, at least two formats will be supported by a given plug-in. The first is a common, plug-in neutral format called `com.sun.imageio_1.0`, which is defined in the class comment for the `IIOMetadata` interface. The second will be a highly plug-in specific format that exposes all of the internal structure of the `IIOMetadata` object in the form of a DOM. The latter format is referred to as the native format of the plug-in; its name may be determined by calling the `getNativeMetadataFormatName` method of the `IIOMetadata` object returned by the reader (advanced users may call a method of the same name on the `ImageReaderSpi` object used to instantiate the reader. The latter approach is useful for selecting a plug-in based on its support for a particular format). The names of all of the supported document formats may be determined similarly by calling `getMetadataFormatNames`.

The contents of an `IIOMetadata` object may be accessed in the form of a tree of XML `Node` objects by calling its `getAsTree` method. This method takes a `String` argument which is the name of one of the document formats supported by the plug-in. This document may then be manipulated as a standard XML DOM tree.

As an example, to print the contents of an XML DOM tree, the following code may be used:

```
public void displayMetadata(Node root) {
     displayMetadata(root, 0);
}

void indent(int level) {
     for (int i = 0; i < level; i++) {
          System.out.print("  ");
```

```
    }
}

void displayMetadata(Node node, int level) {
    indent(level); // emit open tag
    System.out.print("<" + node.getNodeName());
    NamedNodeMap map = node.getAttributes();
    if (map != null) { // print attribute values
        int length = map.getLength();
        for (int i = 0; i < length; i++) {
            Node attr = map.item(i);
            System.out.print(" " + attr.getNodeName() +
                             "=\"" + attr.getNodeValue() + "\"");
        }
    }

    Node child = node.getFirstChild();
    if (child != null) {
        System.out.println(">"); // close current tag
        while (child != null) { // emit child tags recursively
            displayMetadata(child, level + 1);
            child = child.getNextSibling();
        }
        indent(level); // emit close tag
        System.out.println("</" + node.getNodeName() + ">");
    } else {
        System.out.println("/>");
    }
}
```

Executing `displayMetadata` on the metadata from the standard PNG test image `http://www.schaik.com/pngsuite/ccwn2c08.png` yields the output:

```
<com.sun.imageio.png_1.0>
  <IHDR width="32" height="32" bitDepth="8" colorType="RGB"
compressionMethod="deflate" filterMethod="adaptive" interlaceMethod="none"/>
  <cHRM whitePointX="31270" whitePointY="32900" redX="64000" redY="33000"
greenX="30000" greenY="60000" blueX="15000" blueY="6000"/>
  <gAMA value="100000"/>
</com.sun.imageio.png_1.0>
```

We see that the image contains `IHDR`, `cHRM`, and `gAMA` chunks. The interpretation of the attribute values requires an understanding of the PNG format; however, it is still possible for an application that does not understand PNG internals to display the values and allow them to be edited interactively.

3.5.1 The `IIOMetadataFormat` Interface

An IIOMetadataFormat object is used to describe the legal structure of a metadata document format. It constrains the types of nodes that may appear, the types of nodes that may be children of a node of a given type, the names and data types of the attributes that may appear at a node of a given type, and the audiotape of an Object value that may be stored at a node of a given type. In XML terms, the information provided by the IIOMetadataFormat interface is somewhere in between a DTD (Document Type Definition), which gives information about node types, children, and attributes, and an XML Schema, which provides detailed information about data types.

For simplicity, only a subset of legal DTD structures can be described by IIOMetadataFormat. For example, the children of a node may be defined in an IIOMetadataFormat as a sequence in which each child must appear once (a, b, c), a sequence in which each child is optional (a?, b?, c?), a choice of a single child (a | b | c), or a repetition of a single node type (a)*, whereas a DTD allows many more combinations.

A node may not contain any textual data, but may contain a reference to an arbitrary Object. The IIOMetadataFormat indicates the class type and optionally the legal enumerated values or a range of values for the Object. Arrays are supported as well.

A DTD does not allow any attribute data types other than character strings; an XML Schema allows extremely complex data types to be built up from simpler ones. IIOMetadataFormat occupies a middle ground; it allows attributes to be constrained to belong to one of a predefined set of simple data types, including integers, floating-point decimals, and dates. Lists of these data types are also allowed.

Since an IIOMetadataFormat might be used to automatically construct a user interface for displaying and editing metadata, restricted the set of legal structures greatly simplifies the mapping between formats and user interface designs.

3.6 Transcoding

Transcoding refers to reading an image file with a reader plug-in, and writing out the image using a writer plug-in, especially, if the writer plug-in cannot directly understand the native metadata format of the reader. In such a case, there will always be some loss of information, even if both plug-ins deal with the same file format. A transcoder plug-in is responsible for converting the stream and image metadata that have been created by a reader plug-in into a better form for use by a writer plug-in. The API does not mandate the presence of any transcoder plug-ins, but they may be installed and registered just like readers and writers.

A set of reader and writer plug-ins may be designed to interpret each other's metadata without the need for a separate transcoder. The relationships between plug-ins can be determined using the ImageReaderSpi.getImageWriterSpiNames

and ImageWriterSpi.getImageReaderSpiNames methods, which indicate which plug-ins are known to work well with a given plug-in of the opposite type. If multiple plug-ins are being written by a single vendor, this mechanism will provide the best quality.

However, when working with plug-ins that have not been designed to interoperate, a transcoder plug-in may be provided by a developer who understands the metadata formats of a given pair of reader and writer plug-ins, even though each was written without knowledge of the other.

Given a particular pair of reader and writer plug-ins, the set of appropriate transcoder plug-ins may be located as follows:

```
ImageReader reader;
ImageWriter writer;

Iterator transcoders = ImageIO.getImageTranscoders(reader, writer);
```

3.7 Listening for Events

As an image is being read or written, the plug-in may provide updates to the application. The application may provide one or more classes that implement interfaces from the javax.image.event package. Instances of these classes are then added to the ImageReader or ImageWriter being used. For example,

```
class MyReadProgressListener implements IIOReadProgressListener {

     public MyReadProgressListener() {}

     public void imageStarted(ImageReader source) {
          System.out.println("Started reading!");
     }

     // Other methods from IIOReadProgressListener omitted
}

IIOReadProgressListener listener = new MyReadProgressListener();
reader.addIIOReadProgressListener(listener);
```

As the ImageReader.read method progresses, methods on listener will be called at various times to indicate how much of the image has been read. Because these methods are being called while ImageReader.read is active, they must not call most methods from the same ImageReader object. They may call ImageReader.abort(), which will cause ImageReader.read to return even if it is only partially complete.

3.7.1 The IIOReadProgressListener Interface

An IIOReadProgressListener may be used to provide simple status information during reading. An estimate of the percentage of completion of a read is provided, which may be used to update a Swing JProgressBar or other progress indicator, or to estimate the time remaining to read a large image.

The imageStarted method will be called at the start of the read. During the read, the imageProgress method will be called multiple times, each time with a different, increasing value for its percentageDone parameter. When the read is about to complete, the imageComplete method will be called.

Similarly, the thumbnailStarted, thumbnailProgress, and thumbnailComplete methods will be called during thumbnail reads.

Other methods exist to indicate the start and end of a sequence of image reads performed by the ImageReader.readAll method. Additionally, it is possible for an ongoing read to be aborted using the ImageReader.abort method; in this case, the listener's readAborted method will be called.

3.7.2 The IIOReadUpdateListener Interface

An IIOReadUpdateListener provides more detailed information on the progress of an image read. Some image formats allow *interlaced* or *progressive* encoding, in which a subset of the pixel data is made available quickly, so that a crude version of the image may be displayed while the remainder of the pixel data is still being received and decoded. A typical scheme might begin by sending only every fourth row, and only every fourth pixel in each of those rows, so that the initial image requires only one sixteenth of the total amount of data to be transmitted and decoded. If interlacing were not used, only the top one-sixteenth of the image would be displayed in the same amount of time. Thus, a person viewing an interlaced image will be able to get a sense of its contents much sooner than if a traditional left-to-right, top-to-bottom order were used.

By implementing the methods of the IIOReadUpdateListener interface, an application class can receive notification when groups of possibly non-contiguous pixels are ready to be displayed. These methods also receive a reference to the BufferedImage that is in the process of being filled in, which may be used to refresh the display to include the newly decoded pixels.

When decoding an interlaced or progressive image, the decoding proceeds in multiple passes. At the start of each pass, the listener's passStarted method will be called to indicate the set of pixels that may be overwritten during the following pass. This estimate is conservative; not every pixel in the region will necessarily be updated during the pass. As the pass progresses, the imageUpdate method will be called with arguments describing the region of pixels that have receive new values.

This region is described by the coordinates of its upper-left corner, its width and height, and the spacing between the pixels that make up the pass (in the example above, both parameters would be equal to 4 for the initial pass). When a pass has completed, the `passComplete` method is called. Similar methods allow the progress of thumbnail image reads to be tracked.

3.7.3 The `IIOReadWarningListener` Interface

By attaching an `IIOReadWarningListener` to an `ImageReader`, information on non-fatal errors may be received. For example, a reader might detect a tag or chunk that should not be present. Even though the reader may choose to ignore the error and proceed with decoding, it may wish to inform the application that the input source was malformed, as this could indicate problems in the application that generated the images.

`ImageReaders` may specify a set of `Locales` for which they can provide localized warning messages. The set of available locales can be obtained from the reader's `getAvailableLocales` method. The desired locale should then be set by calling the reader's `setLocale` method prior to attaching the `IIOReadWarningListener`. Each listener will receive messages in the `Locale` that was in effect at the time it was attached to the reader.

3.7.4 The `IIOWriteProgressListener` and `IIOWriteWarningListener` Interfaces

The `IIOWriteProgressListener` and `IIOWriteWarningListener` interfaces function similarly to their reader counterparts.

3.8 Handling Errors using `IIOException`

In the examples above, the possibility of fatal errors was not considered. Errors may result from a number of sources, including true I/O errors (*e.g.*, file not found, file unreadable, file on corrupt media), security violations (*e.g.*, no permission to read files from an applet), file format problems (file contents corrupted, file using a variant of the format that is not supported by the plug-in), or even bugs in the API implementation or in the plug-in.

The Image I/O API makes use of its own subclass of the standard `IOException` class, called `IIOException`. `IIOExceptions` are used to signal all errors encountered during the parsing of a source file (e.g., an incorrect checksum or an invalid value for a particular byte within the file), including true I/O errors that result in an `IOException` being thrown within the reader.

An `IIOException` contains a (non-localized) message describing the reason for the exception, as well as a reference to another `Exception` that was the cause of the `IIOException`, if one exists.

Thus, application code that attempts to provide graceful handling of errors will look something like:

```
File f = new File("c:\images\myimage.gif");
ImageInputStream iis = null;
try {
     iis = ImageIO.createImageInputStream(f);
} catch (IIOException iioe1) {
     System.out.println("Unable to create an input stream!");
     return;
}

reader.setInput(stream);
try {
     reader.read(0, param);
} catch (IIOException iioe2) {
     System.out.println("An error occurred during reading: " +
                        iioe2.getMessage());
     Throwable t = iioe2.getCause();
     if ((t != null) && (t instanceof IOException)) {
          System.out.println("Caused by IOException: " +
                             t.getMessage());
     }
}
```

CHAPTER 4

Writing Image I/O Plug-ins

4.1 The Plug-in Concept

The Image I/O API is designed as a pluggable framework into which any developer may add their own "plug-ins." A plug-in is defined as a set of Java™ programming language classes which may be loaded into the API at run-time and which add functionality to the API. In the context of the Java™ Image I/O API, a plug-in may provide the ability to read image data from a new file format, to write image data in a new format, to "transcode" non-image metadata between two formats, or to read or write streaming data from or to a new data source or sink. A plug-in may also provide support for the same format as another plug-in, perhaps providing better performance, more features, or a different view of the data stored by the format.

In order for plug-ins to be added to a running Java™ 2 Virtual Machine[1], they must be compiled into Java bytecode files (class files). These class files will contain the code for subclasses of various classes defined by the API. For example, a plug-in that provides the capability to read images will include a new subclass of the abstract `javax.imageio.ImageReader` class defined by the API. The usual reversed Internet domain name convention may be used to guarantee uniqueness of class names.

Because loading and instantiating an entire plug-in may be expensive, an additional class is used as a "stand-in" to provide information about the plug-in. This class may be used, for example, to determine what formats are handled by the plug-in prior to actually instantiating the plug-in.

The "stand-in" object is lightweight enough that it can be loaded and a single instance instantiated every time the API is used within a given invocation of the Java virtual machine. This pattern, in which a small class is used to provide information about an available service, is referred to as a "service provider interface."

1. As used in this guide, the terms "Java Virtual Machine" or "JVM" mean a virtual machine for the Java platform.

4.2 Embedding Plug-ins in JAR Files

Since a plug-in consists of several classes, the JAR file mechanism is used to allow them to be combined into a single file. In addition to class files, JAR files may contain additional files used to describe their contents. In particular, a JAR file may contain a `META-INF/services` directory that is used to list any service providers that are contained in the file. For each service provider interface that is implemented by a class stored in the JAR file, a file whose name is the fully-qualified class name of the service provider interface is placed within the `services` directory. The file should contain the fully-qualified class names of the implementation classes present in the JAR file, one per line. For example, if the JAR file contains a service provider class named `com.mycompany.mypackage.MyImageReaderSpi`, which implements the `javax.imageio.spi.ImageReaderSpi` interface, there should be a file named `META-INF/services/javax.imageio.spi.ImageReaderSpi` containing the line `com.mycompany.mypackage.MyImageReaderSpi`.

The Image I/O API will automatically examine any JAR files that are found on the class path, and identify those that contain Image I/O plug-ins. For each plug-in found, a single instance of its service provider class will be instantiated and stored in a run-time registry class, `javax.imageio.spi.IIORegistry`.

JAR files on the application class path (*i.e.*, the path set using the `CLASSPATH` variable), or elsewhere (e.g., available via a network URL) may be loaded by the application. They are not loaded by default in order to minimize startup time.

The `javax.imageio.spi.IIORegistry` class may also be used at run-time to deregister installed plug-ins and to register new ones. In particular, it is possible to use a `ClassLoader` to load a plug-in over the network, which may then be registered and used exactly like a locally installed plug-in.

Most applications should not need to deal directly with the registry. Instead, they may make use of convenience methods contained within the `javax.imageio.ImageIO` class that search for appropriate plug-ins automatically.

4.3 Writing Reader Plug-Ins

A minimal reader plug-in consists of an `ImageReaderSpi` subclass, and an `ImageReader` subclass. Optionally, it may contain implementations of the `IIOMetadata` interface representing the stream and image metadata, and an `IIOMetadataFormat` object describing the structure of the metadata.

In the following sections, we will sketch out the implementation of a simple reader plug-in for a hypothetical format called “MyFormat”. It will consist of the classes MyFormatImageReaderSpi, MyFormatImageReader, and MyFormatMetadata.

The format itself is defined to begin with the characters ‘myformat\n’, followed by two four-byte integers representing the width, height, and a single byte indicating the color type of the image, which may be either gray or RGB. Next, after a newline character, metadata values may stored as alternating lines containing a keyword and a value, terminated by the special keyword ‘END’. The string values are stored using UTF8 encoding followed by a newline. Finally, the image samples are stored in left-to-right, top-to-bottom order as either byte grayscale values, or three bytes representing red, green, and blue.

MyFormatImageReaderSpi

The MyFormatImageReaderSpi class provides information about the plug-in, including the vendor name, plug-in version string and description, format name, file suffixes associated with the format, MIME types associated with the format, input source classes that the plug-in can handle, and the ImageWriterSpis of plug-ins that are able to interoperate specially with the reader. It also must provide an implementation of the canDecodeInput method, which is used to locate plug-ins based on the contents of a source image file.

The ImageReaderSpi class provides implementations of most of its methods. These methods mainly return the value of various protected instance variables, which the MyFormatImageReaderSpi may set directly or via the superclass constructor, as in the example below:

```
package com.mycompany.imageio;

import java.io.IOException;
import java.util.Locale;
import javax.imageio.ImageReader;
import javax.imageio.spi.ImageReaderSpi;
import javax.imageio.stream.ImageInputStream;

public class MyFormatImageReaderSpi extends ImageReaderSpi {

    static final String vendorName = “My Company”;
    static final String version = “1.0_beta33_build9467”;
    static final String readerClassName =
        “com.mycompany.imageio.MyFormatImageReader”;
    static final String[] names = { “myformat” };
    static final String[] suffixes = { “myf” };
    static final String[] MIMETypes = {
        “image/x-myformat” };
    static final String[] writerSpiNames = {
        “com.mycompany.imageio.MyFormatImageWriterSpi” };

    // Metadata formats, more information below
```

```
static final boolean supportsStandardStreamMetadataFormat = false;
static final String nativeStreamMetadataFormatName = null
static final String nativeStreamMetadataFormatClassName = null;
static final String[] extraStreamMetadataFormatNames = null;
static final String[] extraStreamMetadataFormatClassNames = null;
static final boolean supportsStandardImageMetadataFormat = false;
static final String nativeImageMetadataFormatName =
     "com.mycompany.imageio.MyFormatMetadata_1.0";
static final String nativeImageMetadataFormatClassName =
     "com.mycompany.imageio.MyFormatMetadata";
static final String[] extraImageMetadataFormatNames = null;
static final String[] extraImageMetadataFormatClassNames = null;

public MyFormatImageReaderSpi() {
     super(vendorName, version,
           names, suffixes, MIMETypes,
           readerClassName,
           STANDARD_INPUT_TYPE, // Accept ImageInputStreams
           writerSpiNames,
           supportsStandardStreamMetadataFormat,
           nativeStreamMetadataFormatName,
           nativeStreamMetadataFormatClassName,
           extraStreamMetadataFormatNames,
           extraStreamMetadataFormatClassNames,
           supportsStandardImageMetadataFormat,
           nativeImageMetadataFormatName,
           extraImageMetadataFormatNames,
           extraImageMetadataFormatClassNames);
     }

     public String getDescription(Locale locale) {
           // Localize as appropriate
           return “Description goes here”;
     }

     public boolean canDecodeInput(Object input)
           throws IOException {
           // see below
     }

       public ImageReader createReaderInstance(Object extension) {
           return new MyFormatImageReader(this);
        }
}
```

Most plug-ins need read only from `ImageInputStream` sources, since it is possible to “wrap” most other types of input with an appropriate `ImageInputStream`. However, it is possible for a plug-in to work directly with other `Objects`, for example an `Object` that provides an interface to a digital camera or scanner. This interface need not provide a “stream” view of the device at all. Rather, a plug-in that is aware of the interface may use it to drive the device directly.

The plug-in advertises which input classes it can handle via its getInputTypes method, which returns an array of Class objects. An implementation of getInputTypes is provided in the superclass, which returns the value of the inputTypes instance variable, which in turn is set by the seventh argument to the superclass constructor. The value used in the example above, STANDARD_INPUT_TYPE, is shorthand for an array containing the single element javax.imageio.stream.ImageInputStream.class, indicating that the plug-in accepts only ImageInputStreams.

The canDecodeInput method is responsible for determining two things: first, whether the input parameter is an instance of a class that the plug-in can understand, and second, whether the file contents appear to be in the format handled by the plug-in. It must leave its input in the same state as it was when it was passed in. For an ImageInputStream input source, the mark and reset methods may be used. For example, since files in the "MyFormat" format all begin with the characters 'myformat', canDecodeInput may be implemented as:

```
public boolean canDecodeInput(Object input) {
     if (!(input instanceof ImageInputStream)) {
          return false;
     }

     ImageInputStream stream = (ImageInputStream)input;
     byte[] b = new byte[8];
     try {
          stream.mark();
          stream.readFully(b);
          stream.reset();
     } catch (IOException e) {
          return false;
     }

     // Cast unsigned character constants prior to comparison
     return (b[0] == (byte)'m' && b[1] == (byte)'y' &&
             b[2] == (byte)'f' && b[3] == (byte)'o' &&
             b[4] == (byte)'r' && b[5] == (byte)'m' &&
             b[6] == (byte)'a' && b[7] == (byte)'t');
}
```

MyFormatImageReader

The heart of a reader plug-in is its extension of the ImageReader class. This class is responsible for responding to queries about the images actually stored in an input file or stream, as well as the actual reading of images, thumbnails, and metadata. For simplicity, we will ignore thumbnail images in this example.

A sketch of some of the methods of a hypothetical MyFormatImageReader class is shown below:

```
package com.mycompany.imageio;
```

```
public class MyFormatImageReader extends ImageReader {

     ImageInputStream stream = null;
     int width, height;
     int colorType;

     // Constants enumerating the values of colorType
     static final int COLOR_TYPE_GRAY = 0;
     static final int COLOR_TYPE_RGB = 1;

     boolean gotHeader = false;

     public MyFormatImageReader(ImageReaderSpi originatingProvider) {
          super(originatingProvider);
     }

     public void setInput(Object input, boolean isStreamable) {
          super.setInput(input, isStreamable);
          if (input == null) {
               this.stream = null;
               return;
          }
          if (input instanceof ImageInputStream) {
               this.stream = (ImageInputStream)input;
          } else {
               throw new IllegalArgumentException("bad input");
          }
     }

     public int getNumImages(boolean allowSearch)
          throws IIOException {
          return 1; // format can only encode a single image
     }

     private void checkIndex(int imageIndex) {
          if (imageIndex != 0) {
               throw new IndexOutOfBoundsException("bad index");
          }
     }

     public int getWidth(int imageIndex)
          throws IIOException {
          checkIndex(imageIndex); // must throw an exception if != 0
          readHeader();
          return width;
     }

     public int getHeight(int imageIndex)
          throws IIOException {
          checkIndex(imageIndex);
          readHeader();
          return height;
```

```
}
```

The `getImageTypes` *Method*

The reader is responsible for indicating what sorts of images may be used to hold the decoded output. The `ImageTypeSpecifier` class is used to hold a `SampleModel` and `ColorModel` indicating a legal image layout. The `getImageTypes` method returns an `Iterator` of `ImageTypeSpecifiers`:

```
public Iterator getImageTypes(int imageIndex)
      throws IIOException {
      checkIndex(imageIndex);
      readHeader();

      ImageTypeSpecifier imageType = null;
      int datatype = DataBuffer.TYPE_BYTE;
      java.util.List l = new ArrayList();
      switch (colorType) {
      case COLOR_TYPE_GRAY:
            imageType = ImageTypeSpecifier.createGrayscale(8,
                                                           datatype,
                                                           false);
            break;

      case COLOR_TYPE_RGB:
            ColorSpace rgb =
                  ColorSpace.getInstance(ColorSpace.CS_sRGB);
            int[] bandOffsets = new int[3];
            bandOffsets[0] = 0;
            bandOffsets[1] = 1;
            bandOffsets[2] = 2;
            imageType =
                  ImageTypeSpecifier.createInterleaved(rgb,
                                                       bandOffsets,
                                                       datatype,
                                                       false,
                                                       false);
            break;
      }
      l.add(imageType);
      return l.iterator();
}
```

Parsing the Image Header

Several of the methods above depend on a readHeader method, which is responsible for reading enough of the input stream to determine the width, height, and layout of the image. readHeader is defined so it is safe to be called multiple times (note that we are not concerned with multi-threaded access):

```
public void readHeader() {
      if (gotHeader) {
            return;
      }
      gotHeader = true;

      if (stream == null) {
            throw new IllegalStateException("No input stream");
      }

      // Read 'myformat\n' from the stream
      byte[] signature = new byte[9];
      try {
            stream.readFully(signature);
      } catch (IOException e) {
            throw new IIOException("Error reading signature", e);
      }
      if (signature[0] != (byte)'m' || ...) { // etc.
            throw new IIOException("Bad file signature!");
      }
      // Read width, height, color type, newline
      try {
            this.width = stream.readInt();
            this.height = stream.readInt();
            this.colorType = stream.readUnsignedByte();
            stream.readUnsignedByte(); // skip newline character
      } catch (IOException e) {
            throw new IIOException("Error reading header", e)
      }
}
```

The actual reading of the image is handled by the read method:

```
public BufferedImage read(int imageIndex, ImageReadParam param)
      throws IIOException {
      readMetadata(); // Stream is positioned at start of image data
```

Handling the ImageReadParam

The first section of the method is concerned with using a supplied ImageReadParam object to determine what region of the source image is to be read, what sort of subsampling is to be applied, the selection and rearrangement of bands, and the offset in the destination:

```
// Compute initial source region, clip against destination later
Rectangle sourceRegion = getSourceRegion(param, width, height);

// Set everything to default values
int sourceXSubsampling = 1;
int sourceYSubsampling = 1;
int[] sourceBands = null;
int[] destinationBands = null;
Point destinationOffset = new Point(0, 0);

// Get values from the ImageReadParam, if any
if (param != null) {
      sourceXSubsampling = param.getSourceXSubsampling();
      sourceYSubsampling = param.getSourceYSubsampling();
      sourceBands = param.getSourceBands();
      destinationBands = param.getDestinationBands();
      destinationOffset = param.getDestinationOffset();
}
```

At this point, the region of interest, subsampling, band selection, and destination offset have been initialized. The next step is to create a suitable destination image. The ImageReader.getDestination method will return any image that was specified using ImageReadParam.setDestination, or else will create a suitable destination image using a supplied ImageTypeSpecifier, in this case determined by calling getImageTypes(0):

```
// Get the specified detination image or create a new one
BufferedImage dst = getDestination(param,
                                   getImageTypes(0),
                                   width, height);
// Enure band settings from param are compatible with images
int inputBands = (colorType == COLOR_TYPE_RGB) ? 3 : 1;
checkReadParamBandSettings(param, inputBands,
                           dst.getSampleModel().getNumBands());
```

To reduce the amount of code we have to write, we create a Raster to hold a row's worth of data, and copy the pixels from that Raster into the actual image. In this way, band selection and the details of pixel formatting are taken care of, at the expense of an additional copy.

```
int[] bandOffsets = new int[inputBands];
for (int i = 0; i < inputBands; i++) {
      bandOffsets[i] = i;
```

```
}
int bytesPerRow = width*inputBands;
DataBufferByte rowDB = new DataBufferByte(bytesPerRow);
WritableRaster rowRas =
      Raster.createInterleavedRaster(rowDB,
                                     width, 1, bytesPerRow,
                                     inputBands, bandOffsets,
                                     new Point(0, 0));
byte[] rowBuf = rowDB.getData();

// Create an int[] that can a single pixel
int[] pixel = rowRas.getPixel(0, 0, (int[])null);
```

Now we have a byte array, `rowBuf`, which can be filled in from the input data, and which is also the source of pixel data for the `Raster` `rowRaster`. We extract the (single) tile of the destination image, and determine its extent. Then we create child rasters of both the source and destination that select and order their bands according to the settings previously extracted from the `ImageReadParam`:

```
WritableRaster imRas = dst.getWritableTile(0, 0);
int dstMinX = imRas.getMinX();
int dstMaxX = dstMinX + imRas.getWidth() - 1;
int dstMinY = imRas.getMinY();
int dstMaxY = dstMinY + imRas.getHeight() - 1;

// Create a child raster exposing only the desired source bands
if (sourceBands != null) {
      rowRas = rowRas.createWritableChild(0, 0,
                                          width, 1,
                                          0, 0,
                                          sourceBands);
}

// Create a child raster exposing only the desired dest bands
if (destinationBands != null) {
      imRas = imRas.createWritableChild(0, 0,
                                        imRas.getWidth(),
                                        imRas.getHeight(),
                                        0, 0,
                                        destinationBands);
}
```

Reading the Pixel Data

Now we are ready to begin read pixel data from the image. We will read whole rows, and perform subsampling and destination clipping as we proceed. The horizontal clipping is complicated by the need to take subsampling into account. Here we perform per-pixel clipping; a more sophisticated reader could perform horizontal clipping once:

```
for (int srcY = 0; srcY < height; srcY++) {
      // Read the row
      try {
            stream.readFully(rowBuf);
      } catch (IOException e) {
            throw new IIOException("Error reading line " + srcY, e);
      }

      // Reject rows that lie outside the source region,
      // or which aren't part of the subsampling
      if ((srcY < sourceRegion.y) ||
          (srcY >= sourceRegion.y + sourceRegion.height) ||
          (((srcY - sourceRegion.y) %
            sourceYSubsampling) != 0)) {
            continue;
      }

      // Determine where the row will go in the destination
      int dstY = destinationOffset.y +
            (srcY - sourceRegion.y)/sourceYSubsampling;
      if (dstY < dstMinY) {
            continue; // The row is above imRas
      }
      if (dstY > dstMaxY) {
            break; // We're done with the image
      }

      // Copy each (subsampled) source pixel into imRas
      for (int srcX = sourceRegion.x;
           srcX < sourceRegion.x + sourceRegion.width;
           srcX++) {
            if (((srcX - sourceRegion.x) % sourceXSubsampling) != 0) {
                  continue;
            }
            int dstX = destinationOffset.x +
                  (srcX - sourceRegion.x)/sourceXSubsampling;
            if (dstX < dstMinX) {
                  continue;  // The pixel is to the left of imRas
            }
            if (dstX > dstMaxX) {
                  break; // We're done with the row
            }

            // Copy the pixel, sub-banding is done automatically
            rowRas.getPixel(srcX, 0, pixel);
            imRas.setPixel(dstX, dstY, pixel);
      }
}
return dst;
```

For performance, the case where `sourceXSubsampling` is equal to 1 may be broken out separately, since it is possible to copy multiple pixels at once:

```
// Create an int[] that can hold a row's worth of pixels
int[] pixels = rowRas.getPixels(0, 0, width, 1, (int[])null);

// Clip against the left and right edges of the destination image
int srcMinX =
     Math.max(sourceRegion.x,
                dstMinX - destinationOffset.x + sourceRegion.x);
 int srcMaxX =
     Math.min(sourceRegion.x + sourceRegion.width - 1,
              dstMaxX - destinationOffset.x + sourceRegion.x);
int dstX = destinationOffset.x + (srcMinX - sourceRegion.x);
int w = srcMaxX - srcMinX + 1;
rowRas.getPixels(srcMinX, 0, w, 1, pixels);
imRas.setPixels(dstX, dstY, w, 1, pixels);
```

There are several additional features that readers should implement, namely informing listeners of the progress of the read, and allowing the read process to be aborted from another thread.

Listeners

There are three types of listeners that may be attached to a reader: `IIOReadProgressListener`, `IIOReadUpdateListener`, and `IIOReadWarningListener`. Any number of each type may be attached to a reader by means of various add and remove methods that are implemented in the ImageReader superclass. ImageReader also contains various process methods that broadcast information to all of the attached listeners of a given type. For example, when the image read begins, the method processImageStarted(imageIndex) should be called to inform all attached IIOReadProgressListeners of the event.

A reader plug-in is normally responsible for calling `processImageStarted` and `processImageComplete` at the beginning and end of its read method, respectively. `processImageProgress` should be called at least every few scanlines with an estimate of the percentage completion of the read. It is important that this percentage never decrease during the read of a single image. If the reader supports thumbnails, the corresponsing thumbnail progress methods should be called as well. The `processSequenceStarted` and `processSequenceComplete` methods of `IIOReadProgressListener` only need to be called if the plug-in overrides the superclass implementation of `readAll`.

More advanced readers that process incoming data in multiple passes may choose to support `IIOReadUpdateListeners`, which receive more detauled information about which pixels have been read so far. Applications may use this information to perform selective updates of an on-screen image, for example, or to re-encode image data in a streaming fashion.

Aborting the Read Process

While one thread performs an image read, another thread may call the reader's abort method asynchronously. The reading thread should poll the reader's status periodically using the `abortRequested` method, and attempt to cut the decoding short. The partially decoded image should still be returned, although the reader need not make any guarantees about its contents. For example, it could contain compressed or encrypted data in its `DataBuffer` that does not make sense visually.

`IIOReadProgressListener` *Example*

A typical set of `IIOReadProgressListener` calls might look like this:

```
public BufferedImage read(int imageIndex, ImageReadParam param)
     throws IOException {
     // Clear any previous abort request
     boolean aborted = false;
     clearAbortRequested();

     // Inform IIOReadProgressListeners of the start of the image
     processImageStarted(imageIndex);

     // Compute xMin, yMin, xSkip, ySkip from the ImageReadParam
     // ...

     // Create a suitable image
     BufferedImage theImage = new BufferedImage(...);

     // Compute factors for use in reporting percentages
     int pixelsPerRow = (width - xMin + xSkip - 1)/xSkip;
     int rows = (height - yMin + ySkip - 1)/ySkip;
     long pixelsDecoded = 0L;
     long totalPixels = rows*pixelsPerRow;

     for (int y = yMin; y < height; y += yskip) {
          // Decode a (subsampled) scanline of the image
          // ...

          // Update the percentage estimate
          // This may be done only every few rows if desired
          pixelsDecoded += pixelsPerRow;
          processImageProgress(100.0F*pixelsDecoded/totalPixels);

          // Check for an asynchronous abort request
          if (abortRequested()) {
               aborted = true;
               break;
          }
     }
```

```
        // Handle the end of decoding
        if (aborted) {
            processImageAborted();
        } else {
            processImageComplete(imageIndex);
        }

        // If the read was aborted, we still return a partially decoded image
        return theImage;
    }
```

Metadata

The next set of methods in `MyFormatImageReader` deal with metadata. Because our hypothetical format only encodes a single image, we may ignore the concept of "stream" metadata, and use "image" metadata only:

```
    MyFormatMetadata metadata = null; // class defined below

    public IIOMetadata getStreamMetadata()
        throws IIOException {
        return null;
    }

    public IIOMetadata getImageMetadata(int imageIndex)
        throws IIOException {
        if (imageIndex != 0) {
            throw new IndexOutOfBoundsException("imageIndex != 0!");
        }
        readMetadata();
        return metadata;
    }
```

The actual work is done by a format-specific method `readMetadata`, which for this format fills in the keyword/value pairs of the metadata object,

```
    public void readMetadata() throws IIOException {
        if (metadata != null) {
            return;
        }
        readHeader();
        this.metadata = new MyFormatMetadata();
        try {
            while (true) {
                String keyword = stream.readUTF();
                stream.readUnsignedByte();
                if (keyword.equals("END")) {
                    break;
                }
                String value = stream.readUTF();
```

```
                    stream.readUnsignedByte();

                    metadata.keywords.add(keyword);
                    metadata.values.add(value);
                } catch (IIOException e) {
                    throw new IIOException("Exception reading metadata",
                                           e);
                }
            }
        }
```

MyFormatMetadata

Finally, the various interfaces for extracting and editing metadata must be defined. We define a class called `MyFormatMetadata` that extends the `IIOMetadata` class, and additionally can store the keyword/value pairs that are allowed in the file format:

```
package com.mycompany.imageio;

import org.w3c.dom.*;
import javax.xml.parsers.*; // Package name may change in JDK 1.4

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javax.imageio.metadata.IIOInvalidTreeException;
import javax.imageio.metadata.IIOMetadata;
import javax.imageio.metadata.IIOMetadataFormat;
import javax.imageio.metadata.IIOMetadataNode;

public class MyFormatMetadata extends IIOMetadata {

    static final boolean standardMetadataFormatSupported = false;
    static final String nativeMetadataFormatName =
        "com.mycompany.imageio.MyFormatMetadata_1.0";
    static final String nativeMetadataFormatClassName =
        "com.mycompany.imageio.MyFormatMetadata";
    static final String[] extraMetadataFormatNames = null;
    static final String[] extraMetadataFormatClassNames = null;

    // Keyword/value pairs
    List keywords = new ArrayList();
    List values = new ArrayList();
```

The first set of methods are common to most IIOMetadata implementations:

```
    public MyFormatMetadata() {
        super(standardMetadataFormatSupported,
              nativeMetadataFormatName,
              nativeMetadataFormatClassName,
              extraMetadataFormatNames,
              extraMetadataFormatClassNames);
```

```
}

public IIOMetadataFormat getMetadataFormat(String formatName) {
      if (!formatName.equals(nativeMetadataFormatName)) {
            throw new IllegalArgumentException("Bad format name!");
      }
      return MyFormatMetadataFormat.getDefaultInstance();
}
```

The most important method for reader plug-ins is `getAsTree`:

```
public Node getAsTree(String formatName) {
      if (!formatName.equals(nativeMetadataFormatName)) {
            throw new IllegalArgumentException("Bad format name!");
      }

      // Create a root node
      IIOMetadataNode root =
            new IIOMetadataNode(nativeMetadataFormatName);

      // Add a child to the root node for each keyword/value pair
      Iterator keywordIter = keywords.iterator();
      Iterator valueIter = values.iterator();
      while (keywordIter.hasNext()) {
            IIOMetadataNode node =
                  new IIOMetadataNode(“KeywordValuePair”);
            node.setAttribute(“keyword”, (String)keywordIter.next());
            node.setAttribute(“value”, (String)valueIter.next());
            root.appendChild(node);
      }

      return root;
}
```

For writer plug-ins, the ability to edit metadata values is obtained by implementing the `isReadOnly`, `reset`, and `mergeTree` methods:

```
public boolean isReadOnly() {
    return false;
}

public void reset() {
    this.keywords = new ArrayList();
    this.values = new ArrayList();
}

public void mergeTree(String formatName, Node root)
      throws IIOInvalidTreeException {
      if (!formatName.equals(nativeMetadataFormatName)) {
            throw new IllegalArgumentException("Bad format name!");
      }
```

```
            Node node = root;
            if (!node.getNodeName().equals(nativeMetadataFormatName)) {
                  fatal(node, “Root must be “ + nativeMetadataFormatName);
            }
            node = node.getFirstChild();
            while (node != null) {
                  if (!node.getNodeName().equals(“KeywordValuePair”)) {
                        fatal(node, “Node name not KeywordValuePair!”);
                  }
                  NamedNodeMap attributes = node.getAttributes();
                  Node keywordNode = attributes.getNamedItem(“keyword”);
                  Node valueNode = attributes.getNamedItem(“value”);
                  if (keywordNode == null || valueNode == null) {
                        fatal(node, “Keyword or value missing!”);
                  }

                  // Store keyword and value
                  keywords.add((String)keywordNode.getNodeValue());
                  values.add((String)valueNode.getNodeValue());

                  // Move to the next sibling
                  node = node.getNextSibling();
            }
      }

      private void fatal(Node node, String reason)
            throws IIOInvalidTreeException {
            throw new IIOInvalidTreeException(reason, node);
      }
}
```

MyFormatMetadataFormat

The tree structure of the metadata may be described using the IIOMetadataFormat interface. An implementation class, IIOMetadataFormatImpl, takes care of maintaining the “database” of information about elements, their attributes, and the parent-child relationships between them:

```
package com.mycompany.imageio;

import javax.imageio.ImageTypeSpecifier;
import javax.imageio.metadata.IIOMetadataFormatImpl;

public class MyFormatMetadataFormat extends IIOMetadataFormatImpl {

      // Create a single instance of this class (singleton pattern)
      private static MyFormatMetadataFormat defaultInstance =
            new MyFormatMetadataFormat();

      // Make constructor private to enforce the singleton pattern
      private MyFormatMetadataFormat() {
            // Set the name of the root node
```

```
            // The root node has a single child node type that may repeat
            super("com.mycompany.imageio.MyFormatMetadata_1.0",
                  CHILD_POLICY_REPEAT);

            // Set up the "KeywordValuePair" node, which has no children
            addElement("KeywordValuePair",
                       "com.mycompany.imageio.MyFormatMetadata_1.0",
                       CHILD_POLICY_EMPTY);

            // Set up attribute "keyword" which is a String that is required
            // and has no default value
            addAttribute("KeywordValuePair", "keyword", DATATYPE_STRING,
                         true, null);
            // Set up attribute "value" which is a String that is required
            // and has no default value
            addAttribute("KeywordValuePair", "value", DATATYPE_STRING,
                         true, null);
      }

      // Check for legal element name
      public boolean canNodeAppear(String elementName,
                                   ImageTypeSpecifier imageType) {
            return elementName.equals("KeywordValuePair");
      }

      // Return the singleton instance
      public static MyFormatMetadataFormat getDefaultInstance() {
            return defaultInstance;
      }
}
```

4.4 Writing Writer Plug-ins

MyFormatImageWriterSpi

The `MyFormatImageWriterSpi` call plays a similar role to the `MyFormatImageReaderSpi` class discussed in the previous section. However, instead of being responsible for determining whether a given stream can be read, it must deterine whether an image in memory can be written. Rather than inspecting the image itself, an `ImageTypeSpecifier` is used so that writers may be selected before an actual image is available.

```
package com.mycompany.imageio;
```

```
import java.io.IOException;
import java.util.Locale;
import javax.imageio.ImageWriter;
import javax.imageio.ImageTypeSpecifier;
import javax.imageio.spi.ImageWriterSpi;
import javax.imageio.stream.ImageInputStream;

public class MyFormatImageWriterSpi extends ImageWriterSpi {

     static final String vendorName = “My Company”;
     static final String version = “1.0_beta33_build9467”;
     static final String writerClassName =
          “com.mycompany.imageio.MyFormatImageWriter”;
     static final String[] names = { “myformat” };
     static final String[] suffixes = { “myf” };
     static final String[] MIMETypes = { “image/x-myformat” };
     static final String[] readerSpiNames = {
          “com.mycompany.imageio.MyFormatImageReaderSpi” };

     static final boolean supportsStandardStreamMetadataFormat = false;
     static final String nativeStreamMetadataFormatName = null;
     static final String nativeStreamMetadataFormatClassName = null;
     static final String[] extraStreamMetadataFormatNames = null;
     static final String[] extraStreamMetadataFormatClassNames = null;
     static final boolean supportsStandardImageMetadataFormat = false;
     static final String nativeImageMetadataFormatName =
          "com.mycompany.imageio.MyFormatMetadata_1.0";
     static final String nativeImageMetadataFormatClassName =
          "com.mycompany.imageio.MyFormatMetadata";
     static final String[] extraImageMetadataFormatNames = null;
     static final String[] extraImageMetadataFormatClassNames = null;

     public MyFormatImageWriterSpi() {
          super(vendorName, version,
                names, suffixes, MIMETypes,
                writerClassName,
                STANDARD_OUTPUT_TYPE, // Write to ImageOutputStreams
                readerSpiNames,
                supportsStandardStreamMetadataFormat,
                nativeStreamMetadataFormatName,
                nativeStreamMetadataFormatClassName,
                extraStreamMetadataFormatNames,
                extraStreamMetadataFormatClassNames,
                supportsStandardImageMetadataFormat,
                nativeImageMetadataFormatName,
                nativeImageMetadataFormatClassName,
                extraImageMetadataFormatNames,
                extraImageMetadataFormatClassNames);
     }

     public boolean canEncodeImage(ImageTypeSpecifier imageType) {
          int bands = imageType.getNumBands();
          return bands == 1 || bands == 3;
```

```
    }

    public String getDescription(Locale locale) {
        // Localize as appropriate
        return “Description goes here”;
    }

    public ImageWriter createWriterInstance(Object extension) {
        return new MyFormatImageWriter(this);
    }
}
```

MyFormatImageWriter

```
package com.mycompany.imageio;

import java.awt.Rectangle;
import java.awt.image.Raster;
import java.awt.image.RenderedImage;
import java.io.IOException;
import java.util.Iterator;
import javax.imageio.IIOException;
import javax.imageio.IIOImage;
import javax.imageio.ImageTypeSpecifier;
import javax.imageio.ImageWriteParam;
import javax.imageio.ImageWriter;
import javax.imageio.metadata.IIOMetadata;
import javax.imageio.spi.ImageWriterSpi;
import javax.imageio.stream.ImageOutputStream;

public class MyFormatImageWriter extends ImageWriter {

    ImageOutputStream stream = null;

    public MyFormatImageWriter(ImageWriterSpi originatingProvider) {
        super(originatingProvider);
    }

    public void setOutput(Object output) {
        super.setOutput(output);
        if (output != null) {
            if (!(output instanceof ImageOutputStream)) {
                throw new IllegalArgumentException
                    (“output not an ImageOutputStream!”);
            }
            this.stream = (ImageOutputStream)output;
        } else {
            this.stream = null;
        }
    }
```

The ImageWriteParam returned by getDefaultWriteParam must be customized based on the writer's capabilities. Since this writer does not support tiling, progessive encoding, or compression, we pass in values of false or null as appropriate:

```
// Tiling, progressive encoding, compression are disabled by default
public ImageWriteParam getDefaultWriteParam() {
      return new ImageWriteParam(getLocale());
}
```

The format only handles image metadata. The convertImageMetadata method does very little; it could be defined to interpret the metadata classes used by other plug-ins.

```
public IIOMetadata getDefaultStreamMetadata(ImageWriteParam param) {
      return null;
}

public IIOMetadata
      getDefaultImageMetadata(ImageTypeSpecifier imageType,
                              ImageWriteParam param) {
      return new MyFormatMetadata();
}

public IIOMetadata convertStreamMetadata(IIOMetadata inData,
                                         ImageWriteParam param) {
      return null;
}

public IIOMetadata convertImageMetadata(IIOMetadata inData,
                                        ImageTypeSpecifier imageType,
                                        ImageWriteParam param) {
      // We only understand our own metadata
      if (inData instanceof MyFormatMetadata) {
            return inData;
      } else {
            return null;
      }
}
```

The actual writing of the image requires first applying the source region, source bands, and subsampling factors from the ImageWriteParam. The source region and source bands may be handled by creating a child Raster. For simplicity, we extract a single Raster from the source image. If the source image is tiled, we can save memory by extracting smaller Rasters as needed.

```
public void write(IIOMetadata streamMetadata,
                  IIOImage image,
```

```
            ImageWriteParam param) throws IIOException {
RenderedImage im = image.getRenderedImage();

Rectangle sourceRegion =
     new Rectangle(0, 0, im.getWidth(), im.getHeight());
int sourceXSubsampling = 1;
int sourceYSubsampling = 1;
int[] sourceBands = null;
if (param != null) {
     sourceRegion =
          sourceRegion.intersection(param.getSourceRegion());
     sourceXSubsampling = param.getSourceXSubsampling();
     sourceYSubsampling = param.getSourceYSubsampling();
     sourceBands = param.getSourceBands();

     int subsampleXOffset = param.getSubsamplingXOffset();
     int subsampleYOffset = param.getSubsamplingYOffset();
     sourceRegion.x += subsampleXOffset;
     sourceRegion.y += subsampleYOffset;
     sourceRegion.width -= subsampleXOffset;
     sourceRegion.height -= subsampleYOffset;
}

// Grab a Raster containing the region of interest
int width = sourceRegion.width;
int height = sourceRegion.height;
Raster imRas = im.getData(sourceRegion);
int numBands = imRas.getNumBands();

// Check that sourceBands values are in range
if (sourceBands != null) {
     for (int i = 0; i < sourceBands.length; i++) {
          if (sourceBands[i] >= numBands) {
               throw new IllegalArgumentException("bad band!");
          }
     }
}

// Translate imRas to start at (0, 0) and subset the bands
imRas = imRas.createChild(sourceRegion.x, sourceRegion.y,
                          width, height,
                          0, 0,
                          sourceBands);

// Reduce width and height according to subsampling factors
width = (width + sourceXSubsampling - 1)/sourceXSubsampling;
height = (height + sourceYSubsampling - 1)/sourceYSubsampling;

// Assume 1 band image is grayscale, 3 band image is RGB
int colorType;
if (numBands == 1) {
     colorType = MyFormatImageReader.COLOR_TYPE_GRAY;
} else if (numBands == 3) {
```

```
            colorType = MyFormatImageReader.COLOR_TYPE_RGB;
        } else {
            throw new IIOException("Image must have 1 or 3 bands!");
        }
```

Once the image dimensions and color type of the image have been ascertained, the plug-in is ready to write the file header:

```
        try {
            byte[] signature = {
                (byte)'m', (byte)'y',  (byte)'f', (byte)'o',
                (byte)'r', (byte)'m',  (byte)'a', (byte)'t'
            };
            // Output header information
            stream.write(signature);
            stream.write('\n');
            stream.writeInt(width);
            stream.writeInt(height);
            stream.writeByte(colorType);
            stream.write('\n');
```

Next, the plug-in extracts the image metadata from the `write` method's `IIOImage` argument, and attempts to convert it into a `MyFormatMetadata` object by calling `convertImageMetadata`. If the result is non-`null`, the keywords and values are extracted from the metadata and written to the output:

```
            // Attempt to convert metadata, if present
            IIOMetadata imd = image.getMetadata();
            MyFormatMetadata metadata = null;
            if (imd != null) {
                ImageTypeSpecifier type =
                    ImageTypeSpecifier.createFromRenderedImage(im);
                metadata =
                    (MyFormatMetadata)convertImageMetadata(imd,
                                                           type,
                                                           null);
            }

            // Output metadata if present
            if (metadata != null) {
                Iterator keywordIter = metadata.keywords.iterator();
                Iterator valueIter = metadata.values.iterator();
                while (keywordIter.hasNext()) {
                    String keyword = (String)keywordIter.next();
                    String value = (String)valueIter.next();

                    stream.writeUTF(keyword);
                    stream.write('\n');
                    stream.writeUTF(value);
                    stream.write('\n');
                }
            }
```

```
                stream.writeUTF("END");
                stream.write('\n');
```

Finally, the plug-in is ready to begin writing the pixel data. The image `Raster` is copied into an int array, one row at a time using the `getPixels` method. Then these values are subsampled using the horizontal subsampling factor, and copied into a byte array, which is written to the output with a single write call. The source row is then incremented by the vertical subsampling factor until the end of the source region is reached, and the output stream is flushed:

```
                // Output (subsampled) pixel values
                int rowLength = width*numBands;
                int xSkip = sourceXSubsampling*numBands;
                int[] rowPixels = imRas.getPixels(0, 0, width, 1,
                                                  (int[])null);
                byte[] rowSamples = new byte[rowLength];

                // Output every (sourceYSubsampling)^th row
                for (int y = 0; y < height; y += sourceYSubsampling) {
                    imRas.getPixels(0, y, width, 1, rowPixels);

                    // Subsample horizontally and convert to bytes
                    int count = 0;
                    for (int x = 0; x < width; x += xSkip) {
                        if (colorType ==
                            MyFormatImageReader.COLOR_TYPE_GRAY) {
                            rowSamples[count++] = (byte)rowPixels[x];
                        } else {
                            rowSamples[count++] = (byte)rowPixels[x];
                            rowSamples[count++] =
                                (byte)rowPixels[x + 1];
                            rowSamples[count++] =
                                (byte)rowPixels[x + 2];
                        }
                    }

                    // Output a row's worth of bytes
                    stream.write(rowSamples, 0, width*numBands);
                }
                stream.flush();
            } catch (IOException e) {
                throw new IIOException("I/O error!", e);
            }
        }
}
```

4.5 Writing Transcoder Plug-ins

A transcoder plug-in consists of an ImageTranscoderSpi, which performs the same functions as do the Spi classes for other plug-ins, and an object that implements the ImageTranscoder interface:

```
IIOMetadata convertStreamMetadata(IIOMetadata inData,
                                  ImageWriteParam param);

IIOMetadata convertImageMetadata(IIOMetadata inData,
                                 ImageTypeSpecifier imageType,
                                 ImageWriteParam param);
```

The ImageTranscoder may use the standard interfaces to unpack the incoming metadata, or it may make use of interfaces that are specific to the actual object at hand. For example, it could access the keywords and values instance variables of the MyFormatMetadata class defined above; these were made public, but not documented, precisely in order to allow a transcoder plug-in developer to access them without having to go through a DOM representation.

4.6 Writing Stream Plug-ins

The standard classes in the javax.imageio.stream package are able to create ImageInputStreams and ImageOutputStreams that:

- Read from a File
- Read from an InputStream, using a temporary File as a cache
- Read from an InputStream, using a byte array in memory as a cache
- Write to a File
- Write to an OutputStream, using a temporary File as a cache
- Write to an OutputStream, using a byte array in memory as a cache

It is possible to write new implementations of ImageInputStream and InmageOutputStream that interface with other I/O primitives (for example, databases or future fast I/O interfaces). The ImageInputStreamImpl and ImageOutputStreamImpl classes provide a base for subclassing. Only a handful of methods are required to be implemented by the subclass, although it is possible to implement more for greater efficiency:

- int read()
- int read(byte[] b, int off, int len)

- write(int b)
- write(byte[] b, int off, int len)
- seek(long pos)