# *Programming in Java™ Advanced Imaging*

Release 1.0.1
November 1999

JavaSoft
A Sun Microsystems, Inc. Business
901 San Antonio Road
Palo Alto, CA 94303 USA
415 960-1300      fax 415 969-9131

# Contents

# Figures

# Preface

THIS document introduces the Java™ Advanced Imaging API and how to program in it. This document is intended for serious programmers who want to use Java Advanced Imaging for real projects. To best understand this document and the examples, you need a solid background in the Java programming language and some experience with imaging. In addition, you will need a working knowledge of other Java Extension APIs, depending on your intended application:

- Java 2D for simple graphics, text, and fundamental image manipulation
- Java Media Framework for components to play and control time-based media such as audio and video
- Java Sound
- Java 3D
- Java Telephony
- Java Speech

## Disclaimer

This version of *Programming in Java Advanced Imaging* is based on release 1.0.1 of the Java Advanced Imaging API. Please do not rely on this document or the Java Advanced Imaging software for production-quality or mission-critical applications. If any discrepancies between this book and the javadocs are noted, always consider the javadocs to be the most accurate, since they are generated directly from the JAI files and are always the most up to date.

## About This Book

**Chapter 1, "Introduction to Java Advanced Imaging,"** gives an overview of the Java Advanced Imaging API, how it evolved from the original Java Advanced

Windowing Toolkit (AWT), some of its features, and introduces the imaging operations.

**Chapter 2, "Java AWT Imaging,"** reviews the imaging portions of the Java AWT and examines the imaging features of the Java 2D API.

**Chapter 3, "Programming in Java Advanced Imaging,"** describes how to get started programming with the Java Advanced Imaging API.

**Chapter 4, "Image Acquisition and Display,"** describes the Java Advanced Imaging API image data types and the API constructors and methods for image acquisition and display.

**Chapter 5, "Color Space,"** describes the JAI color space, transparency, and the color conversion operators.

**Chapter 6, "Image Manipulation,"** describes the basics of manipulating images to prepare them for processing and display.

**Chapter 7, "Image Enhancement,"** describes the basics of improving the visual appearance of images through enhancement techniques.

**Chapter 8, "Geometric Image Manipulation,"** describes the basics of Java Advanced Imaging's geometric image manipulation functions.

**Chapter 9, "Image Analysis,"** describes the Java Advanced Imaging API image analysis operators.

**Chapter 10, "Graphics Rendering,"** describes the Java Advanced Imaging presentation of shapes and text.

**Chapter 11, "Image Properties,"** describes the tools that allow a programmer to add a simple database of arbitrary data that can be attached to images.

**Chapter 12, "Client-Server Imaging,"** describes Java Advanced Imaging's client-server imaging system.

**Chapter 13, "Writing Image Files,"** describes Java Advanced Imaging's codec system for encoding image data files.

**Chapter 14, "Extending the API,"** describes how the Java Advanced Imaging API is extended.

**Appendix A, "Program Examples,"** contains fully-operational Java Advanced Imaging program examples.

**Appendix B, "Java Advanced Imaging API Summary,"** summarizes the imaging interfaces, and classes, including the `java.awt`, `java.awt.Image`, and `javax.media.jai` classes.

The **Glossary** contains descriptions of significant terms that appear in this book.

## Related Documentation

To obtain a good understanding of the Java programming language, we suggest you start with the SunSoft Press series of books:

- *Instant Java*, by John A. Pew
- *Java in a Nutchell: A Desktop Quick Reference*, by David Flanagan
- *Java by Example*, by Jerry R. Jackson and Alan L. McClellan
- *Just Java*, by Peter van der Linden
- *Core Java*, by Gary Cornell and Gay S. Horstmann
- *Java Distributed Computing*, by Jim Farley

For more information on digital imaging, we suggest you refer to the following books:

- *Fundamentals of Digital Image Processing*, by Anil K. Jain
- *Digital Image Processing: Principles and Applications*, by Gregory A. Baxes
- *Digital Image Processing*, by Kenneth R. Castleman
- *Digital Image Processing*, 2nd. ed., by William K. Pratt

## Additional Information

Since Java Advanced Imaging continues to evolve and periodically add new operators, it is always a good idea to occasionally check the JavaSoft JAI web site for the latest information.

```
http://java.sun.com/products/java-media/jai/
```

The web site contains links to the latest version of JAI, email aliases for obtaining help from the community of JAI developers, and a tutorial that includes examples of the use of many JAI operators.

## Style Conventions

The following style conventions are used in this document:

- `Lucida type` is used to represent computer code and the names of files and directories.
- **`Bold Lucida type`** is used for Java 3D API declarations.
- *Italic type* is used for emphasis and for equations.

Throughout the book, we introduce many API calls with the following format:

**API:** `javax.media.jai.TiledImage`

When introducing an API call for the first time, we add a short summary of the methods, tagged with the API heading.

CHAPTER 1

# Introduction to Java Advanced Imaging

THE Java™ programming language has continued to grow both in popularity and scope since its initial release. Java in its current form is the culmination of several years work, dating back to 1991 when it was conceived as a modular and extensible programming language.

Java is based on the C and C++ programming languages, but differs from these languages is some important ways. The main difference between C/C++ and Java is that in Java all development is done with objects and classes. This main difference provides distinct advantages for programs written in Java, such as multiple threads of control and dynamic loading.

Another advantage to Java is its extensibility. Since the original release of Java, several extensions have been added to the core code, providing greater flexibility and power to applications. These extensions add objects and classes that improve the Java programmer's ability to use such features as:

- Java Swing – a component set to create grapical user interfaces with a cross-platform look and feel
- Java Sound – for high-quality 32-channel audio rendering and MIDI-controlled sound synthesis
- Java 3D – for advanced geometry and 3D spatial sound
- Java Media Framework – for components to play and control time-based media such as audio and video
- Java Telephony (JTAPI) – for computer-telephony applications
- Java Speech – for including speech technology into Java applets and applications

## 1.1 The Evolution of Imaging in Java

Early versions of the Java AWT provided a simple rendering package suitable for rendering common HTML pages, but without the features necessary for complex imaging. The early AWT allowed the generation of simple images by drawing lines and shapes. A very limited number of image files, such as GIF and JPEG, could be read in through the use of a `Toolkit` object. Once read in, the image could be displayed, but there were essentially no image processing operators.

The Java 2D API extended the early AWT by adding support for more general graphics and rendering operations. Java 2D added special graphics classes for the definition of geometric primitives, text layout and font definition, color spaces, and image rendering. The new classes supported a limited set of image processing operators for blurring, geometric transformation, sharpening, contrast enhancement, and thresholding. The Java 2D extensions were added to the core Java AWT beginning with the Java Platform 1.2 release.

The Java Advanced Imaging (JAI) API further extends the Java platform (including the Java 2D API) by allowing sophisticated, high-performance image processing to be incorporated into Java applets and applications. JAI is a set of classes providing imaging functionality beyond that of Java 2D and the Java Foundation classes, though it is compatible with those APIs.

JAI implements a set of core image processing capabilities including image tiling, regions of interest, and deferred execution. JAI also offers a set of core image processing operators including many common point, area, and frequency-domain operators.

JAI is intended to meet the needs of all imaging applications. The API is highly extensible, allowing new image processing operations to be added in such a way as to appear to be a native part of it. Thus, JAI benefits virtually all Java developers who want to incorporate imaging into their applets and applications.

## 1.2 Why Another Imaging API?

Several imaging APIs have been developed – a few have even been marketed and been fairly successful. However, none of these APIs have been universally accepted because they failed to address specific segments of the imaging market or they lacked the power to meet specific needs. As a consequence, many companies have had to “roll their own” in an attempt to meet their specific requirements.

Writing a custom imaging API is a very expensive and time-consuming task and the customized API often has to be rewritten whenever a new CPU or operating system comes along, creating a maintenance nightmare. How much simpler it would be to have an imaging API that meets everyone's needs.

Previous industry and academic experience in the design of image processing libraries, the usefulness of these libraries across a wide variety of application domains, and the feedback from the users of these libraries have been incorporated into the design of JAI.

JAI is intended to support image processing using the Java programming language as generally as possible so that few, if any, image processing applications are beyond its reach. At the same time, JAI presents a simple programming model that can be readily used in applications without a tremendous mechanical programming overhead or a requirement that the programmer be expert in all phases of the API's design.

JAI encapsulates image data formats and remote method invocations within a re-usable image data object, allowing an image file, a network image object, or a real-time data stream to be processed identically. Thus, JAI represents a simple programming model while concealing the complexity of the internal mechanisms.

## 1.3 JAI Features

JAI is intended to meet the requirements of all of the different imaging markets, and more. JAI offers several advantages for applications developers compared to other imaging solutions. Some of these advantages are described in the following paragraphs.

### 1.3.1 Cross-platform Imaging

Whereas most imaging APIs are designed for one specific operating system, JAI follows the Java run time library model, providing platform independence. Implementations of JAI applications will run on any computer where there is a Java Virtual Machine. This makes JAI a true cross-platform imaging API, providing a standard interface to the imaging capabilities of a platform. This means that you write your application once and it will run anywhere.

### 1.3.2 Distributed Imaging

JAI is also well suited for client-server imaging by way of the Java platform's networking architecture and remote execution technologies. Remote execution is based on Java RMI (remote method invocation). Java RMI allows Java code on a client to invoke method calls on objects that reside on another computer without having to move those objects to the client.

### 1.3.3 Object-oriented API

Like Java itself, JAI is totally object-oriented. In JAI, images and image processing operations are defined as objects. JAI unifies the notions of image and operator by making both subclasses of a common parent.

An operator object is instantiated with one or more image sources and other parameters. This operator object may then become an image source for the next operator object. The connections between the objects define the flow of processed data. The resulting editable graphs of image processing operations may be defined and instantiated as needed.

### 1.3.4 Flexible and Extensible

Any imaging API must support certain basic imaging technologies, such as image acquisition and display, basic manipulation, enhancement, geometric manipulation, and analysis. JAI provides a core set of the operators required to support the basic imaging technologies. These operators support many of the functions required of an imaging application. However, some applications require special image processing operations that are seldom, if ever, required by other applications. For these specialized applications, JAI provides an extensible framework that allows customized solutions to be added to the core API.

JAI also provides a standard set of image compression and decompression methods. The core set is based on international standards for the most common compressed file types. As with special image processing functions, some applications also require certain types of compressed image files. It is beyond the scope of any API to support the hundreds of known compression algorithms, so JAI also supports the addition of customized coders and decoders (codecs), which can be added to the core API.

### 1.3.5 Device Independent

The processing of images can be specified in device-independent coordinates, with the ultimate translation to pixels being specified as needed at run time. JAI's “renderable” mode treats all image sources as rendering-independent. You can set up a graph (or chain) of renderable operations without any concern for the source image resolution or size; JAI takes care of the details of the operations.

To make it possible to develop platform-independent applications, JAI makes no assumptions about output device resolution, color space, or color model. Nor does the API assume a particular file format. Image files may be acquired and manipulated without the programmer having any knowledge of the file format being acquired.

### 1.3.6 Powerful

JAI supports complex image formats, including images of up to three dimensions and an arbitrary number of bands. Many classes of imaging algorithms are supported directly, others may be added as needed.

JAI implements a set of core image processing capabilities, including image tiling, regions of interest, and deferred execution. The API also implements a set of core image processing operators, including many common point, area, and frequency-domain operations. For a list of the available operators, see Section 3.6, “JAI API Operators.”

### 1.3.7 High Performance

A variety of implementations are possible, including highly-optimized implementations that can take advantage of hardware acceleration and the media capabilities of the platform, such as MMX on Intel processors and VIS on UltraSparc.

### 1.3.8 Interoperable

JAI is integrated with the rest of the Java Media APIs, enabling media-rich applications to be deployed on the Java platform. JAI works well with other Java APIs, such as Java 3D and Java component technologies. This allows sophisticated imaging to be a part of every Java technology programmer’s tool box.

JAI is a Java Media API. It is classified as a Standard Extension to the Java platform. JAI provides imaging functionality beyond that of the Java Foundation Classes, although it is compatible with those classes in most cases.

## 1.4 A Simple JAI Program

Before proceeding any further, let’s take a look at an example JAI program to get an idea of what it looks like. Listing 1-1 shows a simple example of a complete JAI program. This example reads an image, passed to the program as a command line argument, scales the image by 2× with bilinear interpolation, then displays the result.

**Listing 1-1 Simple Example JAI Program**

```
import java.awt.Frame;
import java.awt.image.renderable.ParameterBlock;
import java.io.IOException;
import javax.media.jai.Interpolation;
import javax.media.jai.JAI;
import javax.media.jai.RenderedOp;
import com.sun.media.jai.codec.FileSeekableStream;
import javax.media.jai.widget.ScrollingImagePanel;

/**
 * This program decodes an image file of any JAI supported
 * formats, such as GIF, JPEG, TIFF, BMP, PNM, PNG, into a
 * RenderedImage, scales the image by 2X with bilinear
 * interpolation, and then displays the result of the scale
 * operation.
 */
public class JAISampleProgram {

    /** The main method. */
    public static void main(String[] args) {
        /* Validate input. */
        if (args.length != 1) {
           System.out.println(“Usage: java JAISampleProgram “ +
                                “input_image_filename”);
            System.exit(-1);
        }

        /*
         * Create an input stream from the specified file name
         * to be used with the file decoding operator.
         */
        FileSeekableStream stream = null;
        try {
            stream = new FileSeekableStream(args[0]);
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(0);
        }
```

**Listing 1-1 Simple Example JAI Program (Continued)**

```
           /* Create an operator to decode the image file. */
           RenderedOp image1 = JAI.create("stream", stream);
           /*
            * Create a standard bilinear interpolation object to be
            * used with the "scale" operator.
            */
           Interpolation interp = Interpolation.getInstance(
                                    Interpolation.INTERP_BILINEAR);

           /**
            * Stores the required input source and parameters in a
            * ParameterBlock to be sent to the operation registry,
            * and eventually to the "scale" operator.
            */
           ParameterBlock params = new ParameterBlock();
           params.addSource(image1);
           params.add(2.0F);          // x scale factor
           params.add(2.0F);          // y scale factor
           params.add(0.0F);          // x translate
           params.add(0.0F);          // y translate
           params.add(interp);        // interpolation method

           /* Create an operator to scale image1. */
           RenderedOp image2 = JAI.create("scale", params);

           /* Get the width and height of image2. */
           int width = image2.getWidth();
           int height = image2.getHeight();

          /* Attach image2 to a scrolling panel to be displayed. */
           ScrollingImagePanel panel = new ScrollingImagePanel(
                                        image2, width, height);

           /* Create a frame to contain the panel. */
           Frame window = new Frame("JAI Sample Program");
           window.add(panel);
           window.pack();
           window.show();
       }
   }
```

CHAPTER 2

# Java AWT Imaging

DIGITAL imaging in Java has been supported since its first release, through the **`java.awt`** and **`java.awt.image`** class packages. The image-oriented part of these class packages is referred to as *AWT Imaging* throughout this guide.

## 2.1 Introduction

The Java Advanced Imaging (JAI) API supports three imaging models:

- The producer/consumer (push) model – the basic AWT imaging model
- The immediate mode model – an advanced AWT imaging model
- The pipeline (pull) model – The JAI model

Table 2-1 lists the interfaces and classes for each of the three models.

**Table 2-1 Imaging Model Interfaces and Classes**

| AWT Push Model | Java 2D Immediate Mode Model | Pull Model |
|---|---|---|
| `Image` | `BufferedImage` | `RenderableImage` |
| `ImageProducer` | `Raster` | `RenderableImageOp` |
| `ImageConsumer` | `BufferedImageOp` | `RenderedOp` |
| `ImageObserver` | `RasterOp` | `RenderableOp` |
| | | `TiledImage` |

### 2.1.1 The AWT Push Model

The AWT push model, supported through the `java.awt` class package, is a simple filter model of image producers and consumers for image processing. An `Image` object is an abstraction that is not manipulated directly; rather it is used to obtain a reference to another object that implements the `ImageProducer` interface. Objects that implement this interface are in turn attached to objects that implement the ImageConsumer interface. Filter objects implement both the producer and consumer interfaces and can thus serve as both a source and sink of image data. Image data has associated with it a ColorModel that describes the pixel layout within the image and the interpretation of the data.

To process images in the push model, an Image object is obtained from some source (for example, through the Applet.getImage() method). The Image.getSource() method can then be used to get the ImageProducer for that Image. A series of FilteredImageSource objects can then be attached to the ImageProducer, with each filter being an ImageConsumer of the previous image source. AWT Imaging defines a few simple filters for image cropping and color channel manipulation.

The ultimate destination for a filtered image is an AWT Image object, created by a call to, for example, Component.createImage(). Once this consumer image has been created, it can by drawn upon the screen by calling Image.getGraphics() to obtain a Graphics object (such as a screen device), followed by Graphics.drawImage().

AWT Imaging was largely designed to facilitate the display of images in a browser environment. In this context, an image resides somewhere on the network. There is no guarantee that the image will be available when required, so the AWT model does not force image filtering or display to completion. The model is entirely a *push* model. An ImageConsumer can never ask for data; it must wait for the ImageProducer to “push” the data to it. Similarly, an ImageConsumer has no guarantee about when the data will be completely delivered; it must wait for a call to its ImageComplete() method to know that it has the complete image. An application can also instantiate an ImageObserver object if it wishes to be notified about completion of imaging operations.

AWT Imaging does not incorporate the idea of an image that is backed by a persistent image store. While methods are provided to convert an input memory array into an ImageProducer, or capture an output memory array from an ImageProducer, there is no notion of a persistent image object that can be reused. When data is wanted from an Image, the programmer must retrieve a handle to the Image’s ImageProducer to obtain it.

The AWT imaging model is not amenable to the development of high-performance image processing code. The push model, the lack of a persistent image data object, the restricted model of an image filter, and the relative paucity of image data formats are all severe constraints. AWT Imaging also lacks a number of common concepts that are often used in image processing, such as operations performed on a region of interest in an image.

### 2.1.2 AWT Push Model Interfaces and Classes

The following are the Java interfaces and classes associated with the AWT push model of imaging.

**Table 2-2 Push Model Imaging Interfaces**

| Interface | Description |
|---|---|
| `Image` | Extends: `Object`<br>The superclass of all classes that represent graphical images. |

**Table 2-3 Push Model Imaging Classes**

| Class | Description |
|---|---|
| `ColorModel` | An abstract class that encapsulates the methods for translating a pixel value to color components (e.g., red, green, blue) and an alpha component. |
| `FilteredImageSource` | An implementation of the `ImageProducer` interface which takes an existing image and a filter object and uses them to produce image data for a new filtered version of the original image. |
| `ImageProducer` | The interface for objects that can produce the image data for Images. Each image contains an `ImageProducer` that is used to reconstruct the image whenever it is needed, for example, when a new size of the `Image` is scaled, or when the width or height of the `Image` is being requested. |
| `ImageConsumer` | The interface for objects expressing interest in image data through the `ImageProducer` interfaces. When a consumer is added to an image producer, the producer delivers all of the data about the image using the method calls defined in this interface. |
| `ImageObserver` | An asynchronous update interface for receiving notifications about `Image` information as the `Image` is constructed. |

## 2.2 The Immediate Mode Model

To alleviate some of the restrictions of the original AWT imaging model and to provide a higher level of abstraction, a new specification called the *Java 2D* API was developed. This new API extends AWT's capabilities for both two-dimensional graphics and imaging. In practice, the Java 2D package is now merged into the AWT specification and is a part of the Java Core (and thus available in all Java implementations). However, for purposes of discussion, the distinction between Java 2D and the AWT is preserved in this chapter.

The Java 2D API specifies a set of classes that extend the Java AWT classes to provide extensive support for both two-dimensional graphics and imaging. The support for 2D graphics is fairly complete, but will not be discussed further here.

For digital imaging, the Java 2D API retains to some extent the AWT producer/consumer model but adds the concept of a memory-backed persistent image data object, an extensible set of 2D image filters, a wide variety of image data formats and color models, and a more sophisticated representation of output devices. The Java 2D API also introduces the notion of resolution-independent image rendering by the introduction of the *Renderable* and *Rendered* interfaces, allowing images to be pulled through a chain of filter operations, with the image resolution selected through a rendering context.

The concepts of rendered and renderable images contained in the Java 2D API are essential to JAI. The next few sections explain these concepts; complete information about the classes discussed can be found in *The Java 2D API Specification* and the *Java 2D API White Paper.*

### 2.2.1 Rendering Independence

Rendering independence for images is a poorly understood topic because it is poorly named. The more general problem is “resolution independence,” the ability to describe an image as you want it to appear, but independent of any specific instance of it. Resolution is but one feature of any such rendering. Others are the physical size, output device type, color quality, tonal quality, and rendering speed. A rendering-independent description is concerned with none of these.

In this document, the term *rendering-independent* is for the more general concept instead of *resolution-independent*. The latter term is used to specifically refer to independence from final display resolution.

For a rendering-independent description of an image, two fundamental elements are needed:

- An unrendered source (sometimes called a *resolution-independent source*). For a still image, this is, conceptually, the viewfinder of an idealized camera trained on a real scene. It has no logical “size.” Rather, one knows what it looks like and can imagine projecting it onto any surface. Furthermore, the ideal camera has an ideal lens that is capable of infinite zooming. The characteristics of this image are that it is dimensional, has a native aspect ratio (that of the capture device), and may have properties that could be queried.
- Operators for describing how to change the character of the image, independent of its final destination. It can be useful to think of this as a pipe of operations.

Together, the unrendered source and the operators specify the visual character that the image should have when it is rendered. This specification can then be associated with any device, display size, or rendering quality. The primary power of rendering independence is that the same visual description can be routed to any display context with an optimal result.

### 2.2.2 Rendering-independent Imaging in Java AWT

The Java AWT API architecture integrates a model of rendering independence with a parallel, device-dependent (rendered) model. The rendering-independent portion of the architecture is a superset of, rather than a replacement for, the traditional model of device-dependent imaging.

The Java AWT API architecture supports context-dependent adaptation, which is superior to full image production and processing. Context-dependent adaptation is inherently more efficient and thus also suited to network sources. Beyond efficiency, it is the mechanism by which optimal image quality can be assured in any context.

The Java AWT API architecture is essentially synchronous is nature. This has several advantages, such as a simplified programming model and explicit controls on the type and order of results. However, the synchronous nature of Java AWT has one distinct disadvantage in that it is not well suited to notions of progressive rendering or network resources. These issues are addressed in JAI.

### 2.2.3 The Renderable Layer vs. the Rendered Layer

The Java AWT API architecture provides for two integrated imaging layers: renderable and rendered.

#### 2.2.3.1 Renderable Layer

The renderable layer is a rendering-independent layer. All the interfaces and classes in the Java AWT API have `renderable` in their names.

The renderable layer provides image sources that can be optimally reused multiple times in different contexts, such as screen display or printing. The renderable layer also provides imaging operators that take rendering-independent parameters. These operators can be linked to form *chains*. The layer is essentially synchronous in the sense that it “pulls” the image through the chain whenever a rendering (such as to a display or a file) is requested. That is, a request is made at the sink end of the chain that is passed up the chain to the source. Such requests are context-specific (such as device specific), and the chain adapts to the context. Only the data required for the context is produced.

### 2.2.3.2 Rendered Layer

Image sources and operators in the parallel *Rendered layer* (the interfaces and classes have `rendered` in their names) are context-specific. A `RenderedImage` is an image that has been rendered to fulfill the needs of the context. Rendered layer operators can also be linked together to form chains. They take context-dependent parameters. Like the Renderable layer, the Rendered layer implements a synchronous "pull" model.

### 2.2.3.3 Using the Layers

Structurally, the Renderable layer is lightweight. It does not directly handle pixel processing. Rather, it makes use of operator objects from the Rendered layer. This is possible because the operator classes from the Rendered layer can implement an interface (the `ContextualRenderedImageFactory` interface) that allows them to adapt to different contexts.

Since the Rendered layer operators implement this interface, they house specific operations in their entirety. That is, all the intelligence required to function in both the Rendered and Renderable layers is housed in a single class. This simplifies the task of writing new operators and makes extension of the architecture manageable.

Figure 2-1 shows a renderable chain. The chain has a sink attached (a Graphics2D object), but no pixels flow through the chain yet.

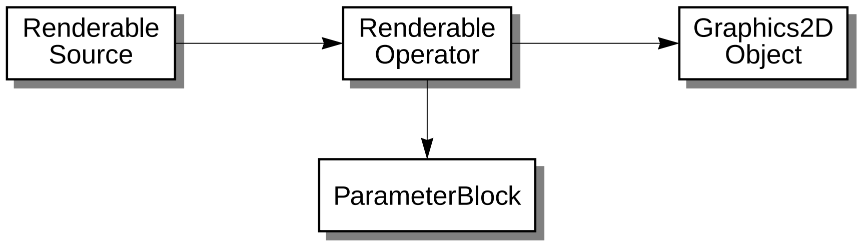


**Figure 2-1 A Renderable Chain**

You may use either the Renderable or Rendered layer to construct an application. Many programmers will directly employ the Rendered layer, but the Renderable layer provides advantages that greatly simplify imaging tasks. For example, a chain of Renderable operators remains editable. Parameters used to construct the chain can be modified repeatedly. Doing so does not cause pixel value computation to occur. Instead, the pixels are computed only when they are needed by a specific rendition obtained from a `RenderableImage` by passing it defined *render contexts*.

### 2.2.4 The Render Context

The renderable layer allows for the construction of a chain of operators (`RenderableImageOps`) connected to a `RenderableImage` source. The end of this chain represents a new `RenderableImage` source. The implication of this is that `RenderableImageOps` must implement the same interface as sources: `RenderableImageOp` implements `RenderableImage`.

Such a source can be asked to provide various specific `RenderedImages` corresponding to a specific context. The required size of the `RenderedImage` in the device space (the size in pixels) must be specified. This information is provided in the form of an affine transformation from the user space of the Renderable source to the desired device space.

Other information can also be provided to the source (or chain) to help it perform optimally for a specific context. A preference for speed over image quality is an example. Such information is provided in the form of an extensible hints table. It may also be useful to provide a means to limit the request to a specific area of the image.

The architecture refers to these parameters collectively as a *render context*. The parameters are housed in a `RenderContext` class. Render contexts form a fundamental link between the Renderable and Rendered layers. A `RenderableImage` source is given a `RenderContext` and, as a result, produces a specific rendering, or `RenderedImage`. This is accomplished by the Renderable chain instantiating a chain of Render layer objects. That is, a chain of `RenderedImages` corresponding to the specific context, the `RenderedImage` object at the end of the chain being returned to the user.

## 2.3 Renderable and Rendered Classes

Many users will be able to employ the Renderable layer, with the advantages of its rendering-independent properties for most imaging purposes. Doing so eliminates the need to deal directly with pixels, greatly simplifying image manipulation. However, in many cases it is either necessary or desirable to work with pixels and the Rendered layer is used for this purpose.

The architecture of the provided classes is discussed in this section. Extending the model by writing new operators or algorithms in the Java 2D API is discussed. Details of how the Rendered layer functions internally within the Renderable layer are also covered.

### 2.3.1 The Renderable Layer

The renderable layer is primarily defined by the `RenderableImage` interface. Any class implementing this interface is a renderable image source, and is expected to adapt to `RenderContexts`. `RenderableImages` are referenced through a user-defined coordinate system. One of the primary functions of the `RenderContext` is to define the mapping between this user space and the specific device space for the desired rendering.

A chain in this layer is a chain of `RenderableImages`. Specifically, it is a chain of `RenderableImageOps` (a class that implements `RenderableImage`), ultimately sourced by a `RenderableImage`.

There is only one `RenderableImageOp` class. It is a lightweight, general purpose class that takes on the functionality of a specific operation through a parameter provided at instantiation time. That parameter is the name of a class that implements a `ContextualRenderedImageFactory` (known as a CRIF, for short). Each instantiation of `RenderableImageOp` derives its specific functionality from the named class. In this way, the Renderable layer is heavily dependent on the Rendered layer.

**Table 2-4 The Renderable Layer Interfaces and Classes**

| Type | Name | Description |
|---|---|---|
| Interface | `RenderableImage` | A common interface for rendering-independent images (a notion that subsumes resolution independence). |
| | `ContextualRenderedImage-Factory` | Extends: `RenderedImageFactory` Provides an interface for the functionality that may differ between instances of `RenderableImageOp`. |

**Table 2-4 The Renderable Layer Interfaces and Classes (Continued)**

| Type | Name | Description |
|---|---|---|
| Class | ParameterBlock | Extends: Object<br>Implements: Cloneable, Serializable<br>Encapsulates all the information about sources and parameters (expressed as base types or Objects) required by a RenderableImageOp and other future classes that manipulate chains of imaging operators. |
| | RenderableImageOp | Extends: Object<br>Implements: RenderableImage<br>Handles the renderable aspects of an operation with help from its associated instance of a ContextualRenderedImageFactory. |
| | RenderableImageProducer | Extends: Object<br>Implements: ImageProducer, Runnable<br>An adapter class that implements ImageProducer to allow the asynchronous production of a RenderableImage. |
| | RenderContext | Extends: Object<br>Implements: Cloneable<br>Encapsulates the information needed to produce a specific rendering from a RenderableImage. |

The other block involved in the construction of RenderableImageOp is a ParameterBlock. The ParameterBlock houses the source(s) for the operation, plus parameters or other objects that the operator may require. The parameters are rendering-independent versions of the parameters that control the (Rendered) operator.

A Renderable chain is constructed by instantiating each successive RenderableImageOp, passing in the last RenderableImage as the source in the ParameterBlock. This chain can then be requested to provide a number of renderings to specific device spaces through the getImage method.

This chain, once constructed, remains editable. Both the parameters for the specific operations in the chain and the very structure of the chain can be changed. This is accomplished by the setParameterBlock method, setting new controlling parameters and/or new sources. These edits only affect future RenderedImages derived from points in the chain below the edits. RenderedImages that were previously obtained from the Renderable chain are immutable and completely independent from the chain from which they were derived.

### 2.3.2 The Rendered Layer

The Rendered layer is designed to work in concert with the Renderable layer. The Rendered layer is comprised of sources and operations for device-specific representations of images or renderings. The Rendered layer is primarily defined by the `RenderedImage` interface. Sources such as `BufferedImage` implement this interface.

Operators in this layer are simply `RenderedImages` that take other `RenderedImages` as sources. Chains, therefore, can be constructed in much the same manner as those of the Renderable layer. A sequence of `RenderedImages` is instantiated, each taking the last `RenderedImage` as a source.

In Figure 2-2, when the user calls `Graphics2D.drawImage()`, a render context is constructed and used to call the `getImage()` method of the renderable operator. A rendered operator to actually do the pixel processing is constructed and attached to the source and sink of the renderable operator and is passed a clone of the renderable operator's parameter block. Pixels actually flow through the rendered operator to the Graphics2D. The renderable operator chain remains available to produce more renderings whenever its `getImage()` method is called.

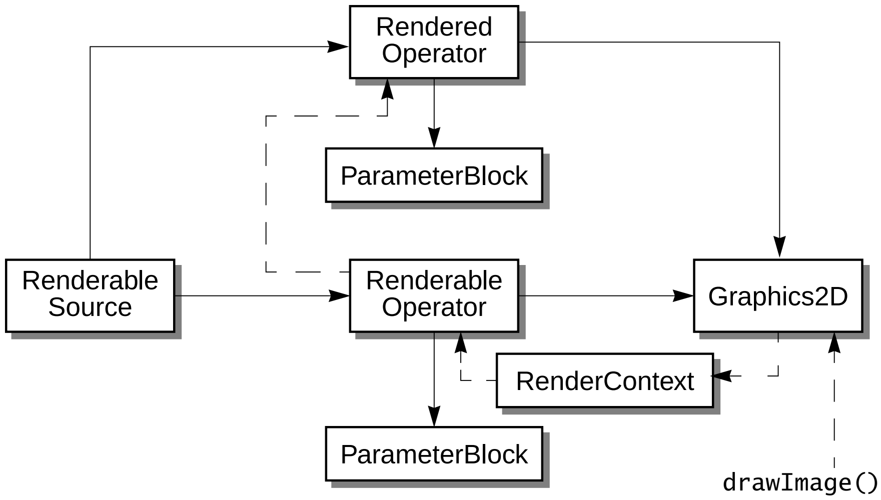


**Figure 2-2 Deriving a Rendering from a Renderable Chain**

**Table 2-5 The Rendered Layer Interfaces and Classes**

| Type | Name | Description |
|---|---|---|
| Interface | `RenderedImage` | A common interface for objects that contain or can produce image data in the form of Rasters. |
| Class | `BufferedImage` | Extends: `Image` Implements: `WritableRenderedImage` A subclass that describes an Image with an accessible buffer of image data. |
| | `WritableRenderedImage` | Extends: `RenderedImage` A common interface for objects that contain or can produce image data that can be modified and/or written over. |

A rendered image represents a virtual image with a coordinate system that maps directly to pixels. A Rendered image does not have to have image data associated with it, only that it be able to produce image data when requested. The `BufferedImage` class, which is the Java 2D API's implementation of `RenderedImage`, however, maintains a full page buffer that can be accessed and written to. Data can be accessed in a variety of ways, each with different properties.

## 2.4 Java Image Data Representation

In the Java AWT API, a sample is the most basic unit of image data. Each pixel is composed of a set of samples. For an RGB pixel, there are three samples; one each for red, green, and blue. All samples of the same kind across all pixels in an image constitute a *band*. For example, in an RGB image, all the red samples together make up a band. Therefore, an RGB image contains three bands.

A three-color subtractive image contains three bands; one each for cyan, magenta, and yellow (CMY). A four-color subtractive image contains four bands; one each for cyan, magenta, yellow, and black (CMYK).

**Table 2-6 Java 2D Image Data Classes**

| Type | Name | Description |
|---|---|---|
| Class | `DataBuffer` | Extends: `Object`<br>Wraps one or more data arrays. Each data array in the `DataBuffer` is referred to as a bank. |
| | `Raster` | Extends: `Object`<br>Represents a rectanglular array of pixels and provides methods for retrieving image data. |
| | `SampleModel` | Extends: `Object`<br>Extracts samples of pixels in images. |
| | `WriteableRaster` | Extends: `Raster`<br>Provides methods for storing image data and inherits methods for retrieving image data from it's parent class `Raster`. |

The basic unit of image data storage is the `DataBuffer`. The `DataBuffer` is a kind of raw storage that contains all of the samples for the image data but does not maintain a notion of how those samples can be put together as pixels. The information about how the samples are put together as pixels is contained in a `SampleModel`. The `SampleModel` class contains methods for deriving pixel data from a `DataBuffer`. Together, a `DataBuffer` and a `SampleModel` constitute a meaningful multi-pixel image storage unit called a `Raster`.

A `Raster` has methods that directly return pixel data for the image data it contains. There are two basic types of `Raster`s:

- `Raster` – a read-only object that has only accessors
- `WritableRaster` – A writable object that has a variety of mutators

There are separate interfaces for dealing with each raster type. The `RenderedImage` interface assumes that the data is read-only and does not contain methods for writing a `Raster`. The `WritableRenderedImage` interface assumes that the image data is writeable and can be modified.

Data from a *tile* is returned in a `Raster` object. A tile is not a class in the architecture; it is a concept. A tile is one of a set of regular rectangular regions that span the image on a regular grid. In the `RenderedImage` interface, there are several methods that relate to tiles and a tile grid. These methods are used by the JAI API, rather than the Java 2D API. In the Java 2D API, the implementation of the `WritableRenderedImage` (`BufferedImage`) is defined to have a single tile. This, the `getWritableTile` method will return all the image data. Other methods that relate to tiling will return the correct degenerative results.

`RenderedImages` do not necessarily maintain a `Raster` internally. Rather, they can return requested rectangles of image data in the form of a (`Writable`)`Raster` (through the `getData`, `getRect`, and `get`(`Writable`)`Tile` methods). This distinction allows `RenderedImages` to be virtual images, producing data only when needed. `RenderedImages` do, however, have an associated `SampleModel`, implying that data returned in `Rasters` from the same image will always be written to the associated `DataBuffer` in the same way.

The Java 2D `BufferedImage` also adds an associated `ColorModel`, which is different from the `SampleModel`. The `ColorModel` determines how the bands are interpreted in a colorimetric sense.

## 2.5 Introducing the Java Advanced Imaging API

The JAI API builds on the foundation of the Java 2D API to allow more powerful and general imaging applications. The JAI API adds the following concepts:

- Multi-tiled images
- Deferred execution
- Networked images
- Image property management
- Image operators with multiple sources
- Three-dimensional image data

The combination of tiling and deferred execution allows for considerable run-time optimization while maintaining a simple imaging model for programmers. New operators may be added and the new operators may participate as first-class objects in the deferred execution model.

The JAI API also provides for a considerable degree of compatibility with the Java AWT and Java 2D imaging models. JAI's operators can work directly on Java 2D `BufferedImage` objects or any other image objects that implement the `RenderedImage` interface. JAI supports the same rendering-independent model as the Java 2D API. using device-independent coordinates. JAI also supports Java 2D-style drawing on both Rendered and Renderable images using the `Graphics` interface.

The JAI API does not make use of the image producer/consumer interfaces introduced in Java AWT and carried forward into the Java 2D API. Instead, the JAI API requires that image sources participate in the "pull" imaging model by responding to requests for arbitrary areas, thus making it impossible to instantiate an `ImageProducer` directly as a source. It is, however, possible to instantiate an `ImageProducer` that makes the JAI API image data available to older AWT applications.

### 2.5.1 Similarities with the Java 2D API

The JAI API is heavily dependent on the abstractions defined in the Java 2D API. In general, the entire mechanism for handling Renderable and Rendered images, pixel samples, and data storage is carried over into JAI. Here are some of the major points of congruity between Java 2D and JAI:

- The `RenderableImage` and `RenderedImage` interfaces defined in the Java 2D API are used as a basis for higher-level abstractions. Further, JAI allows you to create and manipulate directed acyclic graphs of objects implementing these interfaces.
- The primary data object, the `TiledImage`, implements the `WritableRenderedImage` interface and can contain a regular tile grid of `Raster` objects. However, unlike the `BufferedImage` of the Java 2D API, `TiledImage` does not require that a `ColorModel` for photometric interpretation of its image data be present.
- The JAI operator objects are considerably more sophisticated than in the Java 2D API. The `OpImage`, the fundamental operator object, provides considerable support for extensibility to new operators beyone that in the Java 2D API. JAI has a registry mechanism that automates the selection of operations on `RenderedImages`.
- The Java 2D API `SampleModel`, `DataBuffer`, and `Raster` objects are carried over into JAI without change, except that `doubles` and `floats` are allows to be used as the fundamental data types of a `DataBuffer` in addition to the `byte`, `short`, and `int` data types.

### 2.5.2 JAI Data Classes

JAI introduces two new data classes, which extend the Java 2D `DataBuffer` image data class.

**Table 2-7 JAI Data Classes**

| Type | Name | Description |
|---|---|---|
| Class | `DataBufferFloat` | Extends: `DataBuffer`<br>Stores data internally in float form. |
| | `DataBufferDouble` | Extends: `DataBuffer`<br>Stores data internally in double form. |

### 2.5.2.1 The DataBufferFloat Class

**API:** `javax.media.jai.DataBufferFloat`

- `DataBufferFloat(int size)`

  constructs a float-based DataBuffer with a specified size.

  | | | |
  |---|---|---|
  | *Parameters*: | `size` | The number of elements in the `DataBuffer`. |

- `DataBufferFloat(int size, int numBanks)`

  constructs a float-based DataBuffer with a specified number of banks, all of which are of a specified size.

  | | | |
  |---|---|---|
  | *Parameters*: | `size` | The number of elements in each bank of the `DataBuffer`. |
  | | `numBanks` | The number of banks in the `DataBuffer`. |

- `DataBufferFloat(float[] dataArray, int size)`

  constructs a float-based `DataBuffer` with the specified data array. Only the first size elements are available for use by this data buffer. The array must be large enough to hold `size` elements.

  | | | |
  |---|---|---|
  | *Parameters*: | `dataArray` | An array of floats to be used as the first and only bank of this `DataBuffer`. |
  | | `size` | The number of elements of the array to be used. |

- `DataBufferFloat(float[] dataArray, int size, int offset)`

  constructs a float-based `DataBuffer` with the specified data array. Only the elements between `offset` and (`offset` + `size` – 1) are available for use by this `DataBuffer`. The array must be large enough to hold (`offset` + `size`) elements.

  | | | |
  |---|---|---|
  | *Parameters*: | `dataArray` | An array of floats to be used as the first and only bank of this `DataBuffer`. |
  | | `size` | The number of elements of the array to be used. |
  | | `offset` | The offset of the first element of the array that will be used. |

- `DataBufferFloat(float[][] dataArray, int size)`

  constructs a float-based `DataBuffer` with the specified data arrays. Only the first size elements of each array are available for use by this `DataBuffer`. The number of banks will be equal to `dataArray.length`.

| | | |
|---|---|---|
| *Parameters*: | `dataArray` | An array of floats to be used as banks of this `DataBuffer`. |
| | `size` | The number of elements of each array to be used. |

- `DataBufferFloat(float[][] dataArray, int size, int[] offsets)`

  constructs a float-based `DataBuffer` with the specified data arrays, size, and per-bank offsets. The number of banks is equal to `dataArray.length`. Each array must be at least as large as `size` + the corresponding `offset`. There must be an entry in the `offsets` array for each data array.

| | | |
|---|---|---|
| *Parameters*: | `dataArray` | An array of arrays of floats to be used as the banks of this `DataBuffer`. |
| | `size` | The number of elements of each array to be used. |
| | `offset` | An array of integer offsets, one for each bank. |

### 2.5.2.2 The DataBufferDouble Class

**API:** `javax.media.jai.DataBufferDouble`

- `DataBufferDouble(int size)`

  constructs a double-based `DataBuffer` with a specified size.

| | | |
|---|---|---|
| *Parameters*: | `size` | The number of elements in the `DataBuffer`. |

- `DataBufferDouble(int size, int numBanks)`

  constructs a double-based `DataBuffer` with a specified number of banks, all of which are of a specified size.

| | | |
|---|---|---|
| *Parameters*: | `size` | The number of elements in each bank of the `DataBuffer`. |
| | `numBanks` | The number of banks in the `DataBuffer`. |

- `DataBufferDouble(double[] dataArray, int size)`

  constructs a double-based `DataBuffer` with the specified data array. Only the first `size` elements are available for use by this databuffer. The array must be large enough to hold `size` elements.

| | | |
|---|---|---|
| *Parameters*: | `dataArray` | An array of doubles to be used as the first and only bank of this `DataBuffer`. |
| | `size` | The number of elements of the array to be used. |

- `DataBufferDouble(double[] dataArray, int size, int offset)`

  constructs a double-based `DataBuffer` with the specified data array. Only the elements between `offset` and (`offset` + `size` – 1) are available for use by this data buffer. The array must be large enough to hold (`offset` + `size`) elements.

| | | |
|---|---|---|
| *Parameters*: | `dataArray` | An array of doubles to be used as the first and only bank of this `DataBuffer`. |
| | `size` | The number of elements of the array to be used. |
| | `offset` | The offset of the first element of the array that will be used. |

- `DataBufferDouble(double[][] dataArray, int size)`

  constructs a double-based `DataBuffer` with the specified data arrays. Only the first size elements of each array are available for use by this `DataBuffer`. The number of banks will be equal to `dataArray.length`.

| | | |
|---|---|---|
| *Parameters*: | `dataArray` | An array of doubles to be used as banks of this `DataBuffer`. |
| | `size` | The number of elements of each array to be used. |

- `DataBufferDouble(double[][] dataArray, int size, int[] offsets)`

  constructs a double-based `DataBuffer` with the specified data arrays, size, and per-bank offsets. The number of banks is equal to `dataArray.length`. Each array must be at least as large as `size` + the corresponding `offset`. There must be an entry in the offsets array for each data array.

| *Parameters*: | | |
|---|---|---|
| | `dataArray` | An array of arrays of doubles to be used as the banks of this `DataBuffer`. |
| | `size` | The number of elements of each array to be used. |
| | `offset` | An array of integer offsets, one for each bank. |

CHAPTER 3

# Programming in Java Advanced Imaging

THIS chapter describes how to get started programming with the Java Advanced Imaging (JAI) API.

## 3.1 Introduction

An imaging operation within JAI is summarized in the following four steps:

1. Obtain the source image or images. Images may be obtained in one of three ways (see Chapter 4, “Image Acquisition and Display”):
   a. Load from an image file such as GIF, TIFF, or JPEG
   b. Fetch the image from another data source, such as a remote server
   c. Generate the image internally
2. Define the imaging graph. This is a two part process:
   a. Define the image operators (see Section 3.6, “JAI API Operators”)
   b. Define the parent/child relationship between sources and sinks
3. Evaluate the graph using one of three execution models:
   a. Rendered execution model (Immediate mode – see Section 3.3.1, “Rendered Graphs”)
   b. Renderable execution model (Deferred mode – see Section 3.3.2, “Renderable Graphs”)
   c. Remote execution model (Remote mode – see Section 3.4, “Remote Execution”)

4. Process the result. There are four possible destinations:
    a. Save the image in a file
    b. Display the image on the screen
    c. Print the image on a printer or other output device
    d. Send the image to another API, such as Swing

## 3.2 An Overview of Graphs

In JAI, any operation is defined as an object. An operator object is instantiated with zero or more image sources and other parameters that define the operation. Two or more operators may be strung together so that the first operator becomes an image source to the next operator. By linking one operator to another, we create an imaging *graph* or *chain*.

In its simplest form, the imaging graph is a chain of operator objects with one or more image sources at one end and an image *sinc* (or "user") at the other end. The graph that is created is commonly known as a *directed acyclic graph* (DAG), where each object is a *node* in the graph and object references form the *edges* (see Figure 3-1).

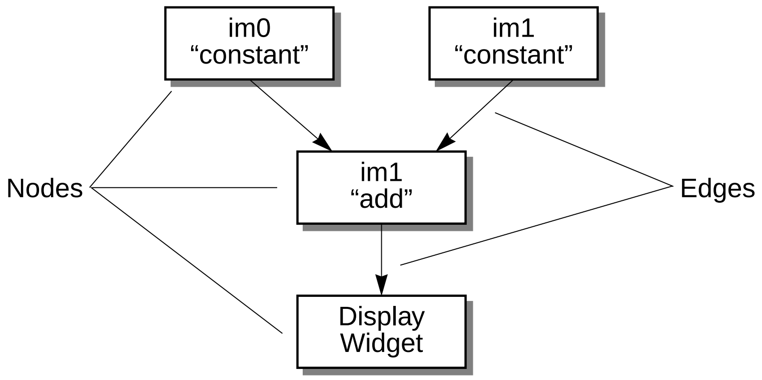


**Figure 3-1 An Example DAG**

Most APIs simply leave the DAG structure of images and operators implicit. However, JAI makes the notion of a *processing graph* explicit and allows such graphs to be considered as entities in their own right. Rather than thinking only of performing a series of operations in sequence, you can consider the graph structure produced by the operations. The graph form makes it easier to visualize the operations.

A directed acyclic graph is a graph containing no cycles. This means that if there is a route from node A to node B then there should be no way back. Normally, when creating a graph by instantiating new nodes one at a time, cycles are easily avoided. However, when reconfiguring a graph, you must be careful not to introduce cycles into the graph.

## 3.3 Processing Graphs

JAI extends rendering independence, which was introduced in the Java 2D API. With rendering independence, you have the ability to describe an image as you want it to appear, independent of any specific instance of it.

In most imaging APIs, the application must know the exact resolution and size of the source image before it can begin any imaging operations on the image. The application must also know the resolution of the output device (computer monitor or color printer) and the color and tonal quality of the original image. A rendering-independent description is concerned with none of these. Rendering-independent sources and operations permit operations to be specified in resolution-independent coordinates.

Think of rendering independence a bit like how a PostScript file is handled in a computer. To display a PostScript file on a monitor or to print the file to a high-resolution phototypesetter, you don’t need to know the resolution of the output device. The PostScript file is essentially rendering independent in that it displays properly no matter what the resolution of the output device is.

JAI has a “renderable” mode in which it treats all image sources as rendering independent. You can set up a graph (or chain) of renderable operations without any concern for the source image resolution or size; JAI takes care of the details of the operations.

JAI introduces two different types of graphs: rendered and renderable.

**Note:** The following two sections, “Rendered Graphs” and “Renderable Graphs,” are for advanced JAI users. Most programmers will use JAI’s Rendered mode and don’t really need to know about the Renderable mode.

### 3.3.1 Rendered Graphs

Rendered graphs are the simplest form of rendering in JAI. Although Renderable graphs have the advantage of rendering-independence, eliminating the need to deal directly with pixels, Rendered graphs are useful when it is necessary to work directly with the pixels.

A Rendered graph processes images in immediate mode. For any node in the graph, the image source is considered to have been evaluated at the moment it is instantiated and added to the graph. Or, put another way, as a new operation is added to the chain, it appears to compute its results immediately.

A Rendered graph is composed of Rendered object nodes. These nodes are usually instances of the `RenderedOp` class, but could belong to any subclass of `PlanarImage`, JAI's version of `RenderedImage`.

Image sources are objects that implement the `RenderedImage` interface. These sources are specified as parameters in the construction of new image objects.

Let's take a look at an example of a rendered graph in Listing 3-1. This example, which is a code fragment rather than an entire class definition, creates two constant images and then adds them together.

**Listing 3-1 Rendered Chain Example**

```
import javax.jai.*;
import javax.jai.widget.*;
import java.awt.Frame;

public class AddExample extends Frame {

     // ScrollingImagePanel is a utility widget that
     // contains a Graphics2D (i.e., is an image sink).
     ScrollingImagePanel imagePanel1;

     // For simplicity, we just do all the work in the
     // class constructor.
     public AddExample(ParameterBlock param1,
                       ParameterBlock param2) {

          // Create a constant image
          RenderedOp im0 = JAI.create(“constant”, param1);

          // Create another constant image.
          RenderedOp im1 = JAI.create(“constant”, param2);
          // Add the two images together.
```

**Listing 3-1 Rendered Chain Example (Continued)**

```
            RenderedOp im2 = JAI.create("add", im0, im1);

            // Display the original in a scrolling window
            imagePanel1 = new ScrollingImagePanel(im2, 100, 100);

            // Add the display widget to our frame.
            add(imagePanel1);
        }
    }
```

The first three lines of the example code specify which classes to import. The classes prefixed with `javax.jai` are the Java Advanced Imaging classes. The `java.awt` prefix specifies the core Java API classes.

```
import javax.jai.*;
import javax.jai.widget.*;
import java.awt.Frame;
```

The next line declares the name of the program and that it runs in a `Frame`, a window with a title and border.

```
public class AddExample extends Frame {
```

The next line of code creates a `ScrollingImagePanel`, which is the ultimate destination of our image:

```
ScrollingImagePanel imagePanel1;
```

Next, a `ParameterBlock` for each source image is defined. The parameters specify the image height, width, origin, tile size, and so on.

```
public AddExample(ParameterBlock param1,
                  ParameterBlock param2) {
```

The next two lines define two operations that create the two "constant" images that will be added together to create the destination image (see Section 4.7, "Creating a Constant Image").

```
RenderedOp im0 = JAI.create("constant", param1);
RenderedOp im1 = JAI.create("constant", param2);
```

Next, our example adds the two images together (see Section 6.5.1, "Adding Two Source Images").

```
RenderedOp im2 = JAI.create("add", im0, im1);
```

Finally, we display the destination image in a scrolling window and add the display widget to our frame.

```
imagePanel1 = new ScrollingImagePanel(im2, 100, 100);
add(imagePanel1);
```

Once pixels start flowing, the graph will look like Figure 3-2. The display widget drives the process. We mention this because the source images are not loaded and no pixels are produced until the display widget actually requests them.

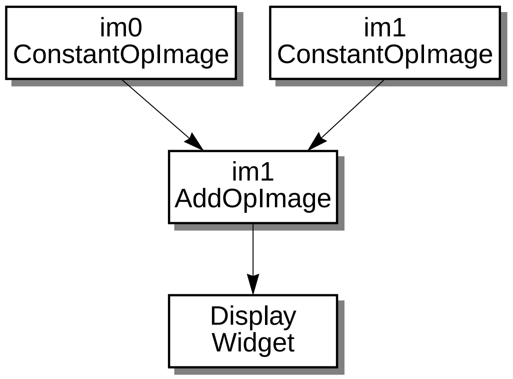


**Figure 3-2 Rendered Chain Example**

### 3.3.2 Renderable Graphs

A *renderable graph* is a graph that is not evaluated at the time it is specified. The evaluation is deferred until there is a specific request for a rendering. This is known as *deferred execution*; evaluation is deferred until there is a specific request for rendering.

In a renderable graph, if a source image should change before there is a request for rendering, the changes will be reflected in the output. This process can be thought of as a “pull” model, in which the requestor pulls the image through the chain, which is the opposite of the AWT imaging push model.

A renderable graph is made up of nodes implementing the `RenderableImage` interface, which are usually instances of the `RenderableOp` class. As the renderable graph is constructed, the sources of each node are specified to form the graph topology. The source of a renderable graph is a Renderable image object.

Let’s take a look at an example of a renderable graph in Listing 3-2. This example reads a TIFF file, inverts its pixel values, then adds a constant value to the pixels. Once again, this example is a code fragment rather than an entire class definition.

**Listing 3-2 Renderable Chain Example**

```
// Get rendered source object from a TIFF source.
// The ParameterBlock ‘pb0’ contains the name
// of the source (file, URL, etc.). The objects ‘hints0’,
// ‘hints1’, and ‘hints2’ contain rendering hints and are
// assumed to be created outside of this code fragment.
RenderedOp sourceImg =
          JAI.create(“TIFF”, pb0);

// Derive the RenderableImage from the source RenderedImage.
ParameterBlock pb = new ParameterBlock();
pb.addSource(sourceImg);
pb.add(null).add(null).add(null).add(null).add(null);

// Create the Renderable operation.
RenderableImage ren = JAI.createRenderable("renderable", pb);

// Set up the parameter block for the first op.
ParameterBlock pb1 = new ParameterBlock();
pb1.addSource(ren);

// Make first Op in Renderable chain an invert.
RenderableOp Op1 = JAI.createRenderable(“invert”, pb1);

// Set up the parameter block for the second Op.
// The constant to be added is “2”.
ParameterBlock pb2 = new ParameterBlock();
pb2.addSource(Op1);        // Op1 as the source
pb2.add(2.0f);             // 2.0f as the constant

// Make a second Op a constant add operation.
RenderableOp Op2 =
          JAI.createRenderable(“addconst”, pb2);

// Set up a rendering context.
AffineTransform screenResolution = ...;
RenderContext rc = new RenderContext(screenResolution);

// Get a rendering.
RenderedImage rndImg1 = Op2.createRendering(rc);

// Display the rendering onscreen using screenResolution.
imagePanel1 = new ScrollingImagePanel(rndImg1, 100, 100);
```

In this example, the image source is a TIFF image. A TIFF `RenderedOp` is created as a source for the subsequent operations:

```
RenderedOp sourceImg =
          JAI.create(“TIFF”, pb0);
```

The rendered source image is then converted to a renderable image:

```
ParameterBlock pb = new ParameterBlock();
pb.addSource(sourceImg);
pb.add(null).add(null).add(null).add(null).add(null);
RenderableImage ren = JAI.createRenderable("renderable", pb);
```

Next, a `ParameterBlock` is set up for the first operation. The parameter block contains sources for the operation and parameters or other objects that the operator may require.

```
ParameterBlock pb1 = new ParameterBlock();
pb1.addSource(sourceImage);
```

An “invert” `RenderableOp` is then created with the TIFF image as the source. The `invert` operation inverts the pixel values of the source image and creates a `RenderableImage` as the result of applying the operation to a tuple (source and parameters).

```
RenderableOp Op1 = JAI.createRenderable(“invert”, pb1);
```

The next part of the code example sets up a `ParameterBlock` for the next operation. The `ParameterBlock` defines the previous operation (Op1) as the source of the next operation and sets a constant with a value of 2.0, which will be used in the next “add constant” operation.

```
ParameterBlock pb2 = new ParameterBlock();
pb2.addSource(Op1);         // Op1 as the source
pb2.add(2.0f);              // 2.0f as the constant
```

The second operation (`Op2`) is an add constant (`addconst`), which adds the constant value (2.0) to the pixel values of a source image on a per-band basis. The `pb2` parameter is the `ParameterBlock` set up in the previous step.

```
RenderableOp Op2 =
          JAI.createRenderable(“addconst”, pb2);
```

After `Op2` is created, the renderable chain thus far is shown in Figure 3-3.

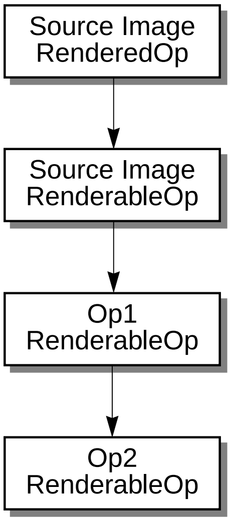


**Figure 3-3 Renderable Chain Example**

Next, a `RenderContext` is created using an `AffineTransform` that will produce a screen-size rendering.

```
AffineTransform screenResolution = ...;
RenderContext rc = new RenderContext(screenResolution);
```

This rendering is created by calling the `RenderableImage.createRendering` method on `Op2`. The `createRendering` method does not actually compute any pixels, bit it does instantiate a `RenderedOp` chain that will produce a rendering at the appropriate pixel dimensions.

```
RenderedImage rndImg1 = Op2.createRendering(rc);
```

The Renderable graph can be thought of as a *template* that, when rendered, causes the instantiation of a parallel Rendered graph to accomplish the actual processing. Now let's take a look at what happens back up the rendering chain in our example:

- When the `Op2.createRendering` method is called, it recursively calls the `Op1.createRendering` method with the `RenderContext rc` as the argument.
- The `Op1` operation then calls the `sourceImg.getImage` method, again with `rc` as the argument. `sourceImg` creates a new `RenderedImage` to hold its source pixels at the required resolution and inserts it into the chain. It then returns a handle to this object to `Op1`.

- `Op1` then uses the `OperationRegistry` to find a `ContextualRenderedImageFactory` (CRIF) that can perform the "invert" operation. The resulting `RenderedOp` object returned by the CRIF is inserted into the chain with the handle returned by `sourceImg` as its source.
- The handle to the "invert" `RenderedImage` is returned to `Op2`, which repeats the process, creating an "addconst" `RenderedOp`, inserting it into the chain and returning a handle to `rndImg1`.
- Finally, `rndImg1` is used in the call to the `ScrollingImagePanel` to display the result on the screen.

After the creation of the `ScrollingImagePanel`, the Renderable and Rendered chains look like Figure 3-4.

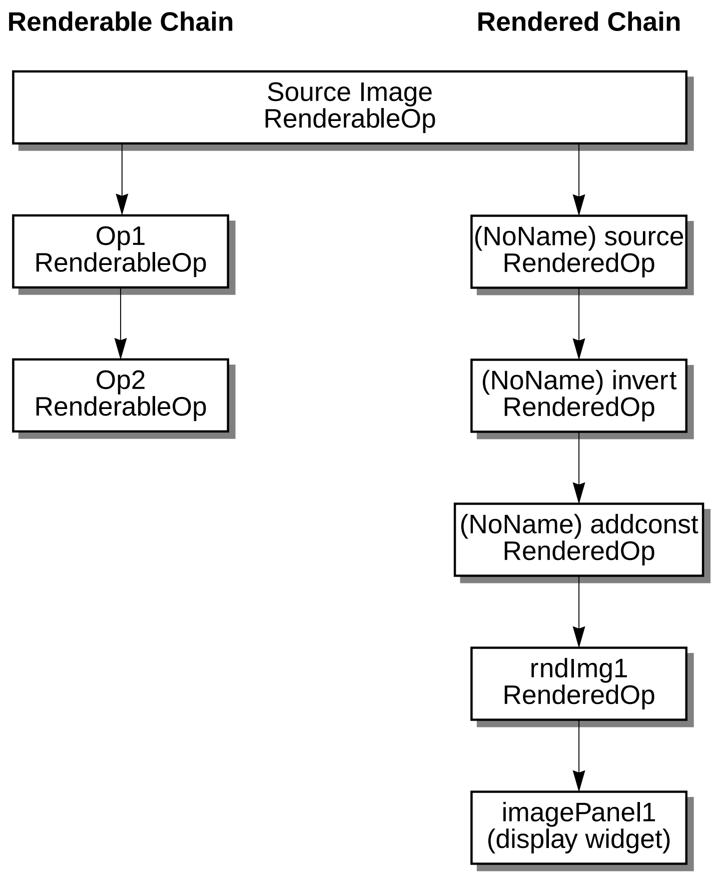

Renderable Chain
Rendered Chain
Source Image RenderableOp
Op1 RenderableOp
Op2 RenderableOp
(NoName) source RenderedOp
(NoName) invert RenderedOp
(NoName) addconst RenderedOp
rndImg1 RenderedOp
imagePanel1 (display widget)

**Figure 3-4 Renderable and Rendered Graphs after the getImage Call**

At this point in the chain, no pixels have been processed and no `OpImages`, which actually calculate the results, have been created. Only when the `ScrollingImagePanel` needs to put pixels on the screen are the `OpImages` created and pixels pulled through the Rendered chain, as done in the final line of code.

```
imagePanel1 = new ScrollingImagePanel(rndImg1, 100, 100);
```

### 3.3.3 Reusing Graphs

Many times, it is more desirable to make changes to an existing graph and reuse it than to create another nearly identical graph. Both Rendered and Renderable graphs are editable, with certain limitations.

#### 3.3.3.1 Editing Rendered Graphs

Initially, a node in a Rendered graph is mutable; it may be assigned new sources, which are considered to be evaluated as soon as they are assigned, and its parameter values may be altered. However, once rendering takes place at a node, it becomes frozen and its sources and parameters cannot be changed.

A chain of Rendered nodes may be cloned without freezing any of its nodes by means of the `RenderedOp.createInstance` method. Using the `createInstance` method, a Rendered graph may be configured and reused at will, as well as serialized and transmitted over a network.

The `RenderedOp` class provides several methods for reconfiguring a Rendered node. The `setParameter` methods can be used to set the node's parameters to a `byte`, `char`, `short`, `int`, `long`, `float`, `double`, or an `Object`. The `setOperationName` method can be used to change the operation name. The `setParameterBlock` method can be used to change the nodes's `ParameterBlock`.

#### 3.3.3.2 Editing Renderable Graphs

Since Renderable graphs are not evaluated until there is a specific request for a rendering, the nodes may be edited at any time. The main concern with editing Renderable graphs is the introduction of cycles, which must be avoided.

The `RenderableOp` class provides several methods for reconfiguring a Renderable node. The `setParameter` methods can be used to set the node's parameters to a `byte`, `char`, `short`, `int`, `long`, `float`, `double`, or an `Object`. The `setParameterBlock` method can be used to change the nodes's `ParameterBlock`. The `setProperty` method can be used to change a node's local property. The `setSource` method can be used to set one of the node's sources to an `Object`.

## 3.4 Remote Execution

Up to this point, we have been talking about standalone image processing. JAI also provides for client-server image processing through what is called the *Remote Execution* model.

Remote execution is based on Java RMI (remote method invocation). Java RMI allows Java code on a client to invoke method calls on objects that reside on another computer without having to move those objects to the client. The advantages of remote execution become obvious if you think of several clients wanting to access the same objects on a server. To learn more about remote method invocation, refer to one of the books on Java described in "Related Documentation" on page xv.

To do remote method invocation in JAI, a `RemoteImage` is set up on the server and a `RenderedImage` chain is set up on the client. For more information, see Chapter 12, "Client-Server Imaging."

## 3.5 Basic JAI API Classes

JAI consists of several classes grouped into five packages:

- `javax.media.jai` – contains the "core" JAI interfaces and classes
- `javax.media.jai.iterator` – contains special iterator interfaces and classes, which are useful for writing extension operations
- `javax.media.jai.operator` – contains classes that describe all of the image operators
- `javax.media.jai.widget` – contains interfaces and classes for creating simple image canvases and scrolling windows for image display

Now, let's take a look at the most common classes in the JAI class hierarchy.

### 3.5.1 The JAI Class

The `JAI` class cannot be instantiated; it is simply a placeholder for static methods that provide a simple syntax for creating Renderable and Rendered graphs. The majority of the methods in this class are used to create a `RenderedImage`, taking an operation name, a `ParameterBlock`, and `RenderingHints` as arguments.

There is one method to create a RenderableImage, taking an operation name, a ParameterBlock, and RenderingHints as arguments.

There are several variations of the create method, all of which take sources and parameters directly and construct a ParameterBlock automatically.

### 3.5.2 The PlanarImage Class

The PlanarImage class is the main class for describing two-dimensional images in JAI. PlanarImage implements the RenderedImage interface from the Java 2D API. TiledImage and OpImage, described later, are subclasses of PlanarImage.

The RenderedImage interface describes a tiled, read-only image with a pixel layout described by a SampleModel and a DataBuffer. Each tile is a rectangle of identical dimensions, laid out on a regular grid pattern. All tiles share a common SampleModel.

In addition to the capabilities offered by RenderedImage, PlanarImage maintains source and sink connections between the nodes of rendered graphs. Since graph nodes are connected bidirectionally, the garbage collector requires assistance to detect when a portion of a graph is no longer referenced from user code and may be discarded. PlanarImage takes care of this by using the *Weak References API* of Java 2.

Any RenderedImages from outside the API are “wrapped” to produce an instance of PlanarImage. This allows the API to make use of the extra functionality of PlanarImage for all images.

### 3.5.3 The CollectionImage Class

CollectionImage is the abstract superclass for four classes representing collections of PlanarImages:

- ImageStack – represents a set of two-dimensional images lying in a common three-dimensional space, such as CT scans or seismic volumes. The images need not lie parallel to one another.
- ImageSequence – represents a sequence of images with associated time stamps and camera positions. This class can be used to represent video or time-lapse photography.
- ImagePyramid – represents a series of images of progressively lesser resolution, each derived from the last by means of an imaging operator.

- `ImageMIPMap` – represents a stack of images with a fixed operational relationship between adjacent slices.

### 3.5.4 The TiledImage Class

The `TiledImage` class represents images containing multiple tiles arranged into a grid. The tiles form a regular grid, which may occupy any rectangular region of the plane.

`TiledImage` implements the `WritableRenderedImage` interface from the Java 2D API, as well as extending `PlanarImage`. A `TiledImage` allows its tiles to be checked out for writing, after which their pixel data may be accessed directly. `TiledImage` also has a `createGraphics` method that allows its contents to be altered using Java 2D API drawing calls.

A `TiledImage` contains a tile grid that is initially empty. As each tile is requested, it is initialized with data from a `PlanarImage` source. Once a tile has been initialized, its contents can be altered. The source image may also be changed for all or part of the `TiledImage` using its `set` methods. In particular, an arbitrary region of interest (ROI) may be filled with data copied from a `PlanarImage` source.

The `TiledImage` class includes a method that allows you to paint a `Graphics2D` onto the `TiledImage`. This is useful for adding text, lines, and other simple graphics objects to an image for annotating the image. For more on the TiledImage class, see Section 4.2.2, “Tiled Image.”

### 3.5.5 The OpImage Class

The OpImage class is the parent class for all imaging operations, such as:

- `AreaOpImage` – for image operators that require only a fixed rectangular source region around a source pixel to compute each destination pixel
- `PointOpImage` – for image operators that require only a single source pixel to compute each destination pixel
- `SourcelessOpImage` – for image operators that have no image sources
- `StatisticsOpImage` – for image operators that compute statistics on a given region of an image, and with a given sampling rate
- `UntiledOpimage` – for single-source operations in which the values of all pixels in the source image contribute to the value of each pixel in the destination image

- `WarpOpImage` – for image operators that perform an image warp
- `ScaleOpImage` – for extension operators that perform image scaling requiring rectilinear backwards mapping and padding by the resampling filter dimensions

The `OpImage` is able to determine what source areas are sufficient for the computation of a given area of the destination by means of a user-supplied `mapDestRect` method. For most operations, this method as well as a suitable implementation of `getTile` is supplied by a standard subclass of `OpImage`, such as `PointOpImage` or `AreaOpImage`.

An `OpImage` is effectively a `PlanarImage` that is defined computationally. In `PlanarImage`, the `getTile` method of `RenderedImage` is left abstract, and `OpImage` subclasses override it to perform their operation. Since it may be awkward to produce a tile of output at a time, due to the fact that source tile boundaries may need to be crossed, the `OpImage` class defines a `getTile` method to cobble (copy) source data as needed and to call a user-supplied `computeRect` method. This method then receives contiguous source `Rasters` that are guaranteed to contain sufficient data to produce the desired results. By calling `computeRect` on subareas of the desired tile, `OpImage` is able to minimize the amount of data that must be cobbled.

A second version of the `computeRect` method that is called with uncobbled sources is available to extenders. This interface is useful for operations that are implemented using *iterators* (see Section 14.4, "Iterators"), which abstract away the notion of tile boundaries.

## 3.5.6 The RenderableOp Class

The `RenderableOp` class provides a lightweight representation of an operation in the Renderable space (see Section 3.3.2, "Renderable Graphs"). `RenderableOps` are typically created using the `createRenderable` method of the `JAI` class, and may be edited at will. `RenderableOp` implements the `RenderableImage` interface, and so may be queried for its rendering-independent dimensions.

When a `RenderableOp` is to be rendered, it makes use of the `OperationRegistry` (described in Chapter 14) to locate an appropriate `ContextualRenderedImageFactory` object to perform the conversion from the Renderable space into a `RenderedImage`.

### 3.5.7 The RenderedOp Class

The `RenderedOp` is a lightweight object similar to `RenderableOp` that stores an operation name, `ParameterBlock`, and `RenderingHints`, and can be joined into a Rendered graph (see Section 3.3.1, “Rendered Graphs”). There are two ways of producing a rendering of a `RenderedOp`:

- Implicit – Any call to a `RenderedImage` method on a `RenderedOp` causes a rendering to be created. This rendering will usually consist of a chain of `OpImages` with a similar geometry to the `RenderedOp` chain. It may have more or fewer nodes, however, since the rendering process may both collapse nodes together by recognizing patterns, and expand nodes by the use of the `RenderedImageFactory` interface. The `OperationRegistry` (described in Chapter 14) is used to guide the `RenderedImageFactory` selection process.
- Explicit – A call to `createInstance` effectively clones the `RenderedOp` and its source `RenderedOps`, resulting in an entirely new Rendered chain with the same non-`RenderedOp` sources (such as `TiledImages`) as the original chain. The bottom node of the cloned chain is then returned to the caller. This node will then usually be implicitly rendered by calling `RenderedImage` methods on it.

`RenderedOps` that have not been rendered may have their sources and parameters altered. Sources are considered evaluated as soon as they are connected to a `RenderedOp`.

## 3.6 JAI API Operators

The JAI API specifies a core set of image processing operators. These operators provide a common ground for applications programmers, since they can then make assumptions about what operators are guaranteed to be present on all platforms.

The general categories of image processing operators supported include:

- Point Operators
- Area Operators
- Geometric Operators
- Color Quantization Operators
- File Operators
- Frequency Operators

- Statistical Operators
- Edge Extraction Operators
- Miscellaneous Operators

The JAI API also supports abstractions for many common types of image collections, such as time-sequential data and image pyramids. These are intended to simplify operations on image collections and allow the development of operators that work directly on these abstractions.

### 3.6.1 Point Operators

Point operators allow you to modify the way in which the image data fills the available range of gray levels. This affects the image's appearance when displayed. Point operations transform an input image into an output image in such a way that each output pixel depends only on the corresponding input pixel. Point operations do not modify the spatial relationships within an image.

Table 3-1 lists the JAI point operators.

**Table 3-1 Point Operators**

| Operator | Description | Reference |
|---|---|---|
| `Absolute` | Takes one rendered or renderable source image, and computes the mathematical absolute value of each pixel. | page 177 |
| `Add` | Takes two rendered or renderable source images, and adds every pair of pixels, one from each source image of the corresponding position and band. | page 166 |
| `AddCollection` | Takes a collection of rendered source images, and adds every pair of pixels, one from each source image of the corresponding position and band. | page 168 |
| `AddConst` | Takes a collection of rendered images and an array of double constants, and for each rendered image in the collection adds a constant to every pixel of its corresponding band. | page 167 |
| `AddConstToCollection` | Takes a collection of rendered images and an array of double constants, and for each rendered image in the collection adds a constant to every pixel of its corresponding band. | page 169 |
| `And` | Takes two rendered or renderable source images and performs a bit-wise logical AND on every pair of pixels, one from each source image, of the corresponding position and band. | page 158 |

**Table 3-1 Point Operators (Continued)**

| Operator | Description | Reference |
|---|---|---|
| `AndConst` | Takes one rendered or renderable source image and an array of integer constants, and performs a bit-wise logical AND between every pixel in the same band of the source and the constant from the corresponding array entry. | page 159 |
| `BandCombine` | Takes one rendered or renderable source image and computes a set of arbitrary linear combinations of the bands using a specified matrix. | page 141 |
| `BandSelect` | Takes one rendered or renderable source image, chooses N bands from the image, and copies the pixel data of these bands to the destination image in the order specified. | page 185 |
| `Clamp` | Takes one rendered or renderable source image and sets all the pixels whose value is below a low value to that low value and all the pixels whose value is above a high value to that high value. The pixels whose value is between the low value and the high value are left unchanged. | page 184 |
| `ColorConvert` | Takes one rendered or renderable source image and performs a pixel-by-pixel color conversion of the data. | page 140 |
| `Composite` | Takes two rendered or renderable source images and combines the two images based on their alpha values at each pixel. | page 243 |
| `Constant` | Takes one rendered or renderable source image and creates a multi-banded, tiled rendered image, where all the pixels from the same band have a constant value. | page 123 |
| `Divide` | Takes two rendered or renderable source images, and for every pair of pixels, one from each source image of the corresponding position and band, divides the pixel from the first source by the pixel from the second source. | page 171 |
| `DivideByConst` | Takes one rendered source image and divides the pixel values of the image by a constant. | page 172 |
| `DivideComplex` | Takes two rendered or renderable source images representing complex data and divides them. | page 174 |
| `DivideIntoConst` | Takes one rendered or renderable source image and an array of double constants, and divides every pixel of the same band of the source into the constant from the corresponding array entry. | page 173 |
| `Exp` | Takes one rendered or renderable source image and computes the exponential of the pixel values. | page 177 |
| `Invert` | Takes one rendered or renderable source image and inverts the pixel values. | page 241 |

**Table 3-1 Point Operators (Continued)**

| Operator | Description | Reference |
|---|---|---|
| `Log` | Takes one rendered or renderable source image and computes the natural logarithm of the pixel values. The operation is done on a per-pixel, per-band basis. For integral data types, the result will be rounded and clamped as needed. | page 241 |
| `Lookup` | Takes one rendered or renderable source image and a lookup table, and performs general table lookup by passing the source image through the table. | page 205 |
| `MatchCDF` | Takes one rendered or renderable source image and performs a piecewise linear mapping of the pixel values such that the Cumulative Distribution Function (CDF) of the destination image matches as closely as possible a specified Cumulative Distribution Function. | page 203 |
| `Max` | Takes two rendered or renderable source images, and for every pair of pixels, one from each source image of the corresponding position and band, finds the maximum pixel value. | page 156 |
| `Min` | Takes two rendered or renderable source images and for every pair of pixels, one from each source image of the corresponding position and band, finds the minimum pixel value. | page 157 |
| `Multiply` | Takes two rendered or renderable source images, and multiplies every pair of pixels, one from each source image of the corresponding position and band. | page 174 |
| `MultiplyComplex` | Takes two rendered source images representing complex data and multiplies the two images. | page 176 |
| `MultiplyConst` | Takes one rendered or renderable source image and an array of double constants, and multiplies every pixel of the same band of the source by the constant from the corresponding array entry. | page 175 |
| `Not` | Takes one rendered or renderable source image and performs a bit-wise logical NOT on every pixel from every band of the source image. | page 164 |
| `Or` | Takes two rendered or renderable source images and performs bit-wise logical OR on every pair of pixels, one from each source image of the corresponding position and band. | page 160 |
| `OrConst` | Takes one rendered or renderable source image and an array of integer constants, and performs a bit-wise logical OR between every pixel in the same band of the source and the constant from the corresponding array entry. | page 161 |
| `Overlay` | Takes two rendered or renderable source images and overlays the second source image on top of the first source image. | page 242 |

**Table 3-1 Point Operators (Continued)**

| Operator | Description | Reference |
|---|---|---|
| `Pattern` | Takes a rendered source image and defines a tiled image consisting of a repeated pattern. | page 80 |
| `Piecewise` | Takes one rendered or renderable source image and performs a piecewise linear mapping of the pixel values. | page 202 |
| `Rescale` | Takes one rendered or renderable source image and maps the pixel values of an image from one range to another range by multiplying each pixel value by one of a set of constants and then adding another constant to the result of the multiplication. | page 200 |
| `Subtract` | Takes two rendered or renderable source images, and for every pair of pixels, one from each source image of the corresponding position and band, subtracts the pixel from the second source from the pixel from the first source. | page 169 |
| `SubtractConst` | Takes one rendered or renderable source image and an array of double constants, and subtracts a constant from every pixel of its corresponding band of the source. | page 170 |
| `SubtractFromConst` | Takes one rendered or renderable source image and an array of double constants, and subtracts every pixel of the same band of the source from the constant from the corresponding array entry. | page 171 |
| `Threshold` | Takes one rendered or renderable source image, and maps all the pixels of this image whose value falls within a specified range to a specified constant. | page 245 |
| `Xor` | Takes two rendered or renderable source images, and performs a bit-wise logical XOR on every pair of pixels, one from each source image of the corresponding position and band. | page 162 |
| `XorConst` | Takes one rendered or renderable source image and an array of integer constants, and performs a bit-wise logical XOR between every pixel in the same band of the source and the constant from the corresponding array entry. | page 163 |

## 3.6.2 Area Operators

The area operators perform geometric transformations, which result in the repositioning of pixels within an image. Using a mathematical transformation, pixels are located from their *x* and *y* spatial coordinates in the input image to new coordinates in the output image.

There are two basic types of area operations: linear and nonlinear. Linear operations include translation, rotation, and scaling. Non-linear operations, also known as *warping transformations*, introduce curvatures and bends to the processed image.

Table 3-2 lists the JAI area operators.

**Table 3-2 Area Operators**

| Operator | Description | Reference |
|---|---|---|
| `Border` | Takes one rendered source image and adds a border around it. | page 191 |
| `BoxFilter` | Takes one rendered source image and determines the intensity of a pixel in the image by averaging the source pixels within a rectangular area around the pixel. | page 224 |
| `Convolve` | Takes one rendered source image and performs a spatial operation that computes each output sample by multiplying elements of a kernel with the samples surrounding a particular source sample. | page 221 |
| `Crop` | Takes one rendered or renderable source image and crops the image to a specified rectangular area. | page 199 |
| `MedianFilter` | Takes a rendered source image and passes it through a non-linear filter that is useful for removing isolated lines or pixels while preserving the overall appearance of the image. | page 226 |

## 3.6.3 Geometric Operators

Geometric operators allow you to modify the orientation, size, and shape of an image. Table 3-3 lists the JAI geometric operators.

**Table 3-3 Geometric Operators**

| Operator | Description | Reference |
|---|---|---|
| `Affine` | Takes one rendered or renderable source image and performs (possibly filtered) affine mapping on it. | page 272 |
| `Rotate` | Takes one rendered or renderable source image and rotates the image about a given point by a given angle, specified in radians. | page 270 |
| `Scale` | Takes one rendered or renderable source image and translates and resizes the image. | page 268 |
| `Shear` | Takes one rendered source image and shears the image either horizontally or vertically. | page 283 |
| `Translate` | Takes one rendered or renderable source image and copies the image to a new location in the plane. | page 266 |
| `Transpose` | Takes one rendered or renderable source image and flips or rotates the image as specified. | page 281 |
| `Warp` | Takes one rendered source image and performs (possibly filtered) general warping on the image. | page 285 |

### 3.6.4 Color Quantization Operators

Color quantization, also known as *dithering*, is often used to reduce the appearance of amplitude contouring on monochrome frame buffers with fewer than eight bits of depth or color frame buffers with fewer than 24 bits of depth. Table 3-4 lists the JAI color quantization operators.

**Table 3-4 Color Quantization Operators**

| Operator | Description | Reference |
|---|---|---|
| `ErrorDiffusion` | Takes one rendered source image and performs color quantization by finding the nearest color to each pixel in a supplied color map and "diffusing" the color quantization error below and to the right of the pixel. | page 181 |
| `OrderedDither` | Takes one rendered source image and performs color quantization by finding the nearest color to each pixel in a supplied color cube and "shifting" the resulting index value by a pseudo-random amount determined by the values of a supplied dither mask. | page 178 |

### 3.6.5 File Operators

The file operators are used to read or write image files. Table 3-5 lists the JAI file operators.

**Table 3-5 File Operators**

| Operator | Description | Reference |
|---|---|---|
| `AWTImage` | Converts a standard java.awt.Image into a rendered image. | page 118 |
| `BMP` | Reads a standard BMP input stream. | page 111 |
| `Encode` | Takes one rendered source image and writes the image to a given OutputStream in a specified format using the supplied encoding parameters. | page 362 |
| `FileLoad` | Reads an image from a file. | page 104 |
| `FileStore` | Takes one rendered source image and writes the image to a given file in a specified format using the supplied encoding parameters. | page 361 |
| `Format` | Takes one rendered or renderable source image and reformats it. This operation is capable of casting the pixel values of an image to a given data type, replacing the SampleModel and ColorModel of an image, and restructuring the image's tile grid layout. | page 119 |
| `FPX` | Reads an image from a FlashPix stream. | page 109 |
| `GIF` | Reads an image from a GIF stream. | page 110 |

**Table 3-5 File Operators (Continued)**

| Operator | Description | Reference |
|---|---|---|
| `IIP` | Provides client-side support of the Internet Imaging Protocol (IIP) in both the rendered and renderable modes. It creates a RenderedImage or a RenderableImage based on the data received from the IIP server, and optionally applies a sequence of operations to the created image. | page 352 |
| `IIPResolution` | Provides client-side support of the Internet Imaging Protocol (IIP) in the rendered mode. It is resolution-specific. It requests from the IIP server an image at a particular resolution level, and creates a RenderedImage based on the data received from the server. | page 357 |
| `JPEG` | Reads an image from a JPEG (JFIF) stream. | page 110 |
| `PNG` | Reads a standard PNG version 1.1 input stream. | page 112 |
| `PNM` | Reads a standard PNM file, including PBM, PGM, and PPM images of both ASCII and raw formats. It stores the image data into an appropriate SampleModel. | page 117 |
| `Stream` | Produces an image by decoding data from a SeekableStream. The allowable formats are those registered with the `com.sun.media.jai.codec.ImageCodec` class. | page 103 |
| `TIFF` | Reads TIFF 6.0 data from a SeekableStream. | page 104 |
| `URL` | Creates an output image whose source is specified by a Uniform Resource Locator (URL). | page 119 |

### 3.6.6 Frequency Operators

Frequency operators are used to decompose an image from its spatial-domain form into a frequency-domain form of fundamental frequency components. Operators also are available to perform an inverse frequency transform, in which the image is converted from the frequency form back into the spatial form.

JAI supports several frequency transform types. The most common frequency transform type is the *Fourier transform*. JAI uses the discrete form known as the *discrete Fourier transform*. The *inverse discrete Fourier transform* can be used to convert the image back to a spatial image. JAI also supports the *discrete cosine transform* and its opposite, the *inverse discrete cosine transform*.

Table 3-6 lists the JAI frequency operators.

**Table 3-6 Frequency Operators**

| Operator | Description | Reference |
|---|---|---|
| `Conjugate` | Takes one rendered or renderable source image containing complex data and negates the imaginary components of the pixel values. | page 236 |
| `DCT` | Takes one rendered or renderable source image and computes the even discrete cosine transform (DCT) of the image. Each band of the destination image is derived by performing a two-dimensional DCT on the corresponding band of the source image. | page 232 |
| `DFT` | Takes one rendered or renderable source image and computes the discrete Fourier transform of the image. | page 228 |
| `IDCT` | Takes one rendered or renderable source image and computes the inverse even discrete cosine transform (DCT) of the image. Each band of the destination image is derived by performing a two-dimensional inverse DCT on the corresponding band of the source image. | page 233 |
| `IDFT` | Takes one rendered or renderable source image and computes the inverse discrete Fourier transform of the image. A positive exponential is used as the basis function for the transform. | page 231 |
| `ImageFunction` | Generates an image on the basis of a functional description provided by an object that is an instance of a class that implements the ImageFunction interface. | page 237 |
| `Magnitude` | Takes one rendered or renderable source image containing complex data and computes the magnitude of each pixel. | page 234 |
| `MagnitudeSquared` | Takes one rendered or renderable source image containing complex data and computes the squared magnitude of each pixel. | page 235 |
| `PeriodicShift` | Takes a rendered or renderable source image and generates a destination image that is the infinite periodic extension of the source image, with horizontal and vertical periods equal to the image width and height, respectively, shifted by a specified amount along each axis and clipped to the bounds of the source image. | page 236 |
| `Phase` | Takes one rendered or renderable source image containing complex data and computes the phase angle of each pixel. | page 235 |
| `PolarToComplex` | Takes two rendered or renderable source images and creates an image with complex-valued pixels from the two images the respective pixel values of which represent the magnitude (modulus) and phase of the corresponding complex pixel in the destination image. | page 237 |

### 3.6.7 Statistical Operators

Statistical operators provide the means to analyze the content of an image. Table 3-7 lists the JAI statistical operators.

**Table 3-7 Statistical Operators**

| Operator | Description | Reference |
|---|---|---|
| `Extrema` | Takes one rendered source image, scans a specific region of the image, and finds the maximum and minimum pixel values for each band within that region of the image. The image data pass through this operation unchanged. | page 308 |
| `Histogram` | Takes one rendered source image, scans a specific region of the image, and generates a histogram based on the pixel values within that region of the image. The histogram data is stored in the user supplied javax.media.jai.Histogram object, and may be retrieved by calling the getProperty method on this operation with `"histogram"` as the property name. The return value will be of type javax.media.jai.Histogram. The image data pass through this operation unchanged. | page 310 |
| `Mean` | Takes a rendered source image, scans a specific region, and computes the mean pixel value for each band within that region of the image. The image data pass through this operation unchanged. | page 307 |

### 3.6.8 Edge Extraction Operators

The edge extraction operators allow image edge enhancement. Edge enhancement reduces an image to show only its edge details. Edge enhancement is implemented through spatial filters that detect a specific *pixel brightness slope* within a group of pixels in an image. A steep brightness slope indicates the presence of an edge.

Table 3-8 lists the JAI edge extraction operators.

**Table 3-8 Edge Extraction Operators**

| Operator | Description | Reference |
|---|---|---|
| `GradientMagnitude` | Takes one rendered source image and computes the magnitude of the image gradient vector in two orthogonal directions. | page 315 |

### 3.6.9 Miscellaneous Operators

The miscellaneous operators do not fall conveniently into any of the previous categories. Table 3-9 lists the JAI miscellaneous operators.

**Table 3-9 Miscellaneous Operators**

| Operator | Description | Reference |
|---|---|---|
| `Renderable` | Takes one rendered source image and produces a RenderableImage consisting of a "pyramid" of RenderedImages at progressively lower resolutions. | page 122 |

## 3.7 Creating Operations

Most image operation objects are created with some variation on the following methods:

#### *For a renderable graph:*

There are four variations on methods for creating operations in the Renderable mode, as listed in Table 3-10.

**Table 3-10 JAI Class Renderable Mode Methods**

| Method | Parameters | Description |
|---|---|---|
| `createRenderable` | `opName` `parameterBlock` | Creates a `RenderableOp` that represents the named operation, using the sources and parameters specified in the `ParameterBlock`. |
| `createRenderableNS` | `opName` `parameterBlock` | The same as the previous method, only this version is non-static. |
| `createRenderable-Collection` | `opName` `parameterBlock` | Creates a `Collection` that represents the named operation, using the sources and parameters specified in the `ParameterBlock`. |
| `createRenderable-CollectionNS` | `opName` `parameterBlock` | The same as the previous method, only this version is non-static. |

For example:

```
   RenderableOp im = JAI.createRenderable(“operationName”,
                                           paramBlock);
```

The `JAI.createRenderable` method creates a renderable node operation that takes two parameters:

- An operation name (see Section 3.7.1, "Operation Name")
- A source and a set of parameters for the operation contained in a parameter block (see Section 3.7.2, "Parameter Blocks")

***For a rendered graph:***

There are a great many more variations on methods for creating operations in the Rendered mode, as listed in Table 3-11. The first five methods in the table take sources and parameters specified in a `ParameterBlock`. The remaining methods are convenience methods that take various numbers of sources and parameters directly.

**Table 3-11 JAI Class Rendered Mode Methods**

| Method | Parameters | Description |
|---|---|---|
| `create` | `opName`<br>`parameterBlock`<br>`hints` | Creates a `RenderedOp` that represents the named operation, using the sources and parameters specified in the `ParameterBlock`, and applying the specified hints to the destination. This method is appropriate only when the final results return a single `RenderedImage`. |
| `createNS` | `opName`<br>`parameterBlock`<br>`hints` | The same as the previous method, only this version is non-static. |
| `createCollection` | `opName`<br>`parameterBlock`<br>`hints` | Creates a `Collection` that represents the named operation, using the sources and parameters specified in the `ParameterBlock`, and applying the specified hints to the destination. This method is appropriate only when the final results return a `Collection`. |
| `createCollectionNS` | `opName`<br>`parameterBlock`<br>`hints` | The same as the previous method, only this version is non-static. |
| `create` | `opName`<br>`parameterBlock` | Creates a `RenderedOp` with null rendering hints. |
| `create` | `opName`<br>`param` | Creates a `RenderedOp` that takes one parameter. |
| `create` | `opName`<br>`param1`<br>`param2` | Creates a `RenderedOp` that takes two parameters. There are two variations on this method, depending on the parameter data type (Object or int). |
| `create` | `opName`<br>`param1`<br>`param2`<br>`param3` | Creates a `RenderedOp` that takes three parameters. There are two variations on this method, depending on the parameter data type (Object or int). |

**Table 3-11 JAI Class Rendered Mode Methods (Continued)**

| Method | Parameters | Description |
|---|---|---|
| `create` | `opName`<br>`param1`<br>`param2`<br>`param3`<br>`param4` | Creates a `RenderedOp` that takes four parameters. There are two variations on this method, depending on the parameter data type (Object or int). |
| `create` | `opName`<br>`renderedImage` | Creates a `RenderedOp` that takes one source image. |
| `create` | `opName`<br>`Collection` | Creates a `RenderedOp` that takes one source collection. |
| `create` | `opName`<br>`renderedImage`<br>`param` | Creates a `RenderedOp` that takes one source and one parameter. There are two variations on this method, depending on the parameter data type (Object or int). |
| `create` | `opName`<br>`renderedImage`<br>`param1`<br>`param2` | Creates a `RenderedOp` that takes one source and two parameters. There are two variations on this method, depending on the parameter data type (Object or float). |
| `create` | `opName`<br>`renderedImage`<br>`param1`<br>`param2`<br>`param3` | Creates a `RenderedOp` that takes one source and three parameters. There are three variations on this method, depending on the parameter data type (Object, int, or float). |
| `create` | `opName`<br>`renderedImage`<br>`param1`<br>`param2`<br>`param3`<br>`param4` | Creates a `RenderedOp` that takes one source and four parameters. There are four variations on this method, depending on the parameter data type (Object, int, or float). |
| `create` | `opName`<br>`renderedImage`<br>`param1`<br>`param2`<br>`param3`<br>`param4`<br>`param5` | Creates a `RenderedOp` that takes one source and five parameters. There are three variations on this method, depending on the parameter data type (Object, int, or float). |
| `create` | `opName`<br>`renderedImage`<br>`param1`<br>`param2`<br>`param3`<br>`param4`<br>`param5`<br>`param6` | Creates a `RenderedOp` that takes one source and six parameters. There are two variations on this method, depending on the parameter data type (Object or int). |
| `create` | `opName`<br>`renderedImage1`<br>`renderedImage2` | Creates a `RenderedOp` that takes two sources. |

**Table 3-11 JAI Class Rendered Mode Methods (Continued)**

| Method | Parameters | Description |
|---|---|---|
| `create` | `opName`<br>`renderedImage1`<br>`renderedImage2`<br>`param1`<br>`param2` | Creates a `RenderedOp` that takes two sources and four parameters. |
| `createCollection` | `opName`<br>`parameterBlock` | Creates a `Collection` with `null` rendering hints. |

Two versions of the `create` method are non-static and are identified as `createNS`. These methods may be used with a specific instance of the `JAI` class and should only be used when the final result returned is a single `RenderedImage`. However, the source (or sources) supplied may be a collection of images or a collection of collections. The following is an example of one of these methods:

```
RenderedOp im = JAI.createNS("operationName", source, param1,
                             param2)
```

The rendering hints associated with this instance of `JAI` are overlaid with the hints passed to this method. That is, the set of keys will be the union of the keys from the instance's hints and the hints parameter. If the same key exists in both places, the value from the hints parameter will be used.

Many of the JAI operations have default values for some of the parameters. If you wish to use any of the default values in an operation, you do not have to specify that particular parameter in the `ParameterBlock`. The default value is automatically used in the operation. Parameters that do not have default values are required; failure to supply a required parameter results in a `NullPointerException`.

## 3.7.1 Operation Name

The operation name describes the operator to be created. The operation name is a string, such as "`add`" for the operation to add two images. See Section 3.6, "JAI API Operators," for a list of the operator names.

The operation name is always enclosed in quotation marks. For example:

```
"Mean"
"BoxFilter"
"UnsharpMask"
```

The operation name parsing is case-insensitive. All of the following variations are legal:

```
“OrConst”
“orConst”
“ORconst”
“ORCONST”
“orconst”
```

### 3.7.2 Parameter Blocks

The parameter block contains the source of the operation and a set parameters used by the operation. The contents of the parameter block depend on the operation being created and may be as simple as the name of the source image or may contain all of the operator parameters (such as the *x* and *y* displacement and interpolation type for the `translate` operation).

Parameter blocks encapsulate all the information about sources and parameters (Objects) required by the operation. The parameters specified by a parameter block are objects.

These controlling parameters and sources can be edited through the `setParameterBlock` method to affect specific operations or even the structure of the rendering chain itself. The modifications affect future `RenderedImages` derived from points in the chain below where the change took place.

There are two separate classes for specifying parameter blocks:

- `java.awt.image.renderable.ParameterBlock` – the main class for specifying and changing parameter blocks.
- `javax.media.jai.ParameterBlockJAI` – extends `ParameterBlock` by allowing the use of default parameter values and the use of parameter names.

The parameter block must contain the same number of sources and parameters as required by the operation (unless ParameterBlockJAI is used and the operation supplies default values). Note that, if the operation calls for one or more source images, they must be specified in the parameter block. For example, the `Add` operation requires two source images and no parameters. The `addConst` operator requires one source and a parameter specifying the constant value.

If the sources and parameters do not match the operation requirements, an exception is thrown. However, when the `ParameterBlockJAI` class is used, if the required parameter values are not specified, default parameter values are automatically inserted when available. For some operations, default parameter values are not available and must be supplied.

#### 3.7.2.1 Adding Sources to a Parameter Block

Sources are added to a parameter block with the `addSource()` method. The following example creates a new `ParameterBlock` named `pb` and then the `addSource()` method is used to add the source image (`im0`) to the `ParameterBlock`.

```
   ParameterBlock pb = new ParameterBlock();
   pb.addSource(im0);
```

To add two sources to a parameter block, use two `addSource()` methods.

```
   ParameterBlock pb = new ParameterBlock();
   pb.addSource(im0);
   pb.addSource(im1);
```

#### 3.7.2.2 Adding or Setting Parameters

As described before, there are two separate classes for specifying parameter blocks: `ParameterBlock` and `ParameterBlockJAI`. Both classes work very much alike, except for two differences: `ParameterBlockJAI` automatically provides default parameter values and allows setting parameters by name; `ParameterBlock` does not.

***ParameterBlock***

The operation parameters are added to a `ParameterBlock` with the `ParameterBlock.add()` method. The following example adds two values (`150` and `200`) to the `ParameterBlock` named `pb`, which was created in the previous example.

```
   pb.add(150);
   pb.add(200);
```

The `add()` method can be used with all of the supported data types: byte, short, integer, long, float, and double. When using the `ParameterBlock` object, all parameters that an operation requires must be added, else the operation will fail.

**API:** `java.awt.image.renderable.ParameterBlock`

- `ParameterBlock addSource(Object source)`

  adds an image to the end of the list of sources. The image is stored as an object to allow new node types in the future.

- `ParameterBlock add(byte b)`

  adds a Byte to the list of parameters.

- `ParameterBlock add(short s)`

  adds a Short to the list of parameters.

- `ParameterBlock add(int i)`

  adds an Integer to the list of parameters.

- `ParameterBlock add(long l)`

  adds a Long to the list of parameters.

- `ParameterBlock add(float f)`

  adds a Float to the list of parameters.

- `ParameterBlock add(double d)`

  adds a Double to the list of parameters.

### *ParameterBlockJAI*

Since the `ParameterBlockJAI` object already contains default values for the parameters at the time of construction, the parameters must be changed (or set) with the `ParameterBlockJAI.set(value, index)` methods rather than the `add()` method. The `add()` methods should not be used since the parameter list is already long enough to hold all of the parameters required by the `OperationDescriptor`.

Listing 3-3 shows the creation of a `ParameterBlockJAI` intended to be passed to a rotate operation. The rotate operation takes four parameters: `xOrigin`, `yOrigin`, `angle`, and `interpolation`. The default values for `xOrigin` and `yOrigin` are 0.0F (for both). In this example, these two values are not set, as the default values are sufficient for the operation. The other two parameters (`angle` and `interpolation`) have default values of `null` and must therefore be set. The source image must also be specified.

**Listing 3-3 Example ParameterBlockJAI**

```
// Specify the interpolation method to be used
interp = Interpolation.create(Interpolation.INTERP_NEAREST);

// Create the ParameterBlockJAI and add the interpolation to it
ParameterBlockJAI pb = new ParameterBlockJAI();
pb.addSource(im);                  // The source image
pb.set(1.2F, “angle”);           // The rotation angle in radians
pb.set(interp, “interpolation”);  // The interpolation method
```

**API:** `javax.media.jai.ParameterBlockJAI`

- `ParameterBlock set(byte b, String paramName)`
  sets a named parameter to a byte value.
- `ParameterBlock set(char c, String paramName)`
  sets a named parameter to a char value.
- `ParameterBlock set(int i, String paramName)`
  sets a named parameter to an int value.
- `ParameterBlock set(short s, String paramName)`
  sets a named parameter to a short value.
- `ParameterBlock set(long l, String paramName)`
  sets a named parameter to a long value.
- `ParameterBlock set(float f, String paramName)`
  sets a named parameter to a float value.
- `ParameterBlock set(double d, String paramName)`
  sets a named parameter to a double value.
- `ParameterBlock set(java.lang.Object obj, String paramName)`
  sets a named parameter to an Object value.

### 3.7.3 Rendering Hints

The rendering hints contain a set of hints that describe how objects are to be rendered. The rendering hints are always optional in any operation.

Rendering hints specify different rendering algorithms for such things as antialiasing, alpha interpolation, and dithering. Many of the hints allow a choice between rendering quality or speed. Other hints turn off or on certain rendering options, such as antialiasing and fractional metrics.

There are two separate classes for specifying rendering hints:

- `java.awt.RenderingHints` – contains rendering hints that can be used by the `Graphics2D` class, and classes that implement `BufferedImageOp` and `Raster`.
- `javax.media.jai.JAI` – provides methods to define the RenderingHints keys specific to JAI.

#### 3.7.3.1 Java AWT Rendering Hints

Table 3-12 lists the rendering hints inherited from `java.awt.RenderingHints`.

**Table 3-12 Java AWT Rendering Hints**

| Key | Value | Description |
|---|---|---|
| `Alpha_Interpolation` | `Alpha_Interpolation_Default` | Rendering is done with the platform default alpha interpolation. |
| | `Alpha_Interpolation_Quality` | Appropriate rendering algorithms are chosen with a preference for output quality. |
| | `Alpha_Interpolation_Speed` | Appropriate rendering algorithms are chosen with a preference for output speed. |
| `Antialiasing` | `Antialias_Default` | Rendering is done with the platform default antialiasing mode. |
| | `Antialias_Off` | Rendering is done without antialiasing. |
| | `Antialias_On` | Rendering is done with antialiasing |

**Table 3-12 Java AWT Rendering Hints (Continued)**

| Key | Value | Description |
|---|---|---|
| `Color_Rendering` | `Color_Render_Default` | Rendering is done with the platform default color rendering. |
| | `Color_Render_Quality` | Appropriate rendering algorithms are chosen with a preference for output quality. |
| | `Color_Render_Speed` | Appropriate rendering algorithms are chosen with a preference for output speed. |
| `Dithering` | `Dither_Default` | Use the platform default for dithering. |
| | `Dither_Disable` | Do not do dither when rendering. |
| | `Dither_Enable` | Dither with rendering when needed. |
| `FractionalMetrics` | `FractionalMetrics_Default` | Use the platform default for fractional metrics. |
| | `FractionalMetrics_Off` | Disable fractional metrics. |
| | `FractionalMetrics_On` | Enable fractional metrics. |
| `Interpolation` | `Interpolation_Bicubic` | Perform bicubic interpolation. |
| | `Interpolation_Bilinear` | Perform bilinear interpolation. |
| | `Interpolation_Nearest_Neighbor` | Perform nearest-neighbor interpolation. |
| `Rendering` | `Render_Default` | The platform default rendering algorithms will be chosen. |
| | `Render_Quality` | Appropriate rendering algorithms are chosen with a preference for output quality. |
| | `Render_Speed` | Appropriate rendering algorithms are chosen with a preference for output speed. |
| `Text_Antialiasing` | `Text_Antialias_Default` | Text rendering is done using the platform default text antialiasing mode. |
| | `Text_Antialias_Off` | Text rendering is done without antialiasing. |
| | `Text_Antialias_On` | Text rendering is done with antialiasing. |

To set the rendering hints, create a `RenderingHints` object and pass it to the `JAI.create` method you want to affect. Setting a rendering hint does not guarantee that a particular rendering algorithm, will be used; not all platforms support modification of the rendering code.

In the following example, the rendering preference is set to quality.

```
   qualityHints = new
                  RenderingHints(RenderingHints.KEY_RENDERING,
                  RenderingHints.VALUE_RENDER_QUALITY);
```

Now that a `RenderingHints` object, `qualityHints`, has been created, the hints can be used in an operation using a `JAI.create` method.

### 3.7.3.2 JAI Rendering Hints

Each instance of a `JAI` object contains a set of rendering hints that will be used for all image or collection creations. These hints are merged with any hints supplied to the `JAI.create` method; directly supplied hints take precedence over the common hints. When a new `JAI` instance is constructed, its hints are initialized to a copy of the hints associated with the default instance. The hints associated with any instance, including the default instance, may be manipulated using the `getRenderingHint`, `setRenderingHints`, and `clearRenderingHints` methods. As a convenience, `getRenderingHint`, `setRenderingHint`, and `removeRenderingHint` methods are provided that allow individual hints to be manipulated. Table 3-13 lists the JAI rendering hints.

**Table 3-13 JAI Rendering hints**

| Key | Value | Description |
|---|---|---|
| `HINT_BORDER_EXTENDER` | `BorderExtenderZero` | Extends an image's border by filling all pixels outside the image bounds with zeros. |
| | `BorderExtenderCon-stant` | Extends an image's border by filling all pixels outside the image bounds with constant values. |
| | `BorderExtenderCopy` | Extends an image's border by filling all pixels outside the image bounds with copies of the edge pixels. |
| | `BorderExtenderWrap` | Extends an image's border by filling all pixels outside the image bounds with copies of the whole image. |
| | `BorderExtenderReflect` | Extends an image's border by filling all pixels outside the image bounds with copies of the whole image. |

**Table 3-13 JAI Rendering hints (Continued)**

| Key | Value | Description |
|---|---|---|
| `HINT_IMAGE_LAYOUT` | `Width` | The image's width. |
| | `Height` | The image's height |
| | `MinX` | The image's minimum *x* coordinate. |
| | `MinY` | The image's minimum *y* coordinate |
| | `TileGridXOffset` | The *x* coordinate of tile (0, 0). |
| | `TileGridYOffset` | The *y* coordinate of tile (0, 0). |
| | `TileWidth` | The width of a tile. |
| | `TileHeight` | The height of a tile. |
| | `SampleModel` | The image's `SampleModel`. |
| | `ColorModel` | The image's `ColorModel`. |
| `HINT_INTERPOLATION` | `InterpolationNearest` | Perform nearest-neighbor interpolation. |
| | `InterpolationBilinear` | Perform bilinear interpolation. |
| | `InterpolationBicubic` | Perform bicubic interpolation. |
| | `InterpolationBicubic2` | Perform bicubic interpolation. |
| `HINT_OPERATION_BOUND` | `OpImage.OP_COMPUTE_BOUND` | An operation is likely to spend its time mainly performing computation. |
| | `OpImage.OP_IO_BOUND` | An operation is likely to spend its time mainly performing local I/O. |
| | `OpImage.OP_NETWORK_BOUND` | An operation is likely to spend its time mainly performing network I/O. |
| `HINT_OPERATION_REGISTRY` | | Key for `OperationRegistry` object values. |
| `HINT_PNG_EMIT_SQUARE_PIXELS` | `True` | Scale non-square pixels read from a PNG format image file to square pixels. |
| | `False` | Do not scale non-square pixels. |
| `HINT_TILE_CACHE` | `capacity` | The capacity of the cache in tiles. |
| | `elementCount` | The number of elements in the cache. |
| | `revolver` | Offset to check for tile cache victims. |
| | `multiplier` | Number of checks to make for tile cache victims. |

Listing 3-4 shows an example of image layout rendering hints being specified for a Scale operation. The image layout rendering hint specifies that the origin of the destination opimage is set to 200 × 200.

**Listing 3-4 Example of JAI Rendering Hints**

```
// Create the parameter block for the scale operation.
ParameterBlock pb = new ParameterBlock();
    pb.addSource(im0);      // The source image
    pb.add(4.0F);           // The x scale factor
    pb.add(4.0F);           // The y scale factor
    pb.add(interp);         // The interpolation method

// Specify the rendering hints.
    layout = new ImageLayout();
    layout.setMinX(200);
    layout.setMinY(200);
    RenderingHints rh =
            new RenderingHints(JAI.KEY_IMAGE_LAYOUT, layout);

// Create the scale operation.
PlanarImage im2 = (PlanarImage)JAI.create("scale", pb, layout)
```

CHAPTER 4

# Image Acquisition and Display

THIS chapter describes the Java Advanced Imaging (JAI) API image data types and the API constructors and methods for image acquisition and display.

## 4.1 Introduction

All imaging applications must perform the basic tasks of acquiring, displaying, and creating (recording) images. Images may be acquired from many sources, including a disk file, the network, a CD, and so on. Images may be acquired, processed, and immediately displayed, or written to a disk file for display at a later time.

As described in Chapter 3, JAI offers the programmer the flexibility to render and display an image immediately or to defer the display of the rendered image until there is a specific request for it.

Image acquisition and display are relatively easy in JAI, in spite of all the high-level information presented in the next several sections. Take for example, the sample code in Listing 4-1. This is a complete code example for a simple application called `FileTest`, which takes a single argument; the path and name of the file to read. `FileTest` reads the named file and displays it in a `ScrollingImagePanel`. The operator that reads the image file, `FileLoad`, is described in Section 4.4.1.2, “The FileLoad Operation.” The `ScrollingImagePanel` is described in Section 4.8, “Image Display.”

**Listing 4-1 Example Program to Read and Display an Image File**

```
// Specify the classes to import.
import java.awt.image.renderable.ParameterBlock;
import java.io.File;
import javax.media.jai.JAI;
import javax.media.jai.PlanarImage;
import javax.media.jai.RenderedOp;
import javax.media.jai.widget.ScrollingImagePanel;

public class FileTest extends WindowContainer {

// Specify a default image in case the user fails to specify
// one at run time.
public static final String DEFAULT_FILE =
                              "./images/earth.jpg";

    public static void main(String args[]) {
        String fileName = null;

// Check for a filename in the argument.
        if(args.length == 0) {
            fileName = DEFAULT_FILE;
        } else if(args.length == 1) {
            fileName = args[0];
        } else {
            System.out.println("\nUsage: java " +
                        (new FileTest()).getClass().getName() +
                              " [file]\n");
            System.exit(0);
        }

        new FileTest(fileName);
    }

    public FileTest() {}
    public FileTest(String fileName) {

   // Read the image from the designated path.
  System.out.println("Creating operation to load image from '" +
                      fileName+"'");
   RenderedOp img =  JAI.create("fileload", fileName);

   // Set display name and layout.
   setTitle(getClass().getName()+": "+fileName);
```

**Listing 4-1 Example Program to Read and Display an Image File (Continued)**

```
          // Display the image.
          System.out.println("Displaying image");
          add(new ScrollingImagePanel(img, img.getWidth(),
                                      img.getHeight()));
          pack();
          show();
      }
  }
```

## 4.1.1 Image Data

Image data is, conceptually, a three-dimensional array of pixels, as shown in Figure 4-1. Each of the three arrays in the example is called a *band*. The number of rows specifies the image height of a band, and the number of columns specifies the image width of a band.

Monochrome images, such as a grayscale image, have only one band. Color images have three or more bands, although a band does not necessarily have to represent color. For example, satellite images of the earth may be acquired in several different spectral bands, such as red, green, blue, and infrared.

In a color image, each band stores the red, green, and blue (RGB) components of an additive image, or the cyan, magenta, and yellow (CMY) components of a three-color subtractive image, or the cyan, magenta, yellow, and black (CMYK) components of a four-color subtractive image. Each pixel of an image is composed of a set of *samples*. For an RGB pixel, there are three samples; one each for red, green, and blue.

An image is sampled into a rectangular array of pixels. Each pixel has an (*x*,*y*) coordinate that corresponds to its location within the image. The *x* coordinate is the pixel's horizontal location; the *y* coordinate is the pixel's vertical location. Within JAI, the pixel at location (0,0) is in the upper left corner of the image, with the *x* coordinates increasing in value to the right and *y* coordinates increasing in value downward. Sometimes the *x* coordinate is referred to as the pixel number and the *y* coordinate as the line number.

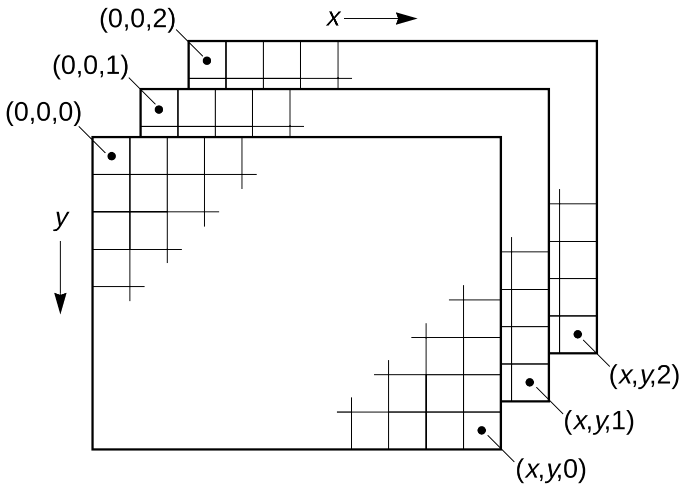


**Figure 4-1 Multi-band Image Structure**

## 4.1.2 Basic Storage Types

In the JAI API, the basic unit of data storage is the `DataBuffer` object. The `DataBuffer` object is a kind of raw storage that holds all the samples that make up the image, but does not contain any information on how those samples are put together as pixels. How the samples are put together is contained in a `SampleModel` object. The `SampleModel` class contains methods for deriving pixel data from a `DataBuffer`.

JAI supports several image data types, so the `DataBuffer` class has the following subclasses, each representing a different data type:

- `DataBufferByte` – stores data internally as bytes (8-bit values)
- `DataBufferShort` – stores data internally as shorts (16-bit values)
- `DataBufferUShort` – stores data internally as unsigned shorts (16-bit values)
- `DataBufferInt` – stores data internally as integers (32-bit values)
- `DataBufferFloat` – stores data internally as single-precision floating-point values.
- `DataBufferDouble` – stores data internally as double-precision floating-point values.

Table 4-1 lists the `DataBuffer` type elements.

**Table 4-1 Data Buffer Type Elements**

| Name | Description |
|---|---|
| `TYPE_INT` | Tag for int data. |
| `TYPE_BYTE` | Tag for unsigned byte data. |
| `TYPE_SHORT` | Tag for signed short data. |
| `TYPE_USHORT` | Tag for unsigned short data. |
| `TYPE_DOUBLE` | Tag for double data. |
| `TYPE_FLOAT` | Tag for float data. |
| `TYPE_UNDEFINED` | Tag for undefined data. |

JAI also supports a large number of image data formats, so the `SampleModel` class provides the following types of sample models:

- `ComponentSampleModel` – used to extract pixels from images that store sample data in separate data array elements in one bank of a `DataBuffer` object.
- `ComponentSampleModelJAI` – used to extract pixels from images that store sample data such that each sample of a pixel occupies one data element of the `DataBuffer`.
- `BandedSampleModel` – used to extract pixels from images that store each sample in a separate data element with bands stored in a sequence of data elements.
- `PixelInterleavedSampleModel` – used to extract pixels from images that store each sample in a separate data element with pixels stored in a sequence of data elements.
- `MultiPixelPackedSampleModel` – used to extract pixels from single-banded images that store multiple one-sample pixels in one data element.
- `SinglePixelPackedSampleModel` – used to extract samples from images that store sample data for a single pixel in one data array element in the first bank of a `DataBuffer` object.
- `FloatComponentSampleModel` – stores *n* samples that make up a pixel in *n* separate data array elements, all of which are in the same bank in a `DataBuffer` object. This class supports different kinds of interleaving.

The combination of a `DataBuffer` object, a `SampleModel` object, and an origin constitute a meaningful multi-pixel image storage unit called a `Raster`. The

`Raster` class has methods that directly return pixel data for the image data it contains.

There are two basic `Raster` types:

- `Raster` – represents a rectangular array of pixels. This is a “read-only” class that only has get methods.
- `WritableRaster` – extends `Raster` to provide pixel writing capabilities.

There are separate interfaces for dealing with each raster type:

- The `RenderedImage` interface assumes the data is read-only and, therefore, does not contain methods for writing a `Raster`.
- The `WriteableRenderedImage` interfaces assumes that the image data can be modified.

A `ColorModel` class provides a color interpretation of pixel data provided by the image’s sample model. The abstract `ColorModel` class defines methods for turning an image’s pixel data into a color value in its associated `ColorSpace`. See Section 5.2.1, “Color Models.”

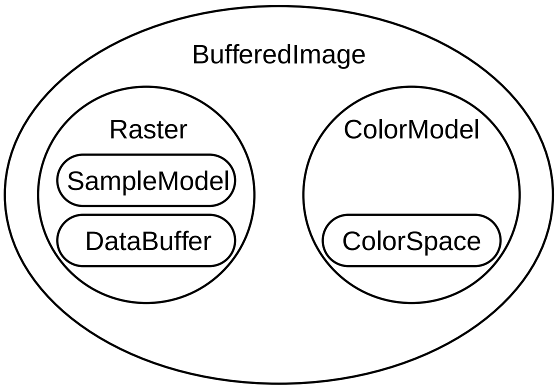


**Figure 4-2 BufferedImage**

As shown in Figure 4-2, the combination of a `Raster` and a `ColorModel` define a `BufferedImage`. The `BufferedImage` class provides general image management for immediate mode imaging.

The `BufferedImage` class supports the following predefined image types:

**Table 4-2 Supported Image Types**

| Name | Description |
|---|---|
| `TYPE_3BYTE_BGR` | Represents an image with 8-bit RGB color components, corresponding to a Windows-style BGR color model, with the colors blue, green, and red stored in three bytes. |
| `TYPE_4BYTE_ABGR` | Represents an image with 8-bit RGBA color components with the colors blue, green, and red stored in three bytes and one byte of alpha. |
| `TYPE_4BYTE_ABGR_PRE` | Represents an image with 8-bit RGBA color components with the colors blue, green, and red stored in three bytes and one byte of alpha. |
| `TYPE_BYTE_BINARY` | Represents an opaque byte-packed binary image. |
| `TYPE_BYTE_GRAY` | Represents a unsigned byte grayscale image, non-indexed. |
| `TYPE_BYTE_INDEXED` | Represents an indexed byte image. |
| `TYPE_CUSTOM` | Image type is not recognized so it must be a customized image. |
| `TYPE_INT_ARGB` | Represents an image with 8-bit RGBA color components packed into integer pixels. |
| `TYPE_INT_ARGB_PRE` | Represents an image with 8-bit RGB color components, corresponding to a Windows- or Solaris- style BGR color model, with the colors blue, green, and red packed into integer pixels. |
| `TYPE_INT_BGR` | Represents an image with 8-bit RGB color components, corresponding to a Windows- or Solaris- style BGR color model, with the colors blue, green, and red packed into integer pixels. |
| `TYPE_INT_RGB` | Represents an image with 8-bit RGB color components packed into integer pixels. |
| `TYPE_USHORT_555_RGB` | Represents an image with 5-5-5 RGB color components (5-bits red, 5-bits green, 5-bits blue) with no alpha. |
| `TYPE_USHORT_565_RGB` | Represents an image with 5-6-5 RGB color components (5-bits red, 6-bits green, 5-bits blue) with no alpha. |
| `TYPE_USHORT_GRAY` | Represents an unsigned short grayscale image, non-indexed). |

## 4.2 JAI Image Types

The JAI API provides a set of classes for describing image data of various kinds. These classes are organized into a class hierarchy, as shown in Figure 4-3.

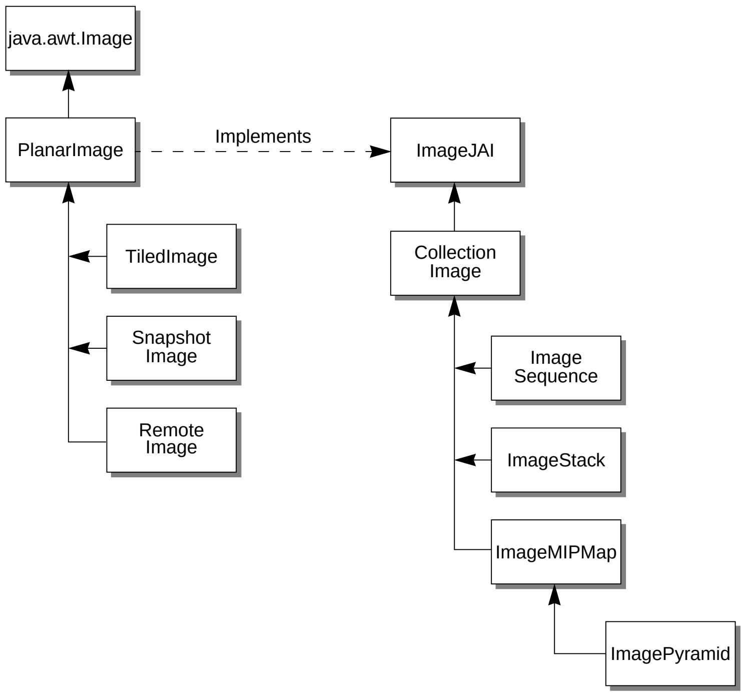


**Figure 4-3 JAI Image Type Hierarchy**

### 4.2.1 Planar Image

The `PlanarImage` class is the main class for defining two-dimensional images. The `PlanarImage` implements the `java.awt.image.RenderedImage` interface, which describes a tiled, read-only image with a pixel layout described by a `SampleModel` and a `DataBuffer`. The `TiledImage` and `OpImage` subclasses manipulate the instance variables they inherit from `PlanarImage`, such as the image size, origin, tile dimensions, and tile grid offsets, as well as the Vectors containing the sources and sinks of the image.

All non-JAI `RenderedImages` that are to be used in JAI must be converted into `PlanarImages` by means of the `RenderedImageAdapter` class and the `WriteableRenderedImageAdapter` class. The `wrapRenderedImage()` method provides a convenient interface to both add a wrapper and take a snapshot if the image is writable. The standard `PlanarImage` constructor used by `OpImages` performs this wrapping automatically. Images that already extend `PlanarImage` will be returned unchanged by `wrapRenderedImage()`.

Going in the other direction, existing code that makes use of the `RenderedImage` interface will be able to use `PlanarImages` directly, without any changes or recompilation. Therefore within JAI, images are returned from methods as `PlanarImages`, even though incoming `RenderedImages` are accepted as arguments directly.

**API:** `javax.media.jai.PlanarImage`

- `PlanarImage()`

  creates a `PlanarImage`.

- `static PlanarImage wrapRenderedImage(RenderedImage im)`

  wraps an arbitrary `RenderedImage` to produce a `PlanarImage`. `PlanarImage` adds various properties to an image, such as source and sink vectors and the ability to produce snapshots, that are necessary for JAI. If the image is not a `PlanarImage`, it is wrapped in a `RenderedImageAdapter`. If the image implements `WritableRenderedImage`, a snapshot is taken.

  | | | |
  |---|---|---|
  | *Parameters*: | `a` | `RenderedImage` to be used as a synchronous source. |

- `PlanarImage createSnapshot()`

  creates a snapshot, that is, a virtual copy of the image's current contents.

- `Raster getData(Rectangle region)`

  returns a specified region of this image in a `Raster`.

  | | | |
  |---|---|---|
  | *Parameter*: | `region` | The rectangular region of this image to be returned. |

- `int getWidth()`

  returns the width of the image.

- `int getHeight()`

  returns the height of the image.

- `int getMinXCoord()`

  returns the X coordinate of the leftmost column of the image.

- `int getMaxXCoord()`

  returns the X coordinate of the rightmost column of the image.

- `int getMinYCoord()`
  returns the X coordinate of the uppermost row of the image.
- `int getMaxYCoord()`
  returns the X coordinate of the bottom row of the image.
- `Rectangle getBounds()`
  returns a Rectangle indicating the image bounds.
- `int getTileWidth()`
  returns the width of a tile.
- `int getTileHeight()`
  returns the height of a tile.
- `int tilesAcross()`
  returns the number of tiles along the tile grid in the horizontal direction. Equivalent to `getMaxTileX() - getMinTileX() + 1`.
- `int tilesDown()`
  returns the number of tiles along the tile grid in the vertical direction. Equivalent to `getMaxTileY() - getMinTileY() + 1`.

There are lots more methods.

### 4.2.2 Tiled Image

The JAI API expands on the tile data concept introduced in the Java 2D API. In Java 2D, a tile is one of a set of rectangular regions that span an image on a regular grid. The JAI API expands on the tile image with the `TiledImage` class, which is the main class for writable images in JAI.

A tile represents all of the storage for its spatial region of the image. If an image contains three bands, every tile represents all three bands of storage. The use of tiled images improves application performance by allowing the application to process an image region within a single tile without bringing the entire image into memory.

`TiledImage` provides a straightforward implementation of the `WritableRenderedImage` interface, taking advantage of that interface's ability to describe images with multiple tiles. The tiles of a `WritableRenderedImage` must share a `SampleModel`, which determines their width, height, and pixel format.

The tiles form a regular grid that may occupy any rectangular region of the plane. Tile pixels that exceed the image's stated bounds have undefined values.

The contents of a `TiledImage` are defined by a single `PlanarImage` source, provided either at construction time or by means of the `set()` method. The `set()` method provides a way to selectively overwrite a portion of a `TiledImage`, possibly using a soft-edged mask.

`TiledImage` also supports direct manipulation of pixels by means of the `getWritableTile` method. This method returns a `WritableRaster` that can be modified directly. Such changes become visible to readers according to the regular thread synchronization rules of the Java virtual machine; JAI makes no additional guarantees. When a writer is finished modifying a tile, it should call the `releaseWritableTile` method. A shortcut is to call the `setData()` method, which copies a rectangular region from a supplied `Raster` directly into the `TiledImage`.

A final way to modify the contents of a `TiledImage` is through calls to the `createGraphics()` method. This method returns a `GraphicsJAI` object that can be used to draw line art, text, and images in the usual AWT manner.

A `TiledImage` does not attempt to maintain synchronous state on its own. That task is left to `SnapshotImage`. If a synchronous (unchangeable) view of a `TiledImage` is desired, its `createSnapshot()` method must be used. Otherwise, changes due to calls to `set()` or direct writing of tiles by objects that call `getWritableTile()` will be visible.

`TiledImage` does not actually cause its tiles to be computed until their contents are demanded. Once a tile has been computed, its contents may be discarded if it can be determined that it can be recomputed identically from the source. The `lockTile()` method forces a tile to be computed and maintained for the lifetime of the `TiledImage`.

**API:** `javax.media.jai.TiledImage`

-
  ```
  TiledImage(Point origin, SampleModel sampleModel,
      int tileWidth, int tileHeight)
  ```

  constructs a `TiledImage` with a `SampleModel` that is compatible with a given `SampleModel`, and given tile dimensions. The width and height are taken from the `SampleModel`, and the image begins at a specified point.

| | | |
|---|---|---|
| *Parameters*: | `origin` | A Point indicating the image's upper left corner. |
| | `sampleModel` | A SampleModel with which to be compatible. |
| | `tileWidth` | The desired tile width. |
| | `tileHeight` | The desired tile height. |

-
  ```
  TiledImage(SampleModel sampleModel, int tileWidth,
      int tileHeight)
  ```

  constructs a `TiledImage` starting at the global coordinate origin.

| | | |
|---|---|---|
| *Parameters*: | `sampleModel` | A `SampleModel` with which to be compatible. |
| | `tileWidth` | The desired tile width. |
| | `tileHeight` | The desired tile height. |

- `TiledImage(int minX, int minY, int width, int height,`
  `int tileGridXOffset, int tileGridYOffset,`
  `SampleModel sampleModel, ColorModel colorModel)`

  constructs a `TiledImage` of a specified width and height.

| | | |
|---|---|---|
| *Parameters*: | `minX` | The index of the leftmost column of tiles. |
| | `minY` | The index of the uppermost row of tiles. |
| | `width` | The width of the `TiledImage`. |
| | `height` | The height of the `TiledImage`. |
| | `tileGridX-Offset` | The *x* coordinate of the upper-left pixel of tile (0, 0). |
| | `tileGridY-Offset` | The *y* coordinate of the upper-left pixel of tile (0, 0). |
| | `sampleModel` | a `SampleModel` with which to be compatible. |
| | `colorModel` | A `ColorModel` to associate with the image. |

- `void setData(Raster r)`

  sets a region of a `TiledImage` to be a copy of a supplied `Raster`. The `Raster`'s coordinate system is used to position it within the image. The computation of all overlapping tiles will be forced prior to modification of the data of the affected area.

| | | |
|---|---|---|
| *Parameter*: | `r` | A `Raster` containing pixels to be copied into the `TiledImage`. |

- `void setData(Raster r, ROI roi)`

  sets a region of a `TiledImage` to be a copy of a supplied `Raster`. The `Raster`'s coordinate system is used to position it within the image. The computation of all overlapping tiles will be forced prior to modification of the data of the affected area.

- `WritableRaster getWritableTile(int tileX, int tileY)`

  retrieves a particular tile from the image for reading and writing. The tile will be computed if it hasn't been previously. Writes to the tile will become visible to readers of this image in the normal Java manner.

| | | |
|---|---|---|
| *Parameters*: | `tileX` | The *x* index of the tile. |
| | `tileY` | The *y* index of the tile. |

- `Raster getTile(int tileX, int tileY)`

  retrieves a particular tile from the image for reading only. The tile will be computed if it hasn't been previously. Any attempt to write to the tile will produce undefined results.

| | | |
|---|---|---|
| *Parameters*: | `tileX` | The *x* index of the tile. |
| | `tileY` | The *y* index of the tile. |

- `boolean isTileWritable(int tileX, int tileY)`

  returns true if a tile has writers.

| | | |
|---|---|---|
| *Parameters*: | `tileX` | The *x* index of the tile. |
| | `tileY` | The *y* index of the tile. |

- `boolean hasTileWriters()`

  returns true if any tile is being held by a writer, false otherwise. This provides a quick way to check whether it is necessary to make copies of tiles – if there are no writers, it is safe to use the tiles directly, while registering to learn of future writers.

- `void releaseWritableTile(int tileX, int tileY)`

  indicates that a writer is done updating a tile. The effects of attempting to release a tile that has not been grabbed, or releasing a tile more than once are undefined.

| | | |
|---|---|---|
| *Parameters*: | `tileX` | The *x* index of the tile. |
| | `tileY` | The *y* index of the tile. |

- `void set(RenderedImage im)`

  overlays a given `RenderedImage` on top of the current contents of the `TiledImage`. The source image must have a `SampleModel` compatible with that of this image.

| | | |
|---|---|---|
| *Parameters*: | `im` | A `RenderedImage` source to replace the current source. |

- `void set(RenderedImage im, ROI roi)`

  overlays a given `RenderedImage` on top of the current contents of the `TiledImage`. The source image must have a `SampleModel` compatible with that of this image.

| | | |
|---|---|---|
| *Parameters*: | `im` | A `RenderedImage` source to replace the current source. |
| | `roi` | The region of interest. |

- `Graphics2D createGraphics()`

  creates a `Graphics2D` object that can be used to paint text and graphics onto the `TiledImage`.

#### 4.2.2.1 Tile Cache

The `TileCache` interface provides a central place for `OpImages` to cache tiles they have computed. The tile cache is created with a given capacity (measured in tiles). By default, the tile capacity for a new tile cache is 300 tiles. The default memory capacity reserved for tile cache is 20M bytes.

The `TileCache` to be used by a particular operation may be set during construction, or by calling the `JAI.setTileCache` method. This results in the provided tile cache being added to the set of common rendering hints.

The `TileScheduler` interface allows tiles to be scheduled for computation. In various implementations, tile computation may make use of multithreading and multiple simultaneous network connections for improved performance.

**API:** `javax.media.jai`

-
  ```
  static TileCache createTileCache(int tileCapacity,
      long memCapacity)
  ```

  constructs a `TileCache` with the given tile capacity in tiles and memory capacity in bytes. Users may supply an instance of `TileCache` to an operation by supplying a `RenderingHint` with a `JAI.KEY_TILE_CACHE` key and the desired `TileCache` instance as its value. Note that the absence of a tile cache hint will result in the use of the `TileCache` belonging to the default `JAI` instance. To force an operation not to perform caching, a `TileCache` instance with a tile capacity of 0 may be used.

| | | |
|---|---|---|
| *Parameters* | `tileCapacity` | The tile capacity, in tiles. |
| | `memCapacity` | The memory capacity, in bytes. |

- `static TileCache createTileCache()`

  constructs a `TileCache` with the default tile capacity in tiles and memory capacity in bytes.

- `void setTileCache(TileCache tileCache)`

  sets the `TileCache` to be used by this `JAI` instance. The `tileCache` parameter will be added to the `RenderingHints` of this JAI instance.

- `TileCache getTileCache()`

  returns the `TileCache` being used by this `JAI` instance.

#### 4.2.2.2 Pattern Tiles

A pattern tile consists of a repeated pattern. The `pattern` operation defines a pattern tile by specifying the width and height; all other layout parameters are optional, and when not specified are set to default values. Each tile of the destination image will be defined by a reference to a shared instance of the pattern.

The `pattern` operation takes three parameters:

| **Parameter** | **Type** | **Description** |
|---|---|---|
| `width` | `Integer` | The width of the image in pixels. |
| `height` | `Integer` | The height of the image in pixels. |
| `pattern` | `Raster` | The Pattern pixel band values. |

Listing 4-2 shows a code sample for a `pattern` operation.

**Listing 4-2 Example Pattern Operation**

```
    // Create the raster.
    WritableRaster raster;
    int[] bandOffsets = new int[3];
    bandOffsets[0] = 2;
    bandOffsets[1] = 1;
    bandOffsets[2] = 0;

    // width, height=64.
    PixelInterleavedSampleModel sm;
    sm = new PixelInterleavedSampleModel(DataBuffer.TYPE_BYTE, 100,
                                           100, 3, 3*100, bandOffsets);
```

**Listing 4-2 Example Pattern Operation (Continued)**

```
// Origin is 0,0.
WritableRaster pattern = Raster.createWritableRaster(sm,
                             new Point(0, 0));
int[] bandValues = new int[3];
bandValues[0] = 90;
bandValues[1] = 45;
bandValues[2] = 45

// Set values for the pattern raster.
for (int y = 0; y < pattern.getHeight(); y++) {
for (int x = 0; x < pattern.getWidth(); x++) {
    pattern.setPixel(x, y, bandValues);
    bandValues[1] = (bandValues[1]+1)%255;
    bandValues[2] = (bandValues[2]+1)%255;
    }
}

// Create a 100x100 image with the given raster.
PlanarImage im0 = (PlanarImage)JAI.create("pattern",
                                          100, 100,
                                          pattern);
```

## 4.2.3 Snapshot Image

The `SnapshotImage` class represents the main component of the deferred execution engine. A `SnapshotImage` provides an arbitrary number of synchronous views of a possibly changing `WritableRenderedImage`. `SnapshotImage` is responsible for stabilizing changing sources to allow deferred execution of operations dependent on such sources.

Any `RenderedImage` may be used as the source of a `SnapshotImage`. If the source is a `WritableRenderedImage`, the `SnapshotImage` will register itself as a `TileObserver` and make copies of tiles that are about to change.

Multiple versions of each tile are maintained internally, as long as they are in demand. `SnapshotImage` is able to track demand and should be able to simply forward requests for tiles to the source most of the time, without the need to make a copy.

When used as a source, calls to `getTile` will simply be passed along to the source. In other words, `SnapshotImage` is completely transparent. However, by calling `createSnapshot()` an instance of a non-public `PlanarImage` subclass (called `Snapshot` in this implementation) will be created and returned. This image will always return tile data with contents as of the time of its construction.

#### 4.2.3.1 Creating a SnapshotImage

This implementation of `SnapshotImage` makes use of a doubly-linked list of `Snapshot` objects. A new `Snapshot` is added to the tail of the list whenever `createSnapshot()` is called. Each `Snapshot` has a cache containing copies of any tiles that were writable at the time of its construction, as well as any tiles that become writable between the time of its construction and the construction of the next `Snapshot`.

#### 4.2.3.2 Using SnapshotImage with a Tile

When asked for a tile, a `Snapshot` checks its local cache and returns its version of the tile if one is found. Otherwise, it forwards the request onto its successor. This process continues until the latest Snapshot is reached; if it does not contain a copy of the tile, the tile is requested from the real source image.

---

**API:** `javax.media.jai.SnapShotImage`

---

- `SnapshotImage(PlanarImage source)`

  constructs a `SnapshotImage` from a `PlanarImage` source.

| | | |
|---|---|---|
| *Parameters*: | `source` | a `PlanarImage` source. |

- `Raster getTile(int tileX, int tileY)`

  returns a non-snapshotted tile from the source.

| | | |
|---|---|---|
| *Parameters*: | `tileX` | the X index of the tile. |
| | `tileY` | the Y index of the tile. |

-
  ```
  void tileUpdate(java.awt.image.WritableRenderedImage source,
      int tileX, int tileY, boolean willBeWritable)
  ```

  receives the information that a tile is either about to become writable, or is about to become no longer writable.

| | | |
|---|---|---|
| *Parameters*: | `source` | the `WritableRenderedImage` for which we are an observer. |
| | `tileX` | the *x* index of the tile. |
| | `tileY` | the *y* index of the tile. |
| | `willBeWrit-able` | true if the tile is becoming writable. |

- `PlanarImage createSnapshot()`

  creates a snapshot of this image. This snapshot may be used indefinitely, and will always appear to have the pixel data that this image has currently. The snapshot is semantically a copy of this image but may be implemented in a more efficient manner. Multiple snapshots taken at different times may share tiles that have not changed, and tiles that are currently static in this image's source do not need to be copied at all.

#### 4.2.3.3 Disposing of a Snapshot Image

When a `Snapshot` is no longer needed, its `dispose()` method may be called. The `dispose()` method will be called automatically when the `Snapshot` is finalized by the garbage collector. The `dispose()` method attempts to push the contents of its tile cache back to the previous `Snapshot` in the linked list. If that image possesses a version of the same tile, the tile is not pushed back and may be discarded.

Disposing of the `Snapshot` allows tile data held by the Snapshot that is not needed by any other `Snapshot` to be disposed of as well.

**API:** `javax.media.jai.PlanarImage`

- `void dispose()`

  provides a hint that an image will no longer be accessed from a reference in user space. The results are equivalent to those that occur when the program loses its last reference to this image, the garbage collector discovers this, and finalize is called. This can be used as a hint in situations where waiting for garbage collection would be overly conservative.

### 4.2.4 Remote Image

A `RemoteImage` is a sub-class of `PlanarImage` which represents an image on a remote server. A `RemoteImage` may be constructed from a `RenderedImage` or from an imaging chain in either the rendered or renderable modes. For more information, see **Chapter 12, "Client-Server Imaging**."

### 4.2.5 Collection Image

The `CollectionImage` class is an abstract superclass for classes representing groups of images. Examples of groups of images include pyramids (`ImagePyramid`), time sequences (`ImageSequence`), and planar slices stacked to form a volume (`ImageStack`).

**API:** `javax.media.jai.CollectionImage`

- `CollectionImage()`
  the default constructor.

- `CollectionImage(java.util.Collection images)`
  constructs a `CollectionImage` object from a Vector of `ImageJAI` objects.

  | | | |
  |---|---|---|
  | *Parameters*: | `images` | A Vector of `ImageJAI` objects. |

### 4.2.6 Image Sequence

The `ImageSequence` class represents a sequence of images with associated timestamps and a camera position. It can be used to represent video or time-lapse photography.

The images are of the type `ImageJAI`. The timestamps are of the type `long`. The camera positions are of the type `Point`. The tuple (image, time stamp, camera position) is represented by class `SequentialImage`.

**API:** `javax.media.jai.ImageSequence`

- `ImageSequence(Collection images)`
  constructs a class that represents a sequence of images from a collection of `SequentialImage`.

### 4.2.7 Image Stack

The `ImageStack` class represents a stack of images, each with a defined spatial orientation in a common coordinate system. This class can be used to represent CT scans or seismic volumes.

The images are of the type `javax.media.jai.PlanarImage`; the coordinates are of the type `javax.media.jai.Coordinate`. The tuple (image, coordinate) is represented by class `javax.media.jai.CoordinateImage`.

**API:** `javax.media.jai.ImageStack`

- `ImageStack(Collection images)`
  constructs an `ImageStack` object from a collection of `CoordinateImage`.

- `ImageJAI getImage(Coordinate coordinate)`
  returns the image associated with the specified coordinate.

- `Coordinate getCoordinate(ImageJAI image)`
  returns the coordinate associated with the specified image.

## 4.2.8 Image MIP Map

An image MIP map is a stack of images with a fixed operational relationship between adjacent slices. Given the highest-resolution slice, the others may be derived in turn by performing a particular operation. Data may be extracted slice by slice or by special iterators.

A MIP map image (*MIP* stands for the Latin *multim im parvo*, meaning "many things in a small space") is usually associated with texture mapping. In texture mapping, the MIP map image contains different-sized versions of the same image in one location. To use mipmapping for texture mapping, you provide all sizes of the image in powers of 2 from the largest image to a $1 \times 1$ map.

The `ImageMIPMap` class takes the original source image at the highest resolution level, considered to be level 0, and a RenderedOp chain that defines how the image at the next lower resolution level is derived from the current resolution level.

The RenderedOp chain may have multiple operations, but the first operation in the chain must take only one source image, which is the image at the current resolution level.

There are three `ImageMIPMap` constructors:

-
  ```
  ImageMIPMap(RenderedImage image, AffineTransform transform,
      Interpolation interpolation)
  ```
  This constructor assumes that the operation used to derive the next lower resolution is a standard *affine* operation.

| | | |
|---|---|---|
| *Parameters*: | `image` | The image at the highest resolution level. |
| | `transform` | The affine transform matrix used by "affine" operation. |
| | `interpolation` | The interpolation method used by "affine" operation. |

  Any number of versions of the original image may be derived by an affine transform representing the geometric relationship between levels of the MIP map. The affine transform may include translation, scaling, and rotation (see "Affine Transformation" on page 272).

- `ImageMIPMap(RenderedImage image, RenderedOp downSampler)`

  This constructor specifies the `downSampler`, which points to the RenderedOp chain used to derive the next lower resolution level.

| *Parameters*: | `image` | The image at the highest resolution level. |
|---|---|---|
| | `downsampler` | The RenderedOp chain used to derive the next lower resolution level. The first operation of this chain must take one source, but must not have a source specified. |

- `ImageMIPMap(RenderedOp downSampler)`

  This constructor specifies only the `downSampler`.

The `downSampler` is a chain of operations used to derive the image at the next lower resolution level from the image at the current resolution level. That is, given an image at resolution level *i*, the `downSampler` is used to obtain the image at resolution level *i* + 1. The chain may contain one or more operation nodes; however, each node must be a `RenderedOp`.

The `downsampler` parameter points to the last node in the chain. The very first node in the chain must be a `RenderedOp` that takes one `RenderedImage` as its source. All other nodes may have multiple sources. When traversing back up the chain, if a node has more than one source, the first source, `source0`, is used to move up the chain. This parameter is saved by reference.

Listing 4-3 shows a complete code example of the use of `ImageMIPMap`.

**Listing 4-3 Example use of ImageMIPMap (Sheet 1 of 3)**

```
import java.awt.geom.AffineTransform;
import java.awt.image.RenderedImage;
import java.awt.image.renderable.ParameterBlock;
import javax.media.jai.JAI;
import javax.media.jai.Interpolation;
import javax.media.jai.InterpolationNearest;
import javax.media.jai.ImageMIPMap;
import javax.media.jai.PlanarImage;
import javax.media.jai.RenderedOp;
import com.sun.media.jai.codec.FileSeekableStream;

public class ImageMIPMapTest extends Test {
```

**Listing 4-3 Example use of ImageMIPMap (Sheet 2 of 3)**

```
protected static String
  file = “/import/jai/JAI_RP/src/share/sample/images/pond.jpg”;

protected Interpolation interp = new InterpolationNearest();

protected ImageMIPMap MIPMap;

protected RenderedOp image;
protected RenderedOp downSampler;

private void test1() {
AffineTransform at = new AffineTransform(0.8, 0.0, 0.0, 0.8,
                                         0.0, 0.0);
InterpolationNearest interp = new InterpolationNearest();

MIPMap = new ImageMIPMap(image, at, interp);

   display(MIPMap.getDownImage());
   display(MIPMap.getImage(4));
   display(MIPMap.getImage(1));
    }

public void test2() {
downSampler = createScaleOp(image, 0.9F);
downSampler.removeSources();
downSampler = createScaleOp(downSampler, 0.9F);

MIPMap = new ImageMIPMap(image, downSampler);

display(MIPMap.getImage(0));
display(MIPMap.getImage(5));
display(MIPMap.getImage(2));
}

public void test3() {
    downSampler = createScaleOp(image, 0.9F);
    downSampler = createScaleOp(downSampler, 0.9F);

MIPMap = new ImageMIPMap(downSampler);

    display(MIPMap.getImage(5));
    System.out.println(MIPMap.getCurrentLevel());
    display(MIPMap.getCurrentImage());
    System.out.println(MIPMap.getCurrentLevel());
    display(MIPMap.getImage(1));
    System.out.println(MIPMap.getCurrentLevel());
}
```

**Listing 4-3 Example use of ImageMIPMap (Sheet 3 of 3)**

```
   protected RenderedOp createScaleOp(RenderedImage src,
                                      float factor) {
      ParameterBlock pb = new ParameterBlock();
      pb.addSource(src);
      pb.add(factor);
      pb.add(factor);
      pb.add(1.0F);
      pb.add(1.0F);
      pb.add(interp);
      return JAI.create("scale", pb);
   }

   public ImageMIPMapTest(String name) {
          super(name);

      try {
         FileSeekableStream stream = new FileSeekableStream(file);
          image = JAI.create("stream", stream);
      } catch (Exception e) {
          System.exit(0);
      }
   }

   public static void main(String args[]) {
       ImageMIPMapTest test = new ImageMIPMapTest("ImageMIPMap");
       // test.test1();
       // test.test2();
       test.test3();
     }
   }
```

**API:** `javax.media.jai.ImageMIPMap`

- `int getCurrentLevel()`

  returns the current resolution level. The highest resolution level is defined as level 0.

- `RenderedImage getCurrentImage()`

  returns the image at the current resolution level.

- `RenderedImage getImage(int level)`

  returns the image at the specified resolution level. The requested level must be greater than or equal to the current resolution level or `null` will be returned.

- `RenderedImage getDownImage()`

  returns the image at the next lower resolution level, obtained by applying the `downSampler` on the image at the current resolution level.

### 4.2.9 Image Pyramid

The `ImagePyramid` class implements a pyramid operation on a `RenderedImage`. Supposing that we have a `RenderedImage` of 1024 × 1024, we could generate ten additional images by successively averaging 2 × 2 pixel blocks, each time discarding every other row and column of pixels. We would be left with images of 512 × 512, 256 × 256, and so on down to 1 × 1.

In practice, the lower-resolution images may be derived by performing any chain of operations to repeatedly down sample the highest-resolution image slice. Similarly, once a lower resolution image slice is obtained, the higher resolution image slices may be derived by performing another chain of operations to repeatedly up sample the lower resolution image slice. Also, a third operation chain may be used to find the difference between the original slice of image and the resulting slice obtained by first down sampling then up sampling the original slice.

This brings us to the discussion of the parameters required of this class:

| **Parameter** | **Description** |
|---|---|
| `downSampler` | A RenderedOp chain used to derive the lower resolution images. The first operation in the chain must take only one source. See Section 4.2.9.1, "The Down Sampler." |
| `upSampler` | A RenderedOp chain that derives the image at a resolution level higher than the current level. The first operation in the chain must take only one source. See Section 4.2.9.2, "The Up Sampler." |
| `differencer` | A RenderedOp chain that finds the difference of two images. The first operation in the chain must take exactly two sources. See Section 4.2.9.3, "The Differencer." |
| `combiner` | A RenderedOp chain that combines two images. The first operation in the chain must take exactly two sources. See Section 4.2.9.4, "The Combiner." |

Starting with the image at the highest resolution level, to find an image at a lower resolution level we use the `downSampler`. But, at the same time we also use the `upSampler` to retrieve the image at the higher resolution level, then use the `differencer` to find the difference image between the original image and the derived image from the `upSampler`. We save this difference image for later use.

To find an image at a higher resolution, we use the `upSampler`, then combine the earlier saved difference image with the resulting image using the `combiner` to get the final higher resolution level.

For example:

> We have an image at level *n*
> *n* + 1 = downSampler(*n*)
> diff *n* = upSampler(*n* + 1)
> *diff n* = differencer(*n*, *n*') — This diff *n* is saved for each level
> Later we want to get *n* from *n* + 1
> *n*' = upSampler(*n* + 1)
> *n* = combiner(*n*', diff *n*)

### 4.2.9.1 The Down Sampler

The `downSampler` is a chain of operations used to derive the image at the next lower resolution level from the image at the current resolution level. That is, given an image at resolution level *i*, the `downSampler` is used to obtain the image at resolution level *i* + 1. The chain may contain one or more operation nodes; however, each node must be a `RenderedOp`. The parameter points to the last node in the chain. The very first node in the chain must be a `RenderedOp` that takes one `RenderedImage` as its source. All other nodes may have multiple sources. When traversing back up the chain, if a node has more than one source, the first source, `source0`, is used to move up the chain. This parameter is saved by reference.

The `getDownImage` method returns the image at the next lower resolution level, obtained by applying the `downSampler` on the image at the current resolution level.

### 4.2.9.2 The Up Sampler

The `upSampler` is a chain of operations used to derive the image at the next higher resolution level from the image at the current resolution level. That is, given an image at resolution level *i*, the `upSampler` is used to obtain the image at resolution level *i* – 1. The requirement for this parameter is similar to the requirement for the `downSampler` parameter.

The `getUpImage` method returns the image at the previous higher resolution level. If the current image is already at level 0, the current image is returned without further up sampling. The down-sampled image is obtained by first up sampling the current image, then combining the resulting image with the previously-saved different image using the combiner op chain (see Section 4.2.9.4, “The Combiner”).

#### 4.2.9.3 The Differencer

The differencer is a chain of operations used to find the difference between an image at a particular resolution level and the image obtained by first down sampling that image then up sampling the result image of the down sampling operations. The chain may contain one or more operation nodes; however, each node must be a RenderedOp. The parameter points to the last node in the chain. The very first node in the chain must be a RenderedOp that takes two RenderedImages as its sources. When traversing back up the chain, if a node has more than one source, the first source, source0, is used to move up the chain. This parameter is saved by reference.

The getDiffImage method returns the difference image between the current image and the image obtained by first down sampling the current image then up sampling the resulting image of down sampling. This is done using the differencer op chain. The current level and current image are not changed.

#### 4.2.9.4 The Combiner

The combiner is a chain of operations used to combine the resulting image of the up sampling operations and the different image saved to retrieve an image at a higher resolution level. The requirement for this parameter is similar to the requirement for the differencer parameter.

#### 4.2.9.5 Example

Listing 4-4 shows a complete code example of the use of ImagePyramid.

**Listing 4-4 Example use of ImagePyramid (Sheet 1 of 4)**

```
   import java.awt.image.RenderedImage;
   import java.awt.image.renderable.ParameterBlock;
   import javax.media.jai.JAI;
   import javax.media.jai.Interpolation;
   import javax.media.jai.ImageMIPMap;
   import javax.media.jai.ImagePyramid;
   import javax.media.jai.PlanarImage;
   import javax.media.jai.RenderedOp;
   import com.sun.media.jai.codec.FileSeekableStream;

   public class ImagePyramidTest extends ImageMIPMapTest {
```

**Listing 4-4 Example use of ImagePyramid (Sheet 2 of 4)**

```
    protected RenderedOp upSampler;
    protected RenderedOp differencer;
    protected RenderedOp combiner;

    protected ImagePyramid pyramid;

    private void test1() {
    }

    public void test2() {
        downSampler = createScaleOp(image, 0.9F);
        downSampler.removeSources();
        downSampler = createScaleOp(downSampler, 0.9F);

        upSampler = createScaleOp(image, 1.2F);
        upSampler.removeSources();
        upSampler = createScaleOp(upSampler, 1.2F);

        differencer = createSubtractOp(image, image);
        differencer.removeSources();

        combiner = createAddOp(image, image);
        combiner.removeSources();

       pyramid = new ImagePyramid(image, downSampler, upSampler,
                                       differencer, combiner);
        display(pyramid.getImage(0));
        display(pyramid.getImage(4));
        display(pyramid.getImage(1));
        display(pyramid.getImage(6));
    }

    public void test3() {
        downSampler = createScaleOp(image, 0.9F);
        downSampler = createScaleOp(downSampler, 0.9F);

        upSampler = createScaleOp(image, 1.2F);
        upSampler.removeSources();

        differencer = createSubtractOp(image, image);
        differencer.removeSources();

        combiner = createAddOp(image, image);
        combiner.removeSources();
```

**Listing 4-4 Example use of ImagePyramid (Sheet 3 of 4)**

```
        pyramid = new ImagePyramid(downSampler, upSampler,
                                   differencer, combiner);
        // display(pyramid.getCurrentImage());
        display(pyramid.getDownImage());
        // display(pyramid.getDownImage());
        display(pyramid.getUpImage());
    }

    public void test4() {
        downSampler = createScaleOp(image, 0.5F);

        upSampler = createScaleOp(image, 2.0F);
        upSampler.removeSources();

        differencer = createSubtractOp(image, image);
        differencer.removeSources();

        combiner = createAddOp(image, image);
        combiner.removeSources();

        pyramid = new ImagePyramid(downSampler, upSampler,
                                   differencer, combiner);
        pyramid.getDownImage();

        display(pyramid.getCurrentImage());
        display(pyramid.getDiffImage());
        display(pyramid.getCurrentImage());
    }

    protected RenderedOp createSubtractOp(RenderedImage src1,
                                          RenderedImage src2) {
        ParameterBlock pb = new ParameterBlock();
        pb.addSource(src1);
        pb.addSource(src2);
        return JAI.create("subtract", pb);
    }

    protected RenderedOp createAddOp(RenderedImage src1,
                                     RenderedImage src2) {
        ParameterBlock pb = new ParameterBlock();
        pb.addSource(src1);
        pb.addSource(src2);
        return JAI.create("add", pb);
    }
```

**Listing 4-4 Example use of ImagePyramid (Sheet 4 of 4)**

```
    public ImagePyramidTest(String name) {
        super(name);
    }

    public static void main(String args[]) {
        ImagePyramidTest test = new
ImagePyramidTest("ImagePyramid");
        // test.test2();
        test.test3();
        // test.test4();
    }
}
```

**API:** `javax.media.jai.ImagePyramid`

- `ImagePyramid(RenderedImage image, RenderedOp downsampler,`
  `RenderedOp upSampler, RenderedOp differencer,`
  `RenderedOp combiner)`

  constructs an `ImagePyramid` object. The parameters point to the last operation in each chain. The first operation in each chain must not have any source images specified; that is, its number of sources must be 0.

| *Parameters*: | | |
|---|---|---|
| | `image` | The image with the highest resolution. |
| | `downsampler` | The operation chain used to derive the lower-resolution images. |
| | `upsampler` | The operation chain used to derive the higher-resolution images. |
| | `differencer` | The operation chain used to differ two images. |
| | `combiner` | The operation chain used to combine two images. |

- `ImagePyramid(RenderedOp downSampler, RenderedOp upSampler,`
  `RenderedOp differencer, RenderedOp combiner)`

  constructs an `ImagePyramid` object. The `RenderedOp` parameters point to the last operation node in each chain. The first operation in the `downSampler` chain must have the image with the highest resolution as its source. The first operation in all other chains must not have any source images specified; that is, its number of sources must be 0. All input parameters are saved by reference.

- `public RenderedImage getImage(int level)`

  returns the image at the specified resolution level. The requested level must be greater than or equal to 0 or null will be returned. The image is obtained by either down sampling or up sampling the current image.

- `public RenderedImage getDownImage()`

  returns the image at the next lower resolution level, obtained by applying the `downSampler` on the image at the current resolution level.

- `public RenderedImage getUpImage()`

  returns the image at the previous higher resolution level. If the current image is already at level 0, the current image is returned without further up sampling. The image is obtained by first up sampling the current image, then combining the result image with the previously saved different image using the `combiner` op chain.

- `public RenderedImage getDiffImage()`

  returns the difference image between the current image and the image obtained by first down sampling the current image then up sampling the result image of down sampling. This is done using the `differencer` op chain. The current level and current image will not be changed.

### 4.2.10 Multi-resolution Renderable Images

The `MultiResolutionRenderableImage` class produces renderings based on a set of supplied `RenderedImages` at various resolutions. The `MultiResolutionRenderableImage` is constructed from a specified dimension (height; the width is derived by the source image aspect ratio and is not specified) and a vector of renderedImages of progressively lower resolution.

**API:** `javax.media.jai.MultiResolutionRenderableImage`

- `public MultiResolutionRenderableImage(Vector renderedSources, float minX, float minY, float height)`

  constructs a `MultiResolutionRenderableImage` with given dimensions from a `Vector` of progressively lower resolution versions of a RenderedImage.

| | | |
|---|---|---|
| *Parameters*: | `rendered-Sources` | A `Vector` of `RenderedImages`. |
| | `minX` | The minimum *x* coordinate of the Renderable, as a float. |
| | `minY` | The minimum *y* coordinate of the Renderable, as a float. |
| | `height` | The height of the Renderable, as a float. |

- `RenderedImage createScaledRendering(int width, int height, RenderingHints hints)`

  returns a rendering with a given width, height, and rendering hints. If a JAI rendering hint named `JAI.KEY_INTERPOLATION` is provided, its corresponding `Interpolation` object is used as an argument to the JAI operator used to scale the image. If no such hint is present, an instance of `InterpolationNearest` is used.

| | | |
|---|---|---|
| *Parameters*: | `width` | The width of the rendering in pixels. |
| | `height` | The height of the rendering in pixels. |
| | `hints` | A `Hashtable` of rendering hints. |

- `RenderedImage createDefaultRendering()`

  returns a 100-pixel high rendering with no rendering hints.

- `RenderedImage createRendering(RenderContext renderContext)`

  returns a rendering based on a `RenderContext`. If a JAI rendering hint named `JAI.KEY_INTERPOLATION` is provided, its corresponding `Interpolation` object is used as an argument to the JAI operator used to scale the image. If no such hint is present, an instance of `InterpolationNearest` is used.

| | | |
|---|---|---|
| *Parameters*: | `render-Context` | A `RenderContext` describing the transform and rendering hints. |

- `Object getProperty(String name)`

  gets a property from the property set of this image. If the property name is not recognized, `java.awt.Image.UndefinedProperty` will be returned.

  | | | |
  |---|---|---|
  | *Parameters*: | `name` | The name of the property to get, as a String. |

- `String[] getPropertyNames()`

  returns a list of the properties recognized by this image.

- `float getWidth()`

  returns the floating-point width of the `RenderableImage`.

- `float getHeight()`

  returns the floating-point height of the `RenderableImage`.

- `float getMinX()`

  returns the floating-point minimum *x* coordinate of the `RenderableImage`.

- `float getMaxX()`

  returns the floating-point maximum *x* coordinate of the `RenderableImage`.

- `float getMinY()`

  returns the floating-point minimum *y* coordinate of the `RenderableImage`.

- `float getMaxY()`

  returns the floating-point maximum *y* coordinate of the `RenderableImage`.

## 4.3 Streams

The Java Advanced Imaging API extends the Java family of stream types with the addition of seven “seekable” stream classes, as shown in Figure 4-4. Table 4-3 briefly describes each of the new classes.

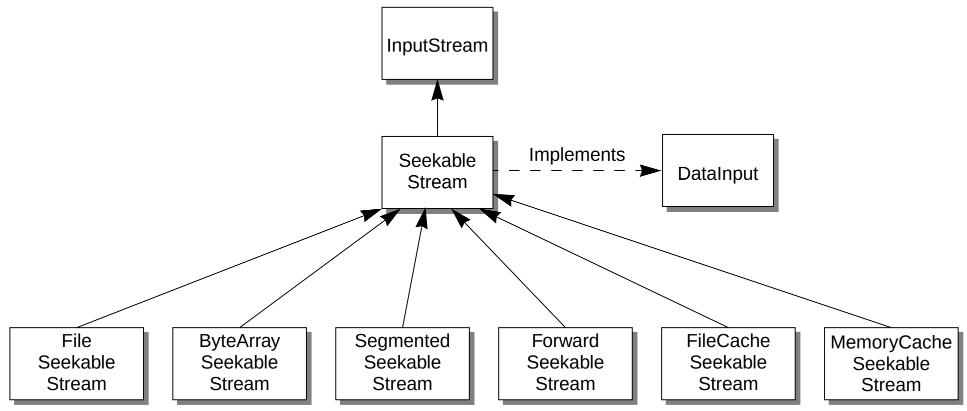

InputStream
Seekable Stream
Implements
DataInput
File Seekable Stream
ByteArray Seekable Stream
Segmented Seekable Stream
Forward Seekable Stream
FileCache Seekable Stream
MemoryCache Seekable Stream

**Figure 4-4 JAI Stream Classes**

The new seekable classes are used to cache the image data being read so that methods can be used to seek backwards and forwards through the data without having to re-read the data. This is especially important for image data types that are segmented or that cannot be easily re-read to locate important information.

**Table 4-3 JAI Stream Classes**

| Class | Description |
|---|---|
| `SeekableStream` | Extends: `InputStream`<br>Implements: `DataInput`<br>An abstract class that combines the functionality of `InputStream` and `RandomAccessFile`, along with the ability to read primitive data types in little-endian format. |
| `FileSeekableStream` | Extends: `SeekableStream`<br>Implements SeekableStream functionality on data stored in a File. |
| `ByteArraySeekableStream` | Extends: `SeekableStream`<br>Implements `SeekableStream` functionality on data stored in an array of bytes. |
| `SegmentedSeekableStream` | Extends: `SeekableStream`<br>Provides a view of a subset of another `SeekableStream` consisting of a series of segments with given starting positions in the source stream and lengths. The resulting stream behaves like an ordinary `SeekableStream`. |
| `ForwardSeekableStream` | Extends: `SeekableStream`<br>Provides `SeekableStream` functionality on data from an `InputStream` with minimal overhead, but does not allow seeking backwards. `ForwardSeekableStream` may be used with input formats that support streaming, avoiding the need to cache the input data. |

**Table 4-3 JAI Stream Classes (Continued)**

| Class | Description |
|---|---|
| `FileCacheSeekableStream` | Extends: `SeekableStream`<br>Provides `SeekableStream` functionality on data from an `InputStream` with minimal overhead, but does not allow seeking backwards. `ForwardSeekableStream` may be used with input formats that support streaming, avoiding the need to cache the input data. In circumstances that do not allow the creation of a temporary file (for example, due to security consideration or the absence of local disk), the `MemoryCacheSeekableStream` class may be used. |
| `MemoryCacheSeekableStream` | Extends: `SeekableStream`<br>Provides `SeekableStream` functionality on data from an `InputStream`, using an in-memory cache to allow seeking backwards. `MemoryCacheSeekableStream` should be used when security or lack of access to local disk precludes the use of `FileCacheSeekableStream`. |

To properly read some image data files requires the ability to seek forward and backward through the data so as to read information that describes the image. The best way of making the data seekable is through a *cache*, a temporary file stored on a local disk or in main memory. The preferred method of storage for the cached data is local disk, but that it not always possible. For security concerns or for diskless systems, the creation of a disk file cache may not always be permitted. When a file cache is not permissible, an in-memory cache may be used.

The `SeekableStream` class allows seeking within the input, similarly to the `RandomAccessFile` class. Additionally, the `DataInput` interface is supported and extended to include support for little-endian representations of fundamental data types.

The `SeekableStream` class adds several `read` methods to the already extensive java.io.`DataInput` class, including methods for reading data in little-endian (LE) order. In Java, all values are written in big-endian fashion. However, JAI needs methods for reading data that is not produced by Java; data that is produced on other platforms that produce data in the little-endian fashion. Table 4-4 is a complete list of the methods to read data:

**Table 4-4 Read Data Methods**

| Method | Description |
|---|---|
| `readInt` | Reads a signed 32-bit integer |
| `readIntLE` | Reads a signed 32-bit integer in little-endian order |
| `readShort` | Reads a signed 16-bit number |
| `readShortLE` | Reads a 16-bit number in little-endian order |

**Table 4-4 Read Data Methods (Continued)**

| Method | Description |
|---|---|
| `readLong` | Reads a signed 64-bit integer |
| `readLongLE` | Reads a signed 64-bit integer in little-endian order |
| `readFloat` | Reads a 32-bit float |
| `readFloatLE` | Reads a 32-bit float in little-endian order |
| `readDouble` | Reads a 64-bit double |
| `readDoubleLE` | Reads a 64-bit double in little-endian order |
| `readChar` | Reads a 16-bit Unicode character |
| `readCharLE` | Reads a 16-bit Unicode character in little-endian order |
| `readByte` | Reads an signed 8-bit byte |
| `readBoolean` | Reads a Boolean value |
| `readUTF` | Reads a string of characters in UTF (Unicode Text Format) |
| `readUnsignedShort` | Reads an unsigned 16-bit short integer |
| `readUnsignedShortLE` | Reads an unsigned 16-bit short integer in little-endian order |
| `readUnsignedInt` | Reads an unsigned 32-bit integer |
| `readUnsignedIntLE` | Reads an unsigned 32-bit integer in little-endian order |
| `readUnsignedByte` | Reads an unsigned 8-bit byte |
| `readLine` | Reads in a line that has been terminated by a line-termination character. |
| `readFully` | Reads a specified number of bytes, starting at the current stream pointer |
| `read()` | Reads the next byte of data from the input stream. |

In addition to the familiar methods from `InputStream`, the methods `getFilePointer()` and `seek()`, are defined as in the `RandomAccessFile` class. The `canSeekBackwards()` method returns `true` if it is permissible to seek to a position earlier in the stream than the current value of `getFilePointer()`. Some subclasses of `SeekableStream` guarantee the ability to seek backwards while others may not offer this feature in the interest of efficiency for those users who do not require backward seeking.

Several concrete subclasses of `SeekableStream` are supplied in the `com.sun.media.jai.codec` package. Three classes are provided for the purpose of adapting a standard `InputStream` to the `SeekableStream` interface. The `ForwardSeekableStream` class does not allow seeking backwards, but is inexpensive to use. The `FileCacheSeekableStream` class maintains a copy of all of the data read from the input in a temporary file; this file will be discarded automatically when the `FileSeekableStream` is finalized, or when the JVM exits normally.

The `FileCacheSeekableStream` class is intended to be reasonably efficient apart from the unavoidable use of disk space. In circumstances where the creation of a temporary file is not possible, the `MemoryCacheSeekableStream` class may be used. The `MemoryCacheSeekableStream` class creates a potentially large in-memory buffer to store the stream data and so should be avoided when possible. The `FileSeekableStream` class wraps a `File` or `RandomAccessFile`. It forwards requests to the real underlying file. `FileSeekableStream` performs a limited amount of caching to avoid excessive I/O costs.

A convenience method, `wrapInputStream` is provided to construct a suitable `SeekableStream` instance whose data is supplied by a given `InputStream`. The caller, by means of the `canSeekBackwards` parameter, determines whether support for seeking backwards is required.

## 4.4 Reading Image Files

The JAI codec architecture consists of encoders and decoders capable of writing and reading several different raster image file formats. This chapter describes reading image files. For information on writing image files, see **Chapter 13, "Writing Image Files**."

There are many raster image file formats, most of which have been created to support both image storage and interchange. Some formats have become widely used and are considered de facto standards. Other formats, although very important to individual software vendors, are less widely used.

JAI directly supports several of the most common image file formats, listed in Table 4-5. If your favorite file format is not listed in Table 4-5, you may either be able to create your own file codec (see **Chapter 14, "Extending the API**") or use one obtained from a third party developer.

**Table 4-5 Image File Formats**

| File Format Name | Description |
|---|---|
| BMP | Microsoft Windows bitmap image file |
| FPX | FlashPix format |
| GIF | Compuserve's Graphics Interchange Format |
| JPEG | A file format developed by the Joint Photographic Experts Group |

**Table 4-5 Image File Formats (Continued)**

| File Format Name | Description |
|---|---|
| PNG | Portable Network Graphics |
| PNM | Portable aNy Map file format. Includes PBM, PGM, and PPM. |
| TIFF | Tag Image File Format |

An image file usually has at least two parts: a file header and the image data. The header contains fields of pertinent information regarding the following image data. At the very least, the header must provide all the information necessary to reconstruct the original image from the stored image data. The image data itself may or may not be compressed.

The main class for image decoders and encoders is the `ImageCodec` class. Subclasses of `ImageCodec` are able to perform recognition of a particular file format either by inspection of a fixed-length file header or by arbitrary access to the source data stream. Each `ImageCodec` subclass implements one of two image file recognition methods. The codec first calls the `getNumHeaderBytes()` method, which either returns 0 if arbitrary access to the stream is required, or returns the number of header bytes required to recognize the format. Depending on the outcome of the `getNumHeaderBytes()` method, the codec either reads the stream or the header.

Once the codec has determined the image format, either by reading the stream or the header, it returns the name of the codec associated with the detected image format. If no codec is registered with the name, `null` is returned. The name of the codec defines the subclass that is called, which decodes the image.

For most image types, JAI offers the option of reading an image data file as a `java.io.File` object or as one of the subclasses of `java.io.InputStream`.

JAI offers several file operators for reading image data files, as listed in Table 4-6.

**Table 4-6 Image File Operators**

| Operator | Description |
|---|---|
| `AWTImage` | Imports a standard AWT image into JAI. |
| `BMP` | Reads BMP data from an input stream. |
| `FileLoad` | Reads an image from a file. |
| `FPX` | Reads FlashPix data from an input stream. |
| `FPXFile` | Reads a standard FlashPix file. |

**Table 4-6 Image File Operators (Continued)**

| Operator | Description |
|---|---|
| `GIF` | Reads GIF data from an input stream. |
| `JPEG` | Reads a standard JPEG (JFIF) file. |
| `PNG` | Reads a PNG input stream. |
| `PNM` | Reads a standard PNM file, including PBM, PGM, and PPM images of both ASCII and raw formats. |
| `Stream` | Reads `java.io.InputStream` files. |
| `TIFF` | Reads TIFF 6.0 data from an input stream. |
| `URL` | Creates an image the source of which is specified by a Uniform Resource Locator (URL). |

## 4.4.1 Standard File Readers for Most Data Types

You can read a file type directly with one of the available operation descriptors (such as the `tiff` operation to read TIFF files), by the stream file reader to read `InputStream` files, or the `FileLoad` operator to read from a disk file. The `Stream` and `FileLoad` operations are generic file readers in the sense that the image file type does not have to be known ahead of time, assuming that the file type is one of those recognized by JAI. These file read operations automatically detect the file type when invoked and use the appropriate file reader. This means that the programmer can use the same graph to read any of the “recognized” file types.

The `Stream` and `FileLoad` operations use a set of `FormatRecognizer` classes to query the file types when the image data is called for. A `FormatRecognizer` may be provided for any format that may be definitively recognized by examining the initial portion of the data stream. A new `FormatRecognizer` may be added to the `OperationRegistry` by means of the `registerFormatRecognizer` method (see Section 14.5, “Writing New Image Decoders and Encoders”).

### 4.4.1.1 The Stream Operation

The `Stream` operation reads an image from a `SeekableStream`. If the file is one of the recognized “types,” the file will be read. The `file` operation will query the set of registered `FormatRecognizers`. If a call to the `isFormatRecognized` method returns true, the associated operation name is retrieved by calling the `getOperationName` method and the named operation is instantiated.

If the operation fails to read the file, no other operation will be invoked since the input will have been consumed.

The `Stream` operation takes a single parameter:

| Parameter | Type | Description |
|---|---|---|
| `stream` | `SeekableStream` | The `SeekableStream` to read from. |

Listing 4-5 shows a code sample for a `Stream` operation.

**Listing 4-5 Example Stream Operation**

```
   // Load the source image from a Stream.
   RenderedImage im = JAI.create("stream", stream);
```

#### 4.4.1.2 The FileLoad Operation

The FileLoad operation reads an image from a file. Like the `Stream` operation, if the file is one of the recognized "types," the file will be read. If the operation fails to read the file, no other operation will be invoked since the input will have been consumed.

The `FileLoad` operation takes a single parameter:

| Parameter | Type | Description |
|---|---|---|
| `filename` | `String` | The path of the file to read from. |

Listing 4-6 shows a code sample for a `FileLoad` operation.

**Listing 4-6 Example FileLoad Operation**

```
   // Load the source image from a file.
   RenderedImage src = (RenderedImage)JAI.create("fileload",
                           fileName);
```

### 4.4.2 Reading TIFF Images

The Tag Image File Format (TIFF) is one of the most common digital image file formats. This file format was specifically designed for large arrays of raster image data originating from many sources, including scanners and video frame grabbers. TIFF was also designed to be portable across several different computer platforms, including UNIX, Windows, and Macintosh. The TIFF file format is highly flexible, which also makes it fairly complex.

The `TIFF` operation reads TIFF data from a TIFF `SeekableStream`. The `TIFF` operation takes one parameter:

| **Parameter** | **Type** | **Description** |
|---|---|---|
| `file` | `SeekableStream` | The `SeekableStream` to read from. |

The `TIFF` operation reads the following TIFF image types:

- Bilevel or grayscale, white is zero
- Bilevel or grayscale, black is zero
- Palette-color images
- RGB full color (three samples per pixel)
- RGB full color (four samples per pixel) (Opacity + RGB)
- RGB full color with alpha data
- RGB full color with alpha data (with pre-multiplied color)
- RGB full color with extra components
- Transparency mask

The `TIFF` operation supports the following compression types:

- None (no compression)
- PackBits compression
- Modified Huffman compression (CCITT Group3 1-dimensional facsimile)

For an example of reading a TIFF file, see Listing A-1 on page 417.

### 4.4.2.1 Palette Color Images

For TIFF Palette color images, the `colorMap` always has entries of short data type, the color black being represented by 0,0,0 and white by 65536,65536,65536. To display these images, the default behavior is to dither the short values down to 8 bits. The dithering is done by calling the `decode16BitsTo8Bit` method for each short value that needs to be dithered. The method has the following implementation:

```
byte b;
short s;
s = s & 0xffff;
b = (byte)((s >> 8) & 0xff);
```

If a different algorithm is to be used for the dithering, the `TIFFDecodeParam` class should be subclassed and an appropriate implementation should be provided for the `decode16BitsTo8Bits` method in the subclass.

If it is desired that the Palette be decoded such that the output image is of short data type and no dithering is performed, use the `setDecodePaletteAsShorts` method.

**API:** `com.sun.media.jai.codec.TIFFDecodeParam`

- `void setDecodePaletteAsShorts(boolean decodePaletteAsShorts)`

  if set, the entries in the palette will be decoded as shorts and no short-to-byte lookup will be applied to them.

- `boolean getDecodePaletteAsShorts()`

  returns `true` if palette entries will be decoded as shorts, resulting in a output image with short datatype.

- `byte decode16BitsTo8Bits(int s)`

  returns an unsigned 8-bit value computed by dithering the unsigned 16-bit value. Note that the TIFF specified short datatype is an unsigned value, while Java's `short` datatype is a signed value. Therefore the Java `short` datatype cannot be used to store the TIFF specified short value. A Java `int` is used as input instead to this method. The method deals correctly only with 16-bit unsigned values.

### 4.4.2.2 Multiple Images per TIFF File

A TIFF file may contain more than one Image File Directory (IFD). Each IFD defines a *subfile*, which may be used to describe related images. To determine the number of images in a TIFF file, use the `TIFFDirectory.getNumDirectories()` method.

**API:** `com.sun.media.jai.codec.TIFFDirectory`

- `static int getNumDirectories(SeekableStream stream)`

  returns the number of image directories (subimages) stored in a given TIFF file, represented by a `SeekableStream`.

### 4.4.2.3 Image File Directory (IFD)

The `TIFFDirectory` class represents an Image File Directory (IFD) from a TIFF 6.0 stream. The IFD consists of a count of the number of directories (number of fields), followed by a sequence of field entries identified by a tag that identifies the field. A field is identified as a sequence of values of identical data type. The TIFF 6.0 specification defines 12 data types, which are mapped internally into the Java data types, as described in Table 4-7.

**Table 4-7 TIFF Data Types**

| **TIFF Field Type** | **Java Data Type** | **Description** |
|---|---|---|
| `TIFF_BYTE` | `byte` | 8-bit unsigned integer |
| `TIFF_ASCII` | `String` | Null-terminated ASCII strings. |
| `TIFF_SHORT` | `char` | 16-bit unsigned integers. |
| `TIFF_LONG` | `long` | 32-bit unsigned integers. |
| `TIFF_RATIONAL` | `long[2]` | Pairs of 32-bit unsigned integers. |
| `TIFF_SBYTE` | `byte` | 8-bit signed integers. |
| `TIFF_UNDEFINED` | `byte` | 16-bit signed integers. |
| `TIFF_SSHORT` | `short` | 1-bit signed integers. |
| `TIFF_SLONG` | `int` | 32-bit signed integers. |
| `TIFF_SRATIONAL` | `int[2]` | Pairs of 32-bit signed integers. |
| `TIFF_FLOAT` | `float` | 32-bit IEEE floats. |
| `TIFF_DOUBLE` | `double` | 64-bit IEEE doubles. |

The `TIFFField` class contains several methods to query the set of tags and to obtain the raw field array. In addition, convenience methods are provided for acquiring the values of tags that contain a single value that fits into a `byte`, `int`, `long`, `float`, or `double`.

The `getTag` method returns the tag number. The tag number identifies the field. The tag number is an `int` value between 0 and 65,535. The `getType` method returns the type of data stored in the IFD. For a TIFF 6.0 file, the value will be one of those defined in Table 4-7. The `getCount` method returns the number of elements in the IFD. The count (also known as *length* in earlier TIFF specifications) is the number of values.

**API:** `com.sun.media.jai.codec.TIFFField`

- `int getTag()`

  returns the tag number, between 0 and 65535.

- `int getType()`

  returns the type of the data stored in the IFD.

- `int getCount()`

  returns the number of elements in the IFD.

#### 4.4.2.4 Public and Private IFDs

Every TIFF file is made up of one or more public IFDs that are joined in a linked list, rooted in the file header. A file may also contain so-called *private* IFDs that are referenced from tag data and do not appear in the main list.

The `TIFFDecodeParam` class allows the index of the TIFF directory (IFD) to be set. In a multipage TIFF file, index 0 corresponds to the first image, index 1 to the second, and so on. The index defaults to 0.

**API:** `com.sun.media.jai.codec.TIFFDirectory`

- `TIFFDirectory(SeekableStream stream, int directory)`

  constructs a `TIFFDirectory` from a `SeekableStream`. The directory parameter specifies which directory to read from the linked list present in the stream; directory 0 is normally read but it is possible to store multiple images in a single TIFF file by maintaining multiple directories.

  | *Parameters*: | | |
  |---|---|---|
  | | `stream` | A `SeekableStream`. |
  | | `directory` | The index of the directory to read. |

- `TIFFDirectory(SeekableStream stream, long ifd_offset)`

  constructs a TIFFDirectory by reading a `SeekableStream`. The `ifd_offset` parameter specifies the stream offset from which to begin reading; this mechanism is sometimes used to store private IFDs within a TIFF file that are not part of the normal sequence of IFDs.

- `int getNumEntries()`

  returns the number of directory entries.

- `TIFFField getField(int tag)`

  returns the value of a given tag as a TIFFField, or null if the tag is not present.

- `boolean isTagPresent(int tag)`

  returns true if a tag appears in the directory.

- `int[] getTags()`
  returns an ordered array of ints indicating the tag values.
- `TIFFField[] getFields()`
  returns an array of TIFFFields containing all the fields in this directory.
- `byte getFieldAsByte(int tag, int index)`
  returns the value of a particular index of a given tag as a byte. The caller is responsible for ensuring that the tag is present and has type `TIFFField.TIFF_SBYTE`, `TIFF_BYTE`, or `TIFF_UNDEFINED`.
- `byte getFieldAsByte(int tag)`
  returns the value of index 0 of a given tag as a byte.
- `long getFieldAsLong(int tag, int index)`
  returns the value of a particular index of a given tag as a long.
- `long getFieldAsLong(int tag)`
  returns the value of index 0 of a given tag as a long.
- `float getFieldAsFloat(int tag, int index)`
  returns the value of a particular index of a given tag as a float.
- `float getFieldAsFloat(int tag)`
  returns the value of a index 0 of a given tag as a float.
- `double getFieldAsDouble(int tag, int index)`
  returns the value of a particular index of a given tag as a double.
- `double getFieldAsDouble(int tag)`
  returns the value of index 0 of a given tag as a double.

### 4.4.3 Reading FlashPix Images

FlashPix is a multi-resolution, tiled file format that allows images to be stored at different resolutions for different purposes, such as editing or printing. Each resolution is divided into 64 × 64 blocks, or tiles. Within a tile, pixels can be either uncompressed, JPEG compressed, or single-color compressed.

The FPX operation reads an image from a FlashPix stream. The FPX operation takes one parameter:

| Parameter | Type | Description |
|---|---|---|
| stream | SeekableStream | The SeekableStream to read from. |

Listing 4-7 shows a code sample for a FPX operation.

**Listing 4-7 Example of Reading a FlashPix Image**

```
   // Specify the filename.
   File file = new File(filename);

   // Specify the resolution of the file.
   ImageDecodeParam param = new FPXDecodeParam(resolution);

   // Create the FPX operation to read the file.
   ImageDecoder decoder = ImageCodec.createImageDecoder("fpx",
                                                        file,
                                                        param);

   RenderedImage im = decoder.decodeAsRenderedImage();
   ScrollingImagePanel p =
       new ScrollingImagePanel(im,
                               Math.min(im.getWidth(), 800) + 20,
                              Math.min(im.getHeight(), 800) + 20);
```

## 4.4.4 Reading JPEG Images

The JPEG standard was developed by a working group, known as the Joint Photographic Experts Group (JPEG). The JPEG image data compression standard handles grayscale and color images of varying resolution and size.

The JPEG operation takes a single parameter:

| Parameter | Type | Description |
|---|---|---|
| file | SeekableStream | The SeekableStream to read from. |

## 4.4.5 Reading GIF Images

Compuserve's Graphics Interchange Format (GIF) is limited to 256 colors, but supported by virtually every platform that supports graphics.

The `GIF` operation reads an image from a GIF stream. The `GIF` operation takes a single parameter:

| Parameter | Type | Description |
|---|---|---|
| `stream` | `SeekableStream` | The `SeekableStream` to read from. |

### 4.4.6 Reading BMP Images

The BMP (Microsoft Windows bitmap image file) file format is a commonly-used file format on IBM PC-compatible computers. BMP files can also refer to the OS/2 bitmap format, which is a strict superset of the Windows format. The OS/2 2.0 format allows for multiple bitmaps in the same file, for the CCITT Group3 1bpp encoding, and also a RLE24 encoding.

The `BMP` operation reads BMP data from an input stream. The `BMP` operation currently reads Version2, Version3, and some of the Version 4 images, as defined in the Microsoft Windows BMP file format.

Version 4 of the BMP format allows for the specification of alpha values, gamma values, and CIE colorspaces. These are not currently handled, but the relevant properties are emitted, if they are available from the BMP image file.

The `BMP` operation takes a single parameter:

| Parameter | Type | Description |
|---|---|---|
| `stream` | `SeekableStream` | The `SeekableStream` to read from. |

Listing 4-8 shows a code sample for a `GIF` operation.

**Listing 4-8 Example of Reading a BMP Image**

```
   // Wrap the InputStream in a SeekableStream.
   InputStream is = new FileInputStream(filename);
   SeekableStream s = SeekableStream.wrapInputStream(is, false);

   // Create the ParameterBlock and add the SeekableStream to it.
   ParameterBlock pb = new ParameterBlock();
   pb.add(s);

   // Perform the BMP operation
   op = JAI.create("BMP", pb);
```

**API:** `com.sun.media.jai.codec.SeekableStream`

-
  ```
  static SeekableStream wrapInputStream(java.io.InputStream is,
      boolean canSeekBackwards)
  ```
  returns a SeekableStream that will read from a given InputStream, optionally including support for seeking backwards.

## 4.4.7 Reading PNG Images

The PNG (Portable Network Graphics) is an extensible file format for the lossless, portable, compressed storage of raster images. PNG was developed as a patent-free alternative to GIF and can also replace many common uses of TIFF. Indexed-color, grayscale, and truecolor images are supported, plus an optional alpha channel. Sample depths range from 1 to 16 bits.

For more information on PNG images, see the specification at the following URL:

```
http://www.cdrom.com/pub/png/spec
```

The `PNG` operation reads a standard PNG input stream. The `PNG` operation implements the entire PNG specification, but only provides access to the final, high-resolution version of interlaced images. The output image will always include a `ComponentSampleModel` and either a byte or short `DataBuffer`.

Pixels with a bit depth of less than eight are scaled up to fit into eight bits. One-bit pixel values are output as 0 and 255. Pixels with a bit depth of two or four are left shifted to fill eight bits. Palette color images are expanded into three-banded RGB. PNG images stored with a bit depth of 16 will be truncated to 8 bits of output unless the `KEY_PNG_EMIT_16BITS` hint is set to `Boolean.TRUE`. Similarly, the output image will not have an alpha channel unless the `KEY_PNG_EMIT_ALPHA` hint is set. See Section 4.4.7.3, "Rendering Hints."

The `PNG` operation takes a single parameter:

| Parameter | Type | Description |
|---|---|---|
| `stream` | `SeekableStream` | The `SeekableStream` to read from. |

Listing 4-9 shows a code sample for a PNG operation.

**Listing 4-9 Example of Reading a PNG Image**

```
// Create the ParameterBlock.
InputStream image = new FileInputStream(filename);
ParameterBlock pb = new ParameterBlock();
pb.add(image);

// Create the PNG operation.
op = JAI.create("PNG", pb);
```

Several aspects of the PNG image decoding may be controlled. By default, decoding produces output images with the following properties:

- Images with a bit depth of eight or less use a `DataBufferByte` to hold the pixel data. 16-bit images use a `DataBufferUShort`.
- Palette color images and non-transparent grayscale images with bit depths of one, two, or four will have a `MultiPixelPackedSampleModel` and an `IndexColorModel`. For palette color images, the `ColorModel` palette contains the red, green, blue, and optionally alpha palette information. For grayscale images, the palette is used to expand the pixel data to cover the range 0 to 255. The pixels are stored packed eight, four, or two to the byte.
- All other images are stored using a `PixelInterleavedSampleModel` with each band of a pixel occupying its own `byte` or `short` within the `DataBuffer`. A `ComponentColorModel` is used, which simply extracts the red, green, blue, gray, and/or alpha information from separate `DataBuffer` entries.

Methods in the `PNGDecodeParam` class permit changes to five aspects of the decode process:

- The `setSuppressAlpha()` method prevents an alpha channel from appearing in the output.
- The `setExpandPalette()` method turns palette-color images into three- or four-banded full-color images.
- The `setOutput8BitGray()` method causes one-, two-, or four-bit grayscale images to be output in eight-bit form, using a `ComponentSampleModel` and `ComponentColorModel`.
- The `setOuputGamma()` method causes the output image to be gamma-corrected using a supplied output gamma value.

- The `setExpandGrayAlpha()` method causes two-banded gray/alpha (GA) images to be output as full-color (GGGA) images, which may simplify further processing and display.

**API:** `com.sun.media.jai.codec.PNGDecodeParam`

- `public void setSuppressAlpha(boolean suppressAlpha)`

  when set, suppresses the alpha (transparency) channel in the output image.

- `public void setExpandPalette(boolean expandPalette)`

  when set, causes palette color images (PNG color type 3) to be decoded into full-color (RGB) output images. The output image may have three or four bands, depending on the presence of transparency information. The default is to output palette images using a single band. The palette information is used to construct the output image's `ColorModel`.

- `public void setOutput8BitGray(boolean output8BitGray)`

  when set, causes grayscale images with a bit depth of less than eight (one, two, or four) to be output in eight-bit form. The output values will occupy the full eight-bit range. For example, gray values zero, one, two, and three of a two-bit image will be output as 0, 85, 170, and 255. The decoding of non-grayscale images and grayscale images with a bit depth of 8 or 16 are unaffected by this setting. The default is not to perform expansion. Grayscale images with a depth of one, two, or four bits will be represented using a `MultiPixelPackedSampleModel` and an `IndexColorModel`.

- `public void setOutputGamma(float outputGamma)`

  sets the desired output gamma to a given value. In terms of the definitions in the PNG specification, the output gamma is equal to the viewing gamma divided by the display gamma. The output gamma must be positive. If the output gamma is set, the output image will be gamma-corrected using an overall exponent of output gamma/file gamma. Input files that do not contain gamma information are assumed to have a file gamma of 1.0. This parameter affects the decoding of all image types.

- `public void setExpandGrayAlpha(boolean expandGrayAlpha)`

  when set, causes images containing one band of gray and one band of alpha (GA) to be output in a four-banded format (GGGA). This produces output that may be simpler to process and display. This setting affects both images of color type 4 (explicit alpha) and images of color type 0 (grayscale) that contain transparency information.

### 4.4.7.1 Gamma Correction and Exponents

PNG images can contain a gamma correction value. The gamma value specifies the relationship between the image samples and the desired display output intensity as a power function:

$$\text{sample} = \text{light\_out}^{\text{gamma}}$$

The `getPerformGammaCorrection` method returns `true` if gamma correction is to be performed on the image data. By default, gamma correction is `true`.

If gamma correction is to be performed, the `getUserExponent` and `getDisplayExponent` methods are used in addition to the gamma value stored within the file (or the default value of 1/2.2 is used if no value is found) to produce a single exponent using the following equation:

$$\text{decoding\_exponent} = \frac{\text{user\_exponent}}{\text{gamma\_from\_file} \times \text{display\_exponent}}$$

The `setUserExponent` method is used to set the `user_exponent` value. If the `user_exponent` value is set, the output image pixels are placed through the following transformation:

$$\text{sample} = \frac{\text{integer\_sample}}{(2^{\text{bitdepth}} - 1.0)}$$

$$\text{decoding\_exponent} = \frac{\text{user\_exponent}}{\text{gamma\_from\_file} \times \text{display\_exponent}}$$

$$\text{output} = \text{sample}^{\text{decoding\_exponent}}$$

where `gamma_from_file` is the gamma of the file data, as determined by the gAMA, sRGB, and iCCP chunks. `display_exponent` is the exponent of the intrinsic transfer curve of the display, generally 2.2.

Input files that do not specify any gamma value are assumed to have a gamma of 1/2.2. Such images may be displayed on a CRT with an exponent of 2.2 using the default user exponent of 1.0.

The user exponent may be used to change the effective gamma of a file. If a file has a stored gamma of *X*, but the decoder believes that the true file gamma is *Y*, setting a user exponent of *Y*/*X* will produce the same result as changing the file gamma.

**API:** `com.sun.media.jai.codec.PNGDecodeParam`

- `boolean getPerformGammaCorrection()`

  returns `true` if gamma correction is to be performed on the image data. The default is `true`.

- `void setPerformGammaCorrection(boolean performGammaCorrection)`

  turns gamma correction of the image data on or off.

- `float getUserExponent()`

  returns the current value of the user exponent parameter. By default, the user exponent is equal to 1.0F.

- `void setUserExponent(float userExponent)`

  sets the user exponent to a given value. The exponent must be positive.

- `float getDisplayExponent()`

  returns the current value of the display exponent parameter. By default, the display exponent is 2.2F.

- `void setDisplayExponent(float displayExponent)`

  Sets the display exponent to a given value. The exponent must be positive.

#### 4.4.7.2 Expanding Grayscale Images to GGGA Format

Normally, the `PNG` operation does not expand images that contain one channel of gray and one channel of alpha into a four-channel (GGGA) format. If this type of expansion is desired, use the `setExpandGrayAlpha` method. This setting affects both images of color type 4 (explicit alpha) and images of color type 0 (grayscale) that contain transparency information.

**API:** `com.sun.media.jai.codec.PNGDecodeParam`

- `void setExpandGrayAlpha(boolean expandGrayAlpha)`

  sets or unsets the expansion of two-channel (gray and alpha) PNG images to four-channel (GGGA) images.

#### 4.4.7.3 Rendering Hints

The PNG rendering hints are:

| Hints | Description |
|---|---|
| `KEY_PNG_EMIT_ALPHA` | The alpha channel is set. The alpha channel, representing transparency information on a per-pixel basis, can be included in grayscale and truecolor PNG images. |
| `KEY_PNG_EMIT_16BITS` | Defines a bit depth of 16 bits. |

To read the hints, use the `OperationDescriptorImpl.getHint` method.

**API:** `javax.media.jai.OperationDescriptorImpl`

-
  ```
  Object getHint(RenderingHints.Key key,
      RenderingHints renderHints)
  ```
  queries the rendering hints for a particular hint key and copies it into the hints observed Hashtable if found. If the hint is not found, null is returned and the hints observed are left unchanged.

### 4.4.8 Reading PNM Images

The `PNM` operation reads a standard PNM file, including PBM, PGM, and PPM images of both ASCII and raw formats. The PBM (portable bitmap) format is a monochrome file format (single-banded), originally designed as a simple file format to make it easy to mail bitmaps between different types of machines. The PGM (portable graymap) format is a grayscale file format (single-banded). The PPM (portable pixmap) format is a color image file format (three-banded).

PNM image files are identified by a *magic number* in the file header that identifies the file type variant, as follows:

| Magic Number | File Type | SampleModel Type |
|---|---|---|
| P1 | PBM ASCII | `MultiPixelPackedSampleModel` |
| P2 | PGM ASCII | `PixelInterleavedSampleModel` |
| P3 | PPM ASCII | `PixelInterleavedSampleModel` |
| P4 | PBM raw | `MultiPixelPackedSampleModel` |
| P5 | PGM raw | `PixelInterleavedSampleModel` |
| P6 | PPM raw | `PixelInterleavedSampleModel` |

The PNM operation reads the file header to determine the file type, then stores the image data into an appropriate SampleModel. The PNM operation takes a single parameter:

| Parameter | Type | Description |
|---|---|---|
| stream | SeekableStream | The SeekableStream to read from. |

Listing 4-10 shows a code sample for a PNM operation.

**Listing 4-10 Example of Reading a PNM Image**

```
   // Create the ParameterBlock.
   InputStream image = new FileInputStream(filename);
   ParameterBlock pb = new ParameterBlock();
   pb.add(image);

   // Create the PNM operation.
   op = JAI.create("PNM", pb);
```

### 4.4.9 Reading Standard AWT Images

The AWTImage operation allows standard Java AWT images to be directly imported into JAI, as a rendered image. By default, the width and height of the image are the same as the original AWT image. The sample model and color model are set according to the AWT image data. The layout of the PlanarImage may be specified using the ImageLayout parameter at constructing time.

The AWTImage operation takes one parameter.

| Parameter | Type | Description |
|---|---|---|
| awtImage | Image | The standard Java AWT image to be converted. |

Listing 4-11 shows a code sample for an AWTImage operation.

**Listing 4-11 Example of Reading an AWT Image**

```
   // Create the ParameterBlock.
   ParameterBlock pb = new ParameterBlock();
   pb.add(image);

   // Create the AWTImage operation.
   PlanarImage im = (PlanarImage)JAI.create("awtImage", pb);
```

**API:** `javax.media.jai.PlanarImage`

- `void setImageParameters(ImageLayout layout, RenderedImage im)`
  sets the image bounds, tile grid layout, `SampleModel`, and `ColorModel` to match those of another image.

| *Parameters*: | | |
|---|---|---|
| | `layout` | An `ImageLayout` used to selectively override the image's layout, `SampleModel`, and `ColorModel`. If null, all parameters will be taken from the second argument. |
| | `im` | A `RenderedImage` used as the basis for the layout. |

### 4.4.10 Reading URL Images

The `URL` operation creates an image whose source is specified by a Uniform Resource Locator (URL). The `URL` operation takes one parameter.

| **Parameter** | **Type** | **Description** |
|---|---|---|
| `URL` | `java.net.URL.class` | The path of the file to read from. |

Listing 4-12 shows a code sample for a `URL` operation.

**Listing 4-12 Example of Reading a URL Image**

```
// Define the URL to the image.
url = new URL(“http://webstuff/images/duke.gif”);

// Read the image from the designated URL.
RenderedOp src = JAI.create("url", url);
```

## 4.5 Reformatting an Image

The `Format` operation reformats an image by casting the pixel values of an image to a given data type, replacing the `SampleModel` and `ColorModel` of an image, and restructuring the image's tile grid layout.

The pixel values of the destination image are defined by the following pseudocode:

```
dst[x][y][b] = cast(src[x][y][b], dataType)
```

where `dataType` is one of the constants `DataBuffer.TYPE_BYTE`, `DataBuffer.TYPE_SHORT`, `DataBuffer.TYPE_USHORT`, `DataBuffer.TYPE_INT`, `DataBuffer.TYPE_FLOAT`, or `TDataBuffer.YPE_DOUBLE`.

The output `SampleModel`, `ColorModel`, and tile grid layout are specified by passing an `ImageLayout` object as a `RenderingHint` named `ImageLayout`. The output image will have a `SampleModel` compatible with the one specified in the layout hint wherever possible; however, for output data types of float and double a `ComponentSampleModel` will be used regardless of the value of the `hint` parameter.

The `ImageLayout` may also specify a tile grid origin and size which will be respected.

The typecasting performed by the `Format` operation is defined by the set of expressions listed in Table 4-8, depending on the data types of the source and destination. Casting an image to its current data type is a no-op. See *The Java Language Specification* for the definition of type conversions between primitive types.

In most cases, it is not necessary to explicitly perform widening typecasts since they will be performed automatically by image operators when handed source images having different datatypes.

**Table 4-8 Format Actions**

| Source Type | Destination Type | Action |
|---|---|---|
| BYTE | SHORT | (short)(x & 0xff) |
| | USHORT | (short)(x & 0xff) |
| | INT | (int)(x & 0xff) |
| | FLOAT | (float)(x & 0xff) |
| | DOUBLE | (double)(x & 0xff) |
| SHORT | BYTE | (byte)clamp((int)x, 0, 255) |
| | USHORT | (short)clamp((int)x, 0, 32767) |
| | INT | (int)x |
| | FLOAT | (float)x |
| | DOUBLE | (double)x |

**Table 4-8 Format Actions (Continued)**

| Source Type | Destination Type | Action |
|---|---|---|
| USHORT | BYTE | (byte)clamp((int)x & 0xffff, 0, 255) |
| | SHORT | (short)clamp((int)x & 0xffff, 0, 32767) |
| | INT | (int)(x & 0xffff) |
| | FLOAT | (float)(x & 0xffff) |
| | DOUBLE | (double)(x & 0xffff) |
| INT | BYTE | (byte)clamp(x, 0, 255) |
| | SHORT | (short)x |
| | USHORT | (short)clamp(x, 0, 65535) |
| | FLOAT | (float)x |
| | DOUBLE | (double)x |
| FLOAT | BYTE | (byte)clamp((int)x, 0, 255) |
| | SHORT | (short)x |
| | USHORT | (short)clamp((int)x, 0, 65535) |
| | INT | (int)x |
| | DOUBLE | (double)x |
| DOUBLE | BYTE | (byte)clamp((int)x, 0, 255) |
| | SHORT | (short)x |
| | USHORT | (short)clamp((int)x, 0, 65535) |
| | INT | (int)x |
| | FLOAT | (float)x |

The `clamp` function may be defined as:

```
int clamp(int x, int low, int high) {
    return (x < low) ? low : ((x > high) ? high : x);
}
```

The `Format` operation takes a single parameter:

| Parameter | Type | Description |
|---|---|---|
| `dataType` | `Integer` | The output data type (from `java.awt.image.DataBuffer`). One of `TYPE_BYTE`, `TYPE_SHORT`, `TYPE_USHORT`, `TYPE_INT`, `TYPE_FLOAT`, or `TYPE_DOUBLE`. |

## 4.6 Converting a Rendered Image to Renderable

To use a Renderable DAG with a non-renderable image type, the image must first be converted from a Rendered type to a Renderable type. For example, to use an image obtained from a remote server in a Renderable chain, you would want to treat the source image as a `RenderedImage`, then convert it to a `RenderableImage` for further processing.

The `Renderable` operation produces a `RenderableImage` from a `RenderedImage` source. The `RenderableImage` thus produced consists of a "pyramid" of `RenderedImages` at progressively lower resolutions. The lower resolution images are produced by invoking the chain of operations specified via the `downSampler` parameter on the image at the next higher resolution level of the pyramid. The `downSampler` operation chain must adhere to the specifications described for the constructors of the `ImageMIPMap` class, which accept this type of parameter (see Section 4.2.9.1, "The Down Sampler").

The `downSampler` operation chain must reduce the image width and height at each level of the pyramid. The default operation chain for `downSampler` is a low pass filter implemented using a $5 \times 5$ separable Gaussian kernel derived from the one-dimensional kernel:

```
[0.05 0.25 0.40 0.25 0.05]
```

followed by subsampling by 2. This filter is known as a Laplacian pyramid[1] and makes a perfectly good `downSampler` for most applications. If this downSampler doesn't work for your specific application, you can create your own and call it with the `downSampler` parameter.

The number of levels in the pyramid will be such that the larger dimension (width or height) of the lowest-resolution pyramid level is less than or equal to the value of the `maxLowResDim` parameter, which must be positive. The default value for the `maxLowResDim` parameter is 64, meaning that the lowest-resolution pyramid level will be an image whose largest dimension is 64 pixels or less.

The minimum *x* and *y* coordinates and height in rendering-independent coordinates are supplied by the parameters `minX`, `minY`, and `height`, respectively. The value of the `height` parameter must be positive. It is not necessary to supply a value for the rendering-independent width as this is derived by multiplying the supplied height by the aspect ratio (width divided by height) of the source `RenderedImage`.

1. Burt, P.J. and Adelson, E.H., "The Laplacian pyramid as a compact image code," *IEEE Transactions on Communications*, pp. 532–540, 1983.

The `Renderable` operation takes five parameters, as follows:

| Parameter | Type | Description |
|---|---|---|
| `downSamples` | `RenderedOp` | The operation chain used to derive the lower resolution images. |
| `maxLowResDim` | `Integer` | The maximum dimension of the lowest resolution pyramid level. |
| `minX` | `Float` | The minimum rendering-independent *x* coordinate of the destination. |
| `minY` | `Float` | The minimum rendering-independent *y* coordinate of the destination. |
| `height` | `Float` | The rendering-independent height. |

The default values for these parameters are:

- `downSampler` – a low-pass filter (see Section 4.2.9.1, “The Down Sampler”)
- `maxLowResDim` – 64
- `minX` – 0.0F
- `minY` – 0.0F
- `height` – 1.0F

Listing 4-13 shows a code sample for a `Renderable` operation. The default parameters are used for all five parameters. The output of the `Renderable` operation (`ren`) can be passed to the next renderable operation in the graph.

**Listing 4-13 Example of Converting a Rendered Image to Renderable**

```
// Derive the RenderableImage from the source RenderedImage.
ParameterBlock pb = new ParameterBlock();
pb.addSource(src);
pb.add(null).add(null).add(null).add(null).add(null);

// Create the Renderable operation.
RenderableImage ren = JAI.createRenderable("renderable", pb);
```

## 4.7 Creating a Constant Image

The `constant` operation defines a multi-banded, tiled rendered image where all the pixels from the same band have a constant value. The width and height of the destination image must be specified and greater than 0.

The constant operation takes three parameters, as follows:

| Parameter | Type | Description |
|---|---|---|
| width | Float | The width of the image in pixels. |
| height | Float | The height of the image in pixels. |
| bandValues | Number | The constant pixel band values. |

At least one constant must be supplied. The number of bands of the image is determined by the number of constant pixel values supplied in the bandValues parameter. The data type is determined by the type of the constant from the first entry.

Listing 4-14 shows a code sample for a Constant operation.

**Listing 4-14 Example Constant Operation**

```
   // Create the ParameterBlock.
   Byte[] bandValues = new Byte[1];
   bandValues[0] = alpha1;
   pb = new ParameterBlock();
   pb.add(new Float(src1.getWidth()));   // The width
   pb.add(new Float(src1.getHeight()));  // The height
   pb.add(bandValues);                   // The band values

   // Create the constant operation.
   PlanarImage afa1 = (PlanarImage)JAI.create("constant", pb);
```

## 4.8 Image Display

JAI uses the Java 2D BufferedImage model for displaying images. The BufferedImage manages an image in memory and provides ways to store pixel data, interpret pixel data, and to render the pixel data to a Graphics2D context.

The display of images in JAI may be accomplished in several ways. First, the drawRenderedImage() call on Graphics2D may be used to produce an immediate rendering. Another method is to instantiate a display widget that responds to user requests such as scrolling and panning, as well as expose events, and requests image data from a RenderedImage source. This technique allows image data to be computed on demand.

It is for this purpose that JAI provides a widget, available in the javax.media.jai.widget package, called a ScrollingImagePanel. The ScrollingImagePanel takes a RenderedImage and a specified width and height and creates a panel with scrolling bars on the right and bottom. The image is placed in the center of the panel.

The scrolling image panel constructor takes three parameters. The first parameter is the image itself, which is usually the output of some previous operation in the rendering chain. The next two parameters are the image width and height, which can be retrieved with the `getWidth` and `getHeight` methods of the node in which the image was constructed (such as `RenderedOp`).

The width and height parameters do not have to be the same as the image's width and height. The parameters can be either larger or smaller than the image.

Once the `ScrollingImagePanel` is created, it can be placed anywhere in a `Frame`, just like any other AWT panel. Listing 4-15 shows a code sample demonstrating the use of a scrolling image panel.

**Listing 4-15 Example Scrolling Image Panel**

```
   // Get the image width and height.
   int width = image.getWidth();
   int height = image.getHeight();

   // Attach the image to a scrolling panel to be displayed.
   ScrollingImagePanel panel = new ScrollingImagePanel(
                                   image, width, height);

   // Create a Frame to contain the panel.
   Frame window = new Frame(“Scrolling Image Panel Example”);
   window.add(panel);
   window.pack();
   window.show();
```

For a little more interesting example, consider the display of four images in a grid layout. The code sample in Listing 4-16 arranges four images into a 2 × 2 grid. This example uses the `java.awt.Panel` and the `java.awt.GridLayout` objects. These objects are not described in this document. See the Java Platform documentation for more information.

**Listing 4-16 Example Grid Layout of Four Images**

```
   // Display the four images in row order in a 2 x 2 grid.
   setLayout(new GridLayout(2, 2));
```

**Listing 4-16 Example Grid Layout of Four Images (Continued)**

```
   // Add the components, starting with the first entry in the
   // first row, the second, etc.
   add(new ScrollingImagePanel(im1, width, height));
   add(new ScrollingImagePanel(im2, width, height));
   add(new ScrollingImagePanel(im3, width, height));
   add(new ScrollingImagePanel(im4, width, height));

   pack();
   show();
```

The constructor for the `GridLayout` object specifies the number of rows and columns in the display (2 × 2 in this example). The four images (`im1`, `im2`, `im3`, and `im4`) are then added to the panel in separate `ScrollingImagePanels`. The resulting image is arranged as shown in Figure 4-5.

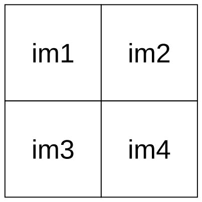


**Figure 4-5 Grid Layout of Four Images**

**API:** `javax.media.jai.RenderedOp`

- `int getWidth()`
  returns the width of the rendered image.

- `int getHeight()`
  returns the height of the rendered image.

**API:** `javax.media.jai.widget.ScrollingImagePanel`

- `ScrollingImagePanel(RenderedImage im, int width, int height)`
  constructs a `ScrollingImagePanel` of a given size for a given `RenderedImage`.

| | | |
|---|---|---|
| *Parameters*: | `im` | The `RenderedImage` displayed by the `ImageCanvas`. |
| | `width` | The panel width. |
| | `height` | The panel height. |

## 4.8.1 Positioning the Image in the Panel

You can define the position of the image within the `ScrollingImagePanel` by specifying either the position of the image origin or the image center location. The `setOrigin` method sets the origin of the image to a given (*x*, *y*) position within the `ScrollingImagePanel`. The `setCenter` method sets the image center to a given (*x*, *y*) position within the `ScrollingImagePanel`.

**API:** `javax.media.jai.widget.ScrollingImagePanel`

- `void setOrigin(int x, int y)`
  sets the image origin to a given (x, y) position. The scrollbars are updated appropriately.

| | | |
|---|---|---|
| *Parameters*: | `x` | The image *x* origin. |
| | `y` | The image *y* origin. |

- `void setCenter(int x, int y)`
  sets the image center to a given (x, y) position. The scrollbars are updated appropriately.

| | | |
|---|---|---|
| *Parameters*: | `x` | The image *x* center. |
| | `y` | The image *y* center. |

## 4.8.2 The ImageCanvas Class

A canvas in Java is a rectangular area in which you draw. JAI extends the `java.awt.Canvas` class with the `ImageCanvas` class, which allows you to "draw" an image in the canvas. Like `Canvas`, the `ImageCanvas` class inherits most of its methods from `java.awt.Component`, allowing you to use the same event handlers for keyboard and mouse input.

The `ImageCanvas` class is a simple output widget for a `RenderedImage` and can be used in any context that calls for a `Canvas`. The `ImageCanvas` class monitors resize and update events and automatically requests tiles from its source on demand. Any displayed area outside the image is displayed in gray.

Use the constructor or the `set` method to include a `RenderedImage` in the canvas, then use the `setOrigin` method to set the position of the image within the canvas.

**API:** `javax.media.jai.widget.ImageCanvas`

- `ImageCanvas(RenderedImage im, boolean drawBorder)`
  constructs an `ImageCanvas` to display a `RenderedImage`.

| *Parameters*: | | |
|---|---|---|
| | `im` | A `RenderedImage` to be displayed. |
| | `drawBorder` | True if a raised border is desired. |

- `ImageCanvas(java.awt.image.RenderedImage im)`
  constructs an `ImageCanvas` to display a `RenderedImage`.

| *Parameters*: | | |
|---|---|---|
| | `im` | A `RenderedImage` to be displayed. |

- `void set(RenderedImage im)`
  changes the source image to a new RenderedImage.

| *Parameters*: | | |
|---|---|---|
| | `im` | The new `RenderedImage` to be displayed. |

- `void paint(java.awt.Graphics g)`
  paint the image onto a `Graphics` object. The painting is performed tile-by-tile, and includes a gray region covering the unused portion of image tiles as well as the general background.

## 4.8.3 Image Origin

The origin of an image is set with the `ImageCanvas.setOrigin` method. The origin of an image is obtained with the `getXOrigin` and `getYOrigin` methods.

Geometric operators are treated differently with respect to image origin control. See **Chapter 8, “Geometric Image Manipulation**.”

**API:** `javax.media.jai.widget.ImageCanvas`

- `void setOrigin(int x, int y)`
  sets the origin of the image at `x,y`.
- `int getXOrigin()`
  returns the *x* coordinate of the image origin.
- `int getYOrigin()`
  returns the *y* coordinate of the image origin.

CHAPTER 5

# Color Space

THIS chapter describes the JAI color space, transparency, and the color conversion operators. JAI follows the Java AWT color model.

## 5.1 Introduction

Digital images, specifically digital color images, come in several different forms. The form is often dictated by the means by which the image was acquired or by the image's intended use.

One of the more basic types of color image is RGB, for the three primary colors (red, green, and blue). RGB images are sometimes acquired by a color scanner or video camera. These devices incorporate three sensors that are spectrally sensitive to light in the red, green, and blue portions of the spectrum. The three separate red, green, and blue values can be made to directly drive red, green, and blue light guns in a CRT. This type of color system is called an *additive* linear RGB color system, as the sum of the three full color values produces white.

Printed color images are based on a *subtractive* color process in which cyan, magenta, and yellow (CMY) dyes are deposited onto paper. The amount of dye deposited is subtractively proportional to the amount of each red, blue, and green color value. The sum of the three CMY color values produce black.

The black produced by a CMY color system often falls short of being a true black. To produce a more accurate black in printed images, black is often added as a fourth color component. This is known as the CMYK color system and is commonly used in the printing industry.

The amount of light generated by the red, blue, and green phosphors of a CRT is not linear. To achieve good display quality, the red, blue, and green values must be adjusted – a process known as *gamma correction*. In computer systems, gamma correction often takes place in the frame buffer, where the RGB values are passed through lookup tables that are set with the necessary compensation values.

In television transmission systems, the red, blue, and green gamma-corrected color video signals are not transmitted directly. Instead, a linear transformation between the RGB components is performed to produce a *luminance* signal and a pair of *chrominance* signals. The luminance signal conveys color brightness levels. The two chrominance signals convey the color hue and saturation. This color system is called YCC (or, more specifically, $YC_bC_r$).

Another significant color space standard for JAI is CIEXYZ. This is a widely-used, device-independent color standard developed by the Commission Internationale de l'Éclairage (CIE). The CIEXYZ standard is based on color-matching experiments on human observers.

## 5.2 Color Management

JAI uses three primary classes for the management of color:

- `ColorModel` – describes a particular way that pixel values are mapped to colors. A `ColorModel` is typically associated with an `Image` or `BufferedImage` and provides the information necessary to correctly interpret pixel values. `ColorModel` is defined in the `java.awt.image` package.
- `ColorSpace` – represents a system for measuring colors, typically using three separate values or components. The `ColorSpace` class contains methods for converting between the original color space and one of two standard color spaces, CIEXYZ and RGB. `ColorSpace` is defined in the `java.awt.color` package.
- `Color` – a fixed color, defined in terms of its components in a particular `ColorSpace`. `Color` is defined in the `java.awt` package.

### 5.2.1 Color Models

A `ColorModel` is used to interpret pixel data in an image. This includes:

- Mapping components in the bands of an image to components of a particular color space
- Extracting pixel components from packed pixel data
- Retrieving multiple components from a single band using masks

- Converting pixel data through a lookup table

To determine the color value of a particular pixel in an image, you need to know how the color information is encoded in each pixel. The ColorModel associated with an image encapsulates the data and methods necessary for translating a pixel value to and from its constituent color components.

JAI supports five color models:

- DirectColorModel – works with pixel values that represent RGB color and alpha information as separate samples and that pack all samples for a single pixel into a single int, short, or byte quantity. This class can be used only with ColorSpaces of type ColorSpace.TYPE_RGB.
- IndexColorModel – works with pixel values consisting of a single sample that is an index into a fixed colormap in the default sRGB ColorSpace. The colormap specifies red, green, blue, and optional alpha components corresponding to each index.
- ComponentColorModel – can handle an arbitrary ColorSpace and an array of color components to match the ColorSpace. This model can be used to represent most color models on most types of GraphicsDevices.
- PackedColorModel – a base class for models that represent pixel values in which the color components are embedded directly in the bits of an integer pixel. A PackedColorModel stores the packing information that describes how color and alpha components are extracted from the channel. The DirectColorModel is a PackedColorModel.
- FloatDoubleColorModel – works with pixel values that represent color and alpha information as separate samples, using float or double elements.

The following sample code shows the construction of a ComponentColorModel for an RGB color model.

```
// Create an RGB color model
int[] bits = { 8, 8, 8 };
ColorModel colorModel = new
 ComponentColorModel(ColorSpace.getInstance(ColorSpace.CS_sRGB),
                     bits, false, false,
                     Transparency.OPAQUE,
                     DataBuffer.TYPE_BYTE);
```

The following sample code shows the construction of a `ComponentColorModel` for a grayscale color model.

```
// Create a grayscale color model.
ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);
int bits[] = new int[] {8};
ColorModel cm = new ComponentColorModel(cs, bits, false, false,
                                        Transparency.OPAQUE,
                                        DataBuffer.TYPE_BYTE);
```

The following sample code shows the construction of a `FloatDoubleColorModel` for a linear RGB color model.

```
ColorSpace colorSpace =
    ColorSpace.getInstance(ColorSpace.CS_LINEAR_RGB);
int[] bits = new int[3];
bits[0] = bits[1] = bits[2] = 32;
ColorModel cm = new FloatDoubleColorModel(colorSpace,
                                          false,
                                          false,
                                          Transparency.OPAQUE,
                                        DataBuffer.TYPE_FLOAT);
```

**API:** `java.awt.image.ComponentColorModel`

- `ComponentColorModel(ColorSpace colorSpace, int[] bits,`
  `boolean hasAlpha, boolean isAlphaPremultiplied,`
  `int transparency, int transferType)`

  constructs a `ComponentColorModel` from the specified parameters.

| | | |
|---|---|---|
| *Parameters*: | `colorSpace` | The `ColorSpace` associated with this color model. See Section 5.2.2, “Color Space.” |
| | `bits` | The number of significant bits per component. |
| | `hasAlpha` | If true, this color model supports alpha. |
| | `isAlphaPremultiplied` | If true, alpha is premultiplied. |

| | |
|---|---|
| `transparency` | Specifies what alpha values can be represented by this color model. See Section 5.3, "Transparency." |
| `transferType` | Specifies the type of primitive array used to represent pixel values. One of `DataBuffer.TYPE_BYTE`, `DataBuffer.TYPE_INT`, `DataBuffer.TYPE_SHORT`, `DataBuffer.TYPE_USHORT`, `DataBuffer.TYPE_DOUBLE`, or `DataBuffer.TYPE_FLOAT` |

**API:** `javax.media.jai.FloatDoubleColorModel`

-
  ```
  FloatDoubleColorModel(ColorSpace colorSpace, boolean hasAlpha,
      boolean isAlphaPremultiplied, int transparency,
      int transferType)
  ```

  constructs a `FloatDoubleColorModel` from the specified parameters.

| | | |
|---|---|---|
| *Parameters*: | `colorSpace` | The `ColorSpace` associated with this color model. See Section 5.2.2, "Color Space." |
| | `hasAlpha` | If true, this color model supports alpha. |
| | `isAlphaPremultiplied` | If true, alpha is premultiplied. |
| | `transparency` | Specifies what alpha values can be represented by this color model. See Section 5.3, "Transparency." |
| | `transferType` | Specifies the type of primitive array used to represent pixel values. One of `DataBuffer.TYPE_FLOAT` or `DataBuffer.TYPE_DOUBLE`. |

### 5.2.2 Color Space

The `ColorSpace` class represents a system for measuring colors, typically using three or more separate numeric values. For example, RGB and CMYK are color spaces. A `ColorSpace` object serves as a color space tag that identifies the specific color space of a `Color` object or, through a `ColorModel` object, of an `Image`, `BufferedImage`, or `GraphicsConfiguration`.

ColorSpace provides methods that transform Colors in a specific color space to and from sRGB and to and from a well-defined CIEXYZ color space. All ColorSpace objects must be able to map a color from the represented color space into sRGB and transform an sRGB color into the represented color space.

Table 5-1 lists the variables used to refer to color spaces (such as CS_sRGB and CS_CIEXYZ) and to color space types (such as TYPE_RGB and TYPE_CMYK).

**Table 5-1 Color Space Types**

| Name | Description |
|---|---|
| CS_CIEXYZ | A widely-used, device-independent color standard developed by the Commission Internationale de Eclairage (CIE), based on color-matching experiments on human observers. |
| CS_GRAY | Grayscale color space. |
| CS_LINEAR_RGB | Linear RGB. Images that have not been previously color-corrected. |
| CS_PYCC | PhotoCD YCC conversion color space. A luminance/chrominance standard for Kodak PhotoCD images. |
| CS_sRGB | A proposed default “standard RGB” color standard for use over the Internet. |
| TYPE_2CLR | A generic two-component color space. |
| TYPE_3CLR | A generic three-component color space. |
| TYPE_4CLR | A generic four-component color space. |
| TYPE_5CLR | A generic five-component color space. |
| TYPE_6CLR | A generic six-component color space. |
| TYPE_7CLR | A generic seven-component color space. |
| TYPE_8CLR | A generic eight-component color space. |
| TYPE_9CLR | A generic nine-component color space. |
| TYPE_ACLR | A generic 10-component color space. |
| TYPE_BCLR | A generic 11-component color space. |
| TYPE_CCLR | A generic 12-component color space. |
| TYPE_CMY | Any of the family of CMY color spaces. |
| TYPE_CMYK | Any of the family of CMYK color spaces. |
| TYPE_DCLR | Generic 13-component color spaces. |
| TYPE_ECLR | Generic 14-component color spaces. |
| TYPE_FCLR | Generic 15-component color spaces. |
| TYPE_GRAY | Any of the family of GRAY color spaces. |
| TYPE_HLS | Any of the family of HLS color spaces. |
| TYPE_HSV | Any of the family of HSV color spaces. |

**Table 5-1 Color Space Types (Continued)**

| Name | Description |
|---|---|
| `TYPE_Lab` | Any of the family of Lab color spaces. |
| `TYPE_Luv` | Any of the family of Luv color spaces. |
| `TYPE_RGB` | Any of the family of RGB color spaces. |
| `TYPE_XYZ` | Any of the family of XYZ color spaces. |
| `TYPE_YCbCr` | Any of the family of YCbCr color spaces. |
| `TYPE_Yxy` | Any of the family of Yxy color spaces. |

Conversion between Java color spaces is simplified by a set of methods that map a color from a represented color space to either sRGB or CIEXYZ and transform a sRGB or CIEXYZ color space to the represented color space. There are four methods:

- The `toRGB` method transforms a `Color` in the represented color space to a `Color` in sRGB.
- The `toCIEXYZ` method transforms a `Color` in the represented color space to a `Color` in CIEXYZ.
- The `fromRGB` method takes a `Color` in sRGB and transforms into the represented color space.
- The `fromCIEXYZ` method takes a `Color` in CIEXYZ and transforms into the represented color space.

The sRGB (which stands for “standard” RGB) color space is provided as a convenience to programmers, since many applications are primarily concerned with RGB images. Defining a standard RGB color space makes writing such applications easier. The `toRGB` and `fromRGB` methods are provided so that developers can easily retrieve colors in this standard space. However, the sRGB color space is not intended for use with highly accurate color correction or conversions.

The sRGB color space is somewhat limited in that it cannot represent every color in the full gamut (spectrum of representable colors) of CIEXYZ color. If a color is specified in some space that has a different gammut than sRGB, using sRGB as an intermediate color space results in a loss of information. The CIEXYZ color space is used as an intermediate color space to avoid any loss of color quality. The CIEXYZ color space is known as the *conversion space* for this reason. The `toCIEXYZ` and `fromCIEXYZ` methods support conversions between any two color spaces at a reasonably high degree of accuracy, one color at a time.

**API:** `java.awt.color.ColorSpace`

- `abstract float[] toRGB(float[] colorvalue)`

  transforms a color value assumed to be in this `ColorSpace` into a value in the default `CS_sRGB` color space.

  | | | |
  |---|---|---|
  | *Parameter*: | `colorvalue` | A float array with length of at least the number of components in this `ColorSpace`. |

- `abstract float[] fromRGB(float[] rgbvalue)`

  transforms a color value assumed to be in the default `CS_sRGB` color space into this `ColorSpace`.

  | | | |
  |---|---|---|
  | *Parameter*: | `rgbvalue` | A float array with length of at least 3. |

- `abstract float[] toCIEXYZ(float[] colorvalue)`

  transforms a color value assumed to be in this `ColorSpace` into the `CS_CIEXYZ` conversion color space.

- `abstract float[] fromCIEXYZ(float[] colorvalue)`

  transforms a color value assumed to be in the `CS_CIEXYZ` conversion color space into this `ColorSpace`.

- `static ColorSpace getInstance(int colorspace)`

  returns a ColorSpace representing one of the specific predefined color spaces.

  | | | |
  |---|---|---|
  | *Parameter*: | `colorSpace` | A specific color space identified by one of the predefined class constants (e.g., `CS_sRGB`, `CS_LINEAR_RGB`, `CS_CIEXYZ`, `CS_GRAY`, or `CS_PYCC`). |

- `int getType()`

  returns the color space type of this `ColorSpace` (for example `TYPE_RGB`, `TYPE_XYZ`, etc.).

### 5.2.3 ICC Profile and ICC Color Space

The `ColorSpace` class is an abstract class. It is expected that particular implementations of subclasses of `ColorSpace` will support high performance conversion based on underlying platform color management systems. The `ICC_ColorSpace` class is one such implementation provided in the base AWT. Developers can define their own subclasses to represent arbitrary color spaces, as long as the appropriate “to” and “from” conversion methods are implemented. However, most developers can simply use the default `sRGB` color space or color spaces that are represented by commonly-available ICC profiles, such as profiles for monitors and printers or profiles embedded in image data.

The `ICC_ColorSpace` class is based on ICC profile data as represented by the `ICC_Profile` class. The `ICC_Profile` class is a representation of color profile data for device-independent and device-dependent color spaces based on the *ICC Profile Format Specification,* Version 3.4, August 15, 1997, from the International Color Consortium. ICC profiles describe an *input space* and a *connection space,* and define how to map between them.

The `ICC_Profile` class has two subclasses that correspond to the specific color types:

- `ICC_ProfileRGB`, which represents `TYPE_RGB` color spaces
- `ICC_ProfileGray`, which represents `TYPE_GRAY` color spaces

## 5.3 Transparency

Just as images can have color, they can also have transparency. Transparency defines the specular transmission of light through transparent materials, such as glass, or the lack of transparency for completely opaque objects. The amount of transparency is specified by an alpha ($\alpha$) value. An alpha value of 0.0 specifies complete translucency; an alpha value of 1.0 specifies complete opacity.

Images can carry transparency information, known as the alpha channel, for each pixel in the image. The alpha value is particularly important when colors overlap. The alpha value specifies how much of the previously-rendered color should show through.

The Java `Transparency` interface defines the common transparency modes for implementing classes. Table 5-2 lists the variables used to specify transparency.

**Table 5-2 Transparency**

| Name | Description |
|---|---|
| `BITMASK` | Represents image data that is guaranteed to be either completely opaque, with an alpha value of 1.0, or completely transparent, with an alpha value of 0.0. |
| `OPAQUE` | Represents image data that is guaranteed to be completely opaque, meaning that all pixels have an alpha value of 1.0. |
| `TRANSLUCENT` | Represents image data that contains or might contain arbitrary alpha values between and including 0.0 and 1.0. |

Transparency is specified as part of the color model (see Section 5.2.1, “Color Models”).

## 5.4 Color Conversion

The ColorConvert operation performs a pixel-by-pixel color conversion of the data in a rendered or renderable source image. The data are treated as having no alpha channel, i.e., all bands are color bands. The color space of the source image is specified by the ColorSpace object of the source image ColorModel which must not be null.

JAI does not attempt to verify that the ColorModel of the destination image is consistent with the ColorSpace parameter. To ensure that this is the case, a compatible ColorModel must be provided via an ImageLayout in the RenderingHints (see Section 3.7.3, “Rendering Hints”).

Integral data are assumed to occupy the full range of the respective data type; floating point data are assumed to be normalized to the range [0.0,1.0]. By default, the destination image bounds, data type, and number of bands are the same as those of the source image.

The ColorConvert operation takes one parameter:

| Parameters | Type | Description |
|---|---|---|
| colorSpace | ColorSpace | The destination color space. |

For information on color space, see Section 5.2.2, “Color Space.”

Listing 5-1 shows a code sample for a ColorConvert operation.

**Listing 5-1 Example ColorConvert Operation**

```
// Read the image from the specified file name.
RenderedOp src = JAI.create("fileload", fileName);

// Create the ParameterBlock.
ParameterBlock pb = new ParameterBlock();
pb.addSource(src).add(colorSpace);

// Perform the color conversion.
RenderedOp dst = JAI.create("ColorConvert", pb);
```

## 5.5 Non-standard Linear Color Conversion (BandCombine)

In JAI, the `BandCombine` operation performs a linear color conversion between color spaces other than those listed in Table 5-1. The `BandCombine` operation computes a set of arbitrary linear combinations of the bands of a rendered or renderable source image, using a specified matrix. The matrix must have dimension (# of source bands plus one) by (# of desired destination bands).

The `BandCombine` operation takes one parameter:

| Parameter | Type | Description |
|---|---|---|
| `matrix` | `double` | The matrix specifying the band combination. |

As an example, assume the three-band source image and the matrix shown in Figure 5-1. The equation to calculate the value of the destination pixel in this example would be:

$$dst = (255 * 0.25) + (157 * 0.5) + (28 * 0.75)$$

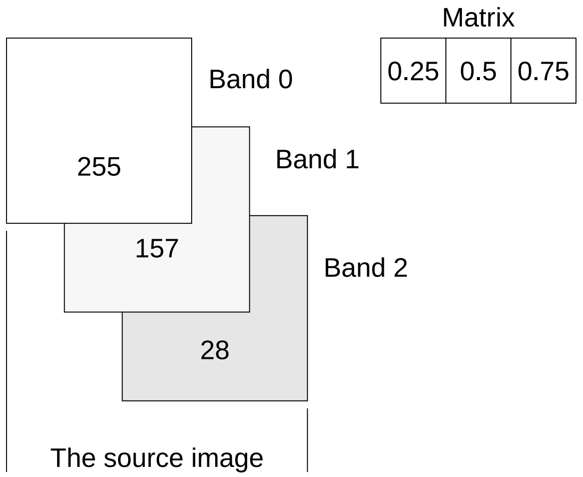


**Figure 5-1 Band Combine Example**

In this example, the number of columns in the matrix is equal to the number of bands in the source image. The number of rows in the matrix must equal the number of bands in the destination image. For a destination image with three bands, the values in the second row of the matrix would be used to calculate the values in the second band of the destination image and the values in the third row would be used to calculate the values in the third band.

If the result of the computation underflows or overflows the minimum or maximum value supported by the destination image, it will be clamped to the minimum or maximum value, respectively.

Listing 5-2 shows a code sample for a `BandCombine` operation.

**Listing 5-2 Example BandCombine Operation**

```
   // Create the matrix.
   // Invert center band.
        double[][] matrix = {
                   { 1.0D,  0.0D, 0.0D,   0.0D },
                   { 0.0D, -1.0D, 0.0D, 255.0D },
                   { 0.0D,  0.0D, 1.0D,   0.0D },
                };

   // Identity.
        double[][] matrix = {
                   { 1.0D, 0.0D, 0.0D, 0.0D },
                   { 0.0D, 1.0D, 0.0D, 0.0D },
                   { 0.0D, 0.0D, 1.0D, 0.0D },
                };

   // Luminance stored into red band (3 band).
         double[][] matrix = {
                    { .114D, 0.587D, 0.299D, 0.0D },
                    { .000D, 0.000D, 0.000D, 0.0D },
                    { .000D, 0.000D, 0.000D, 0.0D }
                 };

   // Luminance (single band output).
         double[][] matrix = {
                    { .114D, 0.587D, 0.299D, 0.0D }
                 };

   // Create the ParameterBlock.
   ParameterBlock pb = new ParameterBlock();
   pb.addSource(src_image);
   pb.add(matrix);

   // Perform the band combine operation.
   dst = (PlanarImage)JAI.create("bandcombine", pb, null);
```

CHAPTER 6

# Image Manipulation

THIS chapter describes the basics of manipulating images to prepare them for further processing.

## 6.1 Introduction

The JAI image manipulation objects and methods are used to enhance and geometrically modify images and to extract information from images. Image manipulation includes:

- Region of interest (ROI) control
- Relational operators
- Logical operators
- Arithmetic operators
- Dithering
- Clamping pixel values
- Band copy

## 6.2 Region of Interest Control

Typically, any image enhancement operation takes place over the entire image. While the image enhancement operation may improve portions of an image, other portions of the image may lose detail. You usually want some way of limiting the enhancement operation to specific regions of the image.

To restrict the image enhancement operations to specific regions of an image, a region-of-interest mask is created. A region of interest (ROI) is conceptually a mask of true or false values. The ROI mask controls which source image pixels are to be processed and which destination pixels are to be recorded.

JAI supports two different types of ROI mask: a Boolean mask and a threshold value. The `ROIShape` class uses a Boolean mask, which allows operations to be performed quickly and with compact storage. The `ROI` class allows the specification of a threshold value; pixel values greater than or equal to the threshold value are included in the ROI. Pixel values less than the threshold are excluded.

The region of interest is usually defined using a `ROIShape`, which stores its area using the `java.awt.Shape` classes. These classes define an area as a geometrical description of its outline. The `ROI` class stores an area as a single-banded image.

An ROI can be attached to an image as a property. See **Chapter 11, "Image Properties**."

### 6.2.1 The ROI Class

The `ROI` class stores an area as a grayscale (single-banded) image. This class represents region information in image form, and can thus be used as a fallback where a `Shape` representation is unavailable. Inclusion and exclusion of pixels is defined by a threshold value. Source pixel values greater than or equal to the threshold value indicate inclusion in the ROI and are processed. Pixel values less than the threshold value are excluded from processing.

Where possible, subclasses such as `ROIShape` are used since they provide a more compact means of storage for large regions.

The `getAsShape()` method may be called optimistically on any instance of `ROI`. However, it may return null to indicate that a `Shape` representation of the `ROI` is not available. In this case, `getAsImage()` should be called as a fallback.

**API:** `javax.media.jai.ROI`

- `ROI(RenderedImage im)`

  constructs an `ROI` object from a `RenderedImage`. The inclusion threshold is taken to be halfway between the minimum and maximum sample values specified by the image's `SampleModel`.

  | | | |
  |---|---|---|
  | *Parameters*: | `im` | A single-banded `RenderedImage`. |

- `ROI(RenderedImage im, int threshold)`

  constructs an `ROI` object from a `RenderedImage`. The inclusion `threshold` is specified explicitly.

| *Parameters*: | | |
|---|---|---|
| | `im` | A single-banded `RenderedImage`. |
| | `threshold` | The inclusion/exclusion threshold of the `ROI`. |

- `Shape getAsShape()`

  returns a `Shape` representation of the `ROI`, if possible. If none is available, null is returned. A proper instance of `ROI` (one that is not an instance of any subclass of `ROI`) will always return null.

- `PlanarImage getAsImage()`

  returns a `PlanarImage` representation of the `ROI`. This method will always succeed.

- `int getThreshold()`

  returns the inclusion/exclusion threshold value.

- `void setThreshold(int threshold)`

  sets the inclusion/exclusion threshold value.

#### 6.2.1.1 Determining the ROI Bounds

The `getBounds` methods in the `ROI` class read the bounds of the `ROI`, as either a `Rectangle` or a `Rectangle2D`.

**API:** `javax.media.jai.ROI`

- `Rectangle getBounds()`

  returns the bounds of the `ROI` as a `Rectangle`.

- `Rectangle2D getBounds2D()`

  returns the bounds of the `ROI` as a `Rectangle2D`.

#### 6.2.1.2 Determining if an Area Lies Within or Intersects the ROI

The `contains` methods in the `ROI` class test whether a given point or rectangular region lie within the `ROI`. The `intersects` methods test whether a given rectangular region intersect with the `ROI`.

**API:** `javax.media.jai.ROI`

- `boolean contains(Point p)`
  returns true if the `Point` lies within the `ROI`.

  | | | |
  |---|---|---|
  | *Parameters*: | `p` | A `Point` identifying the pixel to be queried. |

- `boolean contains(Point2D p)`
  returns true if the `Point2D` lies within the `ROI`.

  | | | |
  |---|---|---|
  | *Parameters*: | `p` | A `Point2D` identifying the pixel to be queried. |

- `boolean contains(int x, int y)`
  returns true if the point lies within the `ROI`.

  | | | |
  |---|---|---|
  | *Parameters*: | `x` | An int specifying the *x* coordinate of the pixel to be queried. |
  | | `y` | An int specifying the *y* coordinate of the pixel to be queried. |

- `boolean contains(double x, double y)`
  returns true if the point lies within the `ROI`.

  | | | |
  |---|---|---|
  | *Parameters*: | `x` | A double specifying the *x* coordinate of the pixel to be queried. |
  | | `y` | A double specifying the *y* coordinate of the pixel to be queried. |

- `boolean contains(Rectangle rect)`
  returns true if the `Rectangle` lies within the `ROI`.

  | | | |
  |---|---|---|
  | *Parameters*: | `rect` | A `Rectangle` specifying the region to be tested for inclusion. |

- `boolean contains(Rectangle2D r)`
  returns true if the `Rectangle2D` lies within the `ROI`.

  | | | |
  |---|---|---|
  | *Parameters*: | `r` | A `Rectangle2D` specifying the region to be tested for inclusion. |

- `boolean contains(int x, int y, int w, int h)`
  returns true if the rectangle lies within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `x` | The int *x* coordinate of the upper left corner of the region. |
| | `y` | The int *y* coordinate of the upper left corner of the region. |
| | `w` | The int width of the region. |
| | `h` | The int height of the region. |

- `boolean contains(double x, double y, double w, double h)`
  returns true if the rectangle lies within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `x` | The double *x* coordinate of the upper left corner of the region. |
| | `y` | The double *y* coordinate of the upper left corner of the region. |
| | `w` | The double width of the region. |
| | `h` | The double height of the region. |

- `boolean intersects(Rectangle rect)`
  returns true if the `Rectangle` intersects the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `rect` | A `Rectangle` specifying the region to be tested for inclusion. |

- `boolean intersects(Rectangle2D r)`
  returns true if the `Rectangle2D` intersects the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `r` | A `Rectangle2D` specifying the region to be tested for inclusion. |

- `boolean intersects(int x, int y, int w, int h)`

  returns true if the rectangle intersects the `ROI`.

| *Parameters*: | | |
|---|---|---|
| | `x` | The int *x* coordinate of the upper left corner of the region. |
| | `y` | The int *y* coordinate of the upper left corner of the region. |
| | `w` | The int width of the region. |
| | `h` | The int height of the region. |

- `boolean intersects(double x, double y, double w, double h)`

  returns true if the rectangle intersects the `ROI`.

| *Parameters*: | | |
|---|---|---|
| | `x` | The double *x* coordinate of the upper left corner of the region. |
| | `y` | The double *y* coordinate of the upper left corner of the region. |
| | `w` | The double width of the region. |
| | `h` | The double height of the region. |

### 6.2.1.3 Creating a New ROI from an Existing ROI

Several methods allow the creation of a new `ROI` from an existing `ROI`. The `add` method adds another ROI to an existing one, creating a new ROI.

**API:** `javax.media.jai.ROI`

- `ROI add(ROI im)`

  adds another `ROI` to this one and returns the result as a new `ROI`. The addition is performed by an “AddROIs” RIF to be specified. The supplied `ROI` will be converted to a rendered form if necessary.

| *Parameters*: | `im` | An `ROI`. |
|---|---|---|

- `ROI subtract(ROI im)`

  subtracts another `ROI` to this one and returns the result as a new `ROI`. The subtraction is performed by a “SubtractROIs” RIF to be specified. The supplied `ROI` will be converted to a rendered form if necessary.

| *Parameters*: | `im` | An `ROI`. |
|---|---|---|

- `ROI intersect(ROI im)`

  intersects the `ROI` with another `ROI` and returns the result as a new `ROI`. The intersection is performed by a “IntersectROIs” RIF to be specified. The supplied `ROI` will be converted to a rendered form if necessary.

| | | |
|---|---|---|
| *Parameters*: | `im` | An `ROI`. |

- `ROI exclusiveOr(ROI im)`

  exclusive-ORs the `ROI` with another `ROI` and returns the result as a new `ROI`. The intersection is performed by an “XorROIs” RIF to be specified. The supplied `ROI` will be converted to a rendered form if necessary.

| | | |
|---|---|---|
| *Parameters*: | `im` | An `ROI`. |

- `ROI transform(AffineTransform at)`

  performs an affine transformation and returns the result as a new `ROI`. The transformation is performed by an “Affine” RIF.

| | | |
|---|---|---|
| *Parameters*: | `at` | An `AffineTransform` specifying the transformation. |

-
  ```
  ROI performImageOp(RenderedImageFactory RIF, ParameterBlock
      paramBlock, int sourceIndex, Hashtable renderHints,
      Hashtable renderHintsObserved)
  ```

  transforms an `ROI` using an imaging operation. The operation is specified by a `RenderedImageFactory`. The operation’s `ParameterBlock`, minus the image source itself is supplied, along with an index indicating where to insert the `ROI` image. The usual `renderHints` and `renderHintsObserved` arguments allow rendering hints to be passed in and information on which hints were followed to be passed out.

| | | |
|---|---|---|
| *Parameters*: | `RIF` | A `RenderedImageFactory` that will be used to create the op. |
| | `paramBlock` | A `ParameterBlock` containing all sources and parameters for the operation except for the `ROI` itself. |
| | `sourceIndex` | The index of the `ParameterBlock`’s sources where the `ROI` is to be inserted. |
| | `renderHints` | A Hashtable of rendering hints. |
| | `renderHints-Observed` | A Hashtable of observed rendering hints. |

- `ROI performImageOp(RenderedImageFactory RIF, ParameterBlock paramBlock, int sourceIndex)`

  transforms an `ROI` using an imaging operation. The operation is specified by a `RenderedImageFactory`. The operation's `ParameterBlock`, minus the image source itself is supplied, along with an index indicating where to insert the `ROI` image. Rendering hints are taken to be null.

| | | |
|---|---|---|
| *Parameters*: | `RIF` | A `RenderedImageFactory` that will be used to create the op. |
| | `paramBlock` | A `ParameterBlock` containing all sources and parameters for the operation except for the `ROI` itself. |
| | `sourceIndex` | The index of the `ParameterBlock`'s sources where the `ROI` is to be inserted. |

- `ROI performImageOp(String name, ParameterBlock paramBlock, int sourceIndex, Hashtable renderHints, Hashtable renderHintsObserved)`

  transforms an `ROI` using an imaging operation. The operation is specified by name; the default JAI registry is used to resolve this into a RIF. The operation's `ParameterBlock`, minus the image source itself is supplied, along with an index indicating where to insert the `ROI` image. The usual `renderHints` and `renderHintsObserved` arguments allow rendering hints to be passed in and information on which hints were followed to be passed out.

| | | |
|---|---|---|
| *Parameters*: | `name` | The name of the operation to be performed. |
| | `paramBlock` | A `ParameterBlock` containing all sources and parameters for the operation except for the `ROI` itself. |
| | `sourceIndex` | The index of the `ParameterBlock`'s sources where the `ROI` is to be inserted. |
| | `renderHints` | A Hashtable of rendering hints. |
| | `renderHints-Observed` | A Hashtable of observed rendering hints. |

- `ROI performImageOp(String name, ParameterBlock paramBlock, int sourceIndex)`

  transforms an `ROI` using an imaging operation. The operation is specified by name; the default JAI registry is used to resolve this into a RIF. The operation's `ParameterBlock`, minus the image source itself is supplied, along with an index indicating where to insert the ROI image. Rendering hints are taken to be null.

| *Parameters*: | | |
|---|---|---|
| | name | The name of the operation to be performed. |
| | paramBlock | A ParameterBlock containing all sources and parameters for the operation except for the ROI itself. |
| | sourceIndex | The index of the ParameterBlock's sources where the ROI is to be inserted. |

- `Shape getAsShape()`

  returns a Shape representation of the ROI, if possible. If none is available, null is returned. A proper instance of ROI (one that is not an instance of any subclass of ROI) will always return null.

- `PlanarImage getAsImage()`

  returns a PlanarImage representation of the ROI. This method will always succeed.

### 6.2.2 The ROIShape Class

The ROIShape class is used to store a region of interest within an image as an instance of a java.awt.Shape. Such regions are binary by definition. Using a Shape representation allows Boolean operations to be performed quickly and with compact storage. If a PropertyGenerator responsible for generating the ROI property of a particular OperationDescriptor (such as a warp) cannot reasonably produce an ROIShape representing the region, it should call the getAsImage() method on its sources and produce its output ROI in image form.

**API:** javax.media.jai.ROIShape

- `ROIShape(Shape s)`

  constructs an ROIShape from a Shape.

| *Parameters*: | | |
|---|---|---|
| | s | A Shape. |

- `ROIShape(Area a)`

  constructs an ROIShape from an Area.

| *Parameters*: | | |
|---|---|---|
| | a | An Area. |

#### 6.2.2.1 Determining the ROI Bounds

The following methods in the `ROIShape` class read the bounds of the `ROI`.

**API:** `javax.media.jai.ROIShape`

- `Rectangle getBounds()`
  returns the bounds of the `ROI` as a `Rectangle`.

- `Rectangle2D getBounds2D()`
  returns the bounds of the `ROI` as a `Rectangle2D`.

#### 6.2.2.2 Determining if an Area Lies Within or Intersects the ROIShape

The `ROIShape.contains` method is used to determine if a given pixel lies within the region of interest. The `ROIShape.intersects` method is used to determine if a rectangular region of the image intersects the ROI.

**API:** `javax.media.jai.ROIShape`

- `boolean contains(Point p)`
  returns true if the pixel lies within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `p` | The coordinates of the pixel to be queried. |

- `boolean contains(Point2D p)`
  returns true if the pixel lies within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `p` | The coordinates of the pixel to be queried. |

- `boolean contains(int x, int y)`
  returns true if the pixel lies within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `x` | The *x* coordinate of the pixel to be queried. |
| | `y` | The *y* coordinate of the pixel to be queried. |

- `boolean contains(double x, double y)`
  returns true if the pixel lies within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `x` | The *x* coordinate of the pixel to be queried. |
| | `y` | The *y* coordinate of the pixel to be queried. |

- `boolean contains(Rectangle rect)`
  returns true if the rectangular region is entirely contained within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `rect` | The region to be tested for inclusion. |

- `boolean contains(Rectangle2D r)`
  returns true if the rectangular region is entirely contained within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `r` | The region to be tested for inclusion. |

- `boolean contains(int x, int y, int w, int h)`
  returns true if the rectangular region is entirely contained within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `x` | The *x* coordinate of the pixel to be queried. |
| | `y` | The *y* coordinate of the pixel to be queried. |
| | `w` | The width of the region. |
| | `h` | The height of the region. |

- `boolean contains(double x, double y, double w, double h)`
  returns true if the rectangular region is entirely contained within the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `x` | The *x* coordinate of the pixel to be queried. |
| | `y` | The *y* coordinate of the pixel to be queried. |
| | `w` | The width of the region. |
| | `h` | The height of the region. |

- `boolean intersects(Rectangle rect)`
  returns true if the rectangular region intersects the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `rect` | The region to be tested for inclusion. |

- `boolean intersects(Rectangle2D r)`
  returns true if the rectangular region intersects the `ROI`.

| | | |
|---|---|---|
| *Parameters*: | `rect` | The region to be tested for inclusion. |

- `boolean intersects(int x, int y, int w, int h)`
  returns true if the rectangular region intersects the `ROI`.

| *Parameters*: | `x` | The *x* coordinate of the upper left corner of the region. |
|---|---|---|
| | `y` | The *y* coordinate of the upper left corner of the region. |
| | `w` | The width of the region. |
| | `h` | The height of the region. |

- `boolean intersects(double x, double y, double w, double h)`
  returns true if the rectangular region intersects the `ROI`.

| *Parameters*: | `x` | The *x* coordinate of the upper left corner of the region. |
|---|---|---|
| | `y` | The *y* coordinate of the upper left corner of the region. |
| | `w` | The width of the region. |
| | `h` | The height of the region. |

#### 6.2.2.3 Creating a New ROIShape from an Existing ROIShape

Several methods allow the creation of a new `ROIShape` from the old `ROIShape`.

**API:** `javax.media.jai.ROIShape`

- `ROI add(ROI im)`
  adds another mask to this one. This operation may force this mask to be rendered.

| *Parameters*: | `im` | An `ROI`. |
|---|---|---|

- `ROI subtract(ROI im)`
  subtracts another mask from this one. This operation may force this mask to be rendered.

| *Parameters*: | `im` | An `ROI`. |
|---|---|---|

- `ROI intersect(ROI im)`

  sets the mask to its intersection with another mask. This operation may force this mask to be rendered.

  | | | |
  |---|---|---|
  | *Parameters*: | `im` | An `ROI`. |

- `ROI exclusiveOr(ROI im)`

  sets the mask to its exclusive-OR with another mask. This operation may force this mask to be rendered.

  | | | |
  |---|---|---|
  | *Parameters*: | `im` | An `ROI`. |

- `ROI transform(AffineTransform at)`

  performs an affine transformation and returns the result as a new `ROI`. The transformation is performed by an “Affine” RIF.

  | | | |
  |---|---|---|
  | *Parameters*: | `at` | The affine transform. |

- `Shape getAsShape()`

  returns the internal `Shape` representation or null if not possible. Since we have a shape available, we simply return it.

- `PlanarImage getAsImage()`

  returns the shape as a `PlanarImage`. This requires performing an antialiased rendering of the internal `Shape`. We use an eight-bit, single channel image with a `ComponentColorModel` and a `ColorSpace.TYPE_GRAY` color space.

## 6.3 Relational Operators

Given two source images and a destination image, the JAI relational operators allow you to:

- Find the larger of the pixels in the two source images and store the results in the destination (`Max`).
- Find the smaller of the pixels in the two source images and store the results in the destination (`Min`).

The relational operators require that both source images and the destination image have the same data type and number of bands. The sizes of the two images (height and width), however, need not be the same.

When determining the maximum and minimum pixels in the two images, JAI performs a band-by-band comparison.

**Note:** Don’t confuse the relational Min and Max operators with the Extrema operation (see Section 9.3, “Finding the Extrema of an Image”), which finds the image-wise minimum and maximum pixel values for each band of an image.

### 6.3.1 Finding the Maximum Values of Two Images

The max operation takes two rendered images, and for every pair of pixels, one from each source image of the corresponding position and band, finds the maximum pixel value.

The two source images may have different numbers of bands and data types. By default, the destination image bound is the intersection of the two source image bounds. If the two source images don’t intersect, the destination will have a width and a height of 0. The number of bands of the destination image is the same as the least number of bands of the source images, and the data type is the biggest data type of the source images.

The pixel values of the destination image are defined by the following pseudocode:

```
if (srcs[0][x][y][b] > srcs[1][x][y][b]) {
    dst[x][y][b] = srcs[0][x][y][b];
} else {
    dst[x][y][b] = srcs[1][x][y][b];
}
```

The max operation takes two source images and no parameters. Listing 6-1 shows a partial code sample of computing the pixelwise maximum value of two images in the rendered mode.

**Listing 6-1 Finding the Maximum Value of Two Images**

```
// Create two constant images
RenderedOp im0 = JAI.create(“constant”, param1);
RenderedOp im1 = JAI.create(“constant”, param2);

// Find the maximum value of the two images
RenderedOp im2 = JAI.create(“max”, im0, im1);
```

### 6.3.2 Finding the Minimum Values of Two Images

The `min` operation takes two rendered images, and for every pair of pixels, one from each source image of the corresponding position and band, finds the minimum pixel value.

The two source images may have different numbers of bands and data types. By default, the destination image bound is the intersection of the two source image bounds. If the two source images don't intersect, the destination will have a width and a height of 0. The number of bands of the destination image is the same as the least number of bands of the source images, and the data type is the biggest data type of the source images.

The pixel values of the destination image are defined by the following pseudocode:

```
if (srcs[0][x][y][b] < srcs[1][x][y][b]) {
    dst[x][y][b] = srcs[0][x][y][b];
} else {
    dst[x][y][b] = srcs[1][x][y][b];
}
```

The `min` operation takes two rendered source images and no parameters. Listing 6-2 shows a partial code sample of computing the pixelwise minimum value of two images in the renderable mode.

**Listing 6-2 Finding the Minimum Value of Two Images**

```
// Set up the parameter block and add the two source images to it
ParameterBlock pb = new ParameterBlock();
pb.add(im0);
pb.add(im1);

// Find the maximum value of the two images
RenderableOp im2 = JAI.createRenderable(“min”, pb, hints);
```

## 6.4 Logical Operators

JAI supports *monadic*, *dyadic*, and *unary* logical operators. The monadic logical operations include pixel-by-pixel AND, OR, and XOR operations between a source image and a constant to produce a destination image. The dyadic logical operations include pixel-by-pixel AND, OR, and XOR operations between two source images to produce a destination image. The unary logical operation is a NOT operation (complement image) on each pixel of a source image on a per-band basis.

JAI supports the following logical operations:

- Take the bitwise AND of the two source images and store the results in the destination (`And`)
- Take the bitwise AND of a source image and one of a set of per-band constants (`AndConst`)
- Take the bitwise OR of the two source images and store the results in the destination (`Or`)
- Take the bitwise OR of a source image and one of a set of per-band constants (`OrConst`)
- Take the bitwise XOR (exclusiveOR) of the two source images and store the results in the destination (`Xor`)
- Take the bitwise XOR of a source image and one of a set of per-band constants (`XorConst`)
- Take the bitwise NOT of a source image on each pixel on a per-band basis (`Not`)

As with the relational operators, the logical operations require that both source images and the destination image have the same data type and number of bands. The sizes of the two images (height and width), however, need not be the same.

## 6.4.1 ANDing Two Images

The `And` operation takes two rendered or renderable source images, and performs a bit-wise logical AND on every pair of pixels, one from each source image, of the corresponding position and band.

Both source images must have integral data types. The two data types may be different.

Unless altered by an `ImageLayout` hint, the destination image bound is the intersection of the two source image bounds. If the two sources don't intersect, the destination will have a width and height of 0. The number of bands of the destination image is equal to the lesser number of bands of the source images, and the data type is the smallest data type with sufficient range to cover the range of both source data types.

The following matrix defines the logical `And` operation.

| **src0** | **src1** | **Result** |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The destination pixel values are defined by the following pseudocode:

```
dst[x][y][b] = srcs[0][x][y][b] & srcs[1][x][y][b];
```

The `And` operation takes two rendered or renderable source images and no parameters.

Listing 6-3 shows a partial code sample of using the `And` operation to AND two images together.

**Listing 6-3 ANDing Two Images**

```
// Set up the parameter block and add the two source images to it.
ParameterBlock pb = new ParameterBlock();
pb.addSource(im0);          // The first image
pb.addSource(im1);          // The second image

// AND the two images together.
RenderableOp op = JAI.createRenderable("and", pb, hints);
```

## 6.4.2 ANDing an Image with a Constant

The `AndConst` operation takes one rendered or renderable image and an array of integer constants, and performs a bit-wise logical AND between every pixel in the same band of the source and the constant from the corresponding array entry. If the number of constants supplied is less than the number of bands of the destination, then the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band.

The source image must have an integral data type. By default, the destination image bound, data type, and number of bands are the same as the source image.

The following matrix defines the logical `AndConst` operation:

| **src** | **const** | **Result** |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |

| src | const | Result |
|---|---|---|
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The destination pixel values are defined by the following pseudocode:

```
if (constants.length < dstNumBands) {
    dst[x][y][b] = srcs[x][y][b] & constants[0];
} else {
    dst[x][y][b] = srcs[x][y][b] & constants[b];
}
```

The `AndConst` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `constants` | `int` | The per-band constants to logically AND with. |

Listing 6-4 shows a partial code sample of using the `AndConst` operation to AND a source image with a defined constant of value 1.2.

**Listing 6-4 ANDing an Image with a Constant**

```
// Set up the parameter block with the source and a constant
// value.
ParameterBlock pb = new ParameterBlock();
pb.addSource(im);       // im as the source image
pb.add(1.2f);     // The constant

// AND the image with the constant.
RenderableOp op = JAI.createRenderable(“andconst”, pb, hints);
```

## 6.4.3 ORing Two Images

The `Or` operation takes two rendered or renderable images, and performs a bit-wise logical OR on every pair of pixels, one from each source image of the corresponding position and band.

Both source images must have integral data types. The two data types may be different.

Unless altered by an `ImageLayout` hint, the destination image bound is the intersection of the two source image bounds. If the two sources don’t intersect, the destination will have a width and height of 0. The number of bands of the destination image is equal to the lesser number of bands of the source images, and the data type is the smallest data type with sufficient range to cover the range of both source data types.

The following matrix defines the logical `OR` operation:

| src0 | src1 | Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

The destination pixel values are defined by the following pseudocode:

```
dst[x][y][b] = srcs[0][x][y][b] | srcs[1][x][y][b];
```

The `Or` operation takes two rendered or renderable source images and no parameters.

Listing 6-5 shows a partial code sample of using the `or` operation to OR two images.

**Listing 6-5 ORing Two Images**

```
// Read the first image.
pb = new ParameterBlock();
pb.addSource(file1);
RenderedOp src1 = JAI.create("stream", pb);

// Read the second image.
pb = new ParameterBlock();
pb.addSource(file2);
RenderedImage src2 = JAI.create("stream", pb);

// OR the two images.
RenderedOp dst = JAI.create("or", src1, src2);
```

### 6.4.4 ORing an Image with a Constant

The `OrConst` operation takes one rendered or renderable image and an array of integer constants, and performs a bit-wise logical OR between every pixel in the same band of the source image and the constant from the corresponding array entry. If the number of constants supplied is less than the number of bands of the destination, the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band.

The source image must have an integral data type. By default, the destination image bound, data type, and number of bands are the same as the source image.

The following matrix defines the logical `OrConst` operation:

| src | const | Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

The destination pixel values are defined by the following pseudocode:

```
if (constants.length < dstNumBands) {
    dst[x][y][b] = src[x][y][b] | constants[0];
} else {
    dst[x][y][b] = src[x][y][b] | constants[b];
}
```

The `OrConst` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `constants` | `int` | The per-band constants to logically OR with. |

## 6.4.5 XORing Two Images

The `Xor` operation takes two rendered or renderable images, and performs a bit-wise logical XOR on every pair of pixels, one from each source image of the corresponding position and band.

Both source images must have integral data types. The two data types may be different.

Unless altered by an `ImageLayout` hint, the destination image bound is the intersection of the two source image bounds. If the two source images don't intersect, the destination will have a width and height of 0. The number of bands of the destination image is equal to the lesser number of bands of the source images, and the data type is the smallest data type with sufficient range to cover the range of both source data types.

The following matrix defines the `Xor` operation:

| src0 | src1 | Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The destination pixel values are defined by the following pseudocode:

```
dst[x][y][b] = srcs[0][x][y][b] ^ srcs[0][x][y][b];
```

The `Xor` operation takes one rendered or renderable source image and no parameters.

## 6.4.6 XORing an Image with a Constant

The `XorConst` operation takes one rendered or renderable image and an array of integer constants, and performs a bit-wise logical OR between every pixel in the same band of the source and the constant from the corresponding array entry. If the number of constants supplied is less than the number of bands of the destination, the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band.

The source image must have an integral data type. By default, the destination image bound, data type, and number of bands are the same as the source image.

The following matrix defines the logical `XorConst` operation:

| src | const | Result |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The destination pixel values are defined by the following pseudocode:

```
if (constants.length < dstNumBands) {
    dst[x][y][b] = src[x][y][b] ^ constants[0];
} else {
    dst[x][y][b] = src[x][y][b] ^ constants[b];
}
```

The `XorConst` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `constant` | `int` | The constant to logically XOR with. |

### 6.4.7 Taking the Bitwise NOT of an Image

The `Not` operation takes one rendered or renderable image, and performs a bitwise logical NOT on every pixel from every band of the source image. This operation, also known as a *complement* operation, creates an image that is somewhat like a photographic negative.

The `Not` operation looks at the values in the source image as binary values and changes all the 1’s in those values to 0’s, and all the 0’s to 1’s. The operation then writes the one’s complement version of the source image to the destination.

The source image must have an integral data type. By default, the destination image bound, data type, and number of bands are the same as the source image.

The following matrix defines the logical NOT operation.

| src | Result |
|---|---|
| 1 | 0 |
| 0 | 1 |

The pixel values of the destination image are defined by the following pseudocode:

```
dst[x][y][b] = ~(src[x][y][b])
```

The `Not` operation takes one rendered or renderable source image and no parameters.

Listing 6-6 shows a partial code sample of using the `Not` operation.

**Listing 6-6 Taking the NOT of an Image**

```
// Read the source image.
pb = new ParameterBlock();
pb.addSource(file);
RenderedOp src = JAI.create("stream", pb);

// Create the Not operation.
RenderedOp dst = JAI.create("Not", src);
```

## 6.5 Arithmetic Operators

JAI supports both *monadic* and *dyadic* arithmetic operators. The monadic arithmetic operations include per-band addition, subtraction, division, and multiplication operations between a source image and a constant to produce a destination image. The dyadic arithmetic operations include per-band addition, subtraction, division, and multiplication operations between two source images to produce a destination image.

The JAI arithmetic operators allow you to:

- Add two source images and store the results in a destination image (`Add`)
- Add a constant value to the pixels in a source image and store the results in a destination image (`AddConst`)
- Add a collection of images and store the results in a destination image (`AddCollection`)
- Add a an array of double constants to a collection of rendered images (`AddConstToCollection`)
- Subtract one source image from an other and store the results in a destination image (`Subtract`)
- Subtract a constant value from the pixels in a source image and store the results in a destination image (`SubtractConst`)
- Divide one source image into an other and store the results in a destination image (`Divide`)
- Divide two source images of complex data and store the results in a destination image (`DivideComplex`)
- Divide a source image by a constant value (`DivideByConst`)
- Divide a source image into a constant value (`DivideIntoConst`)
- Multiply two source images and store the results in a destination image (`Multiply`)
- Multiply a source image by a constant value (`MultiplyConst`)
- Multiply two images representing complex data (`MultiplyComplex`)
- Find the absolute value of pixels in a source image and store the results in a destination image (`Absolute`)
- Take the exponent of an image and store the results in a destination image (`Exp`)

As with the relational and logical operators, the arithmetic operations require that both source images and the destination image have the same data type and number of bands. The sizes of the two images (height and width), however, need not be the same.

When JAI adds two images, it takes the value at location 0,0 in one source image, adds it to the value at location 0,0 in the second source image, and writes the sum at location 0,0 in the destination image. It then does the same for all other points in the images. Subtraction, multiplication, and division are handled similarly.

Arithmetic operations on multi-band images are performed on corresponding bands in the source images. That is, band 0 of the first image is added to band 0 of the second image, and so on.

## 6.5.1 Adding Two Source Images

The `Add` operation takes two rendered or renderable source images, and adds every pair of pixels, one from each source image of the corresponding position and band. The two source images may have different numbers of bands and data types. By default, the destination image bounds are the intersection of the two source image bounds. If the sources don’t intersect, the destination will have a width and height of 0.

The default number of bands of the destination image is equal to the smallest number of bands of the sources, and the data type is the smallest data type with sufficient range to cover the range of both source data types (not necessarily the range of their sums).

As a special case, if one of the source images has $N$ bands (where $N$ is greater than one), the other source has one band, and an `ImageLayout` hint is provided containing a destination `SampleModel` with $K$ bands ($1 < K \leq N$), then the single band of the one1-banded source is added to each of the first $K$ bands of the $N$-band source.

The destination pixel values are defined by the following pseudocode:

```
dst[x][y][dstBand] = clamp(srcs[0][x][y][src0Band] +
                           srcs[1][x][y][src1Band]);
```

If the result of the addition underflows or overflows the minimum or maximum value supported by the destination image, the value will be clamped to the minimum or maximum value, respectively.

The `Add` operation two rendered or renderable source images and no parameters.

Listing 6-7 shows a partial code sample of using the `Add` operation to add two images.

**Listing 6-7  Adding Two Images**

```
// Read the two images.
pb = new ParameterBlock();
pb.addSource(s1);
RenderedImage src1 = (RenderedImage)JAI.create("stream", pb);

pb = new ParameterBlock();
pb.addSource(s2);
RenderedImage src2 = (RenderedImage)JAI.create("stream", pb);

// Create the ParameterBlock for the operation
pb = new ParameterBlock();
pb.addSource(src1);
pb.addSource(src2);

// Create the Add operation.
RenderedImage dst = (RenderedImage)JAI.create("add", pb);
```

## 6.5.2 Adding a Constant Value to an Image

The `AddConst` operation adds one of a set of constant values to every pixel value of a source image on a per-band basis:

```
if (constants.length < dstNumBands) {
    dst[x][y][b] = src[x][y][b] + constants[0];
else {
    dst[x][y][b] = src[x][y][b] + constants[b]
```

The `AddConst` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `constants` | `double` | The per-band constants to be added. |

The set of `constants` must contain one entry for each band of the source image. If the number of constants supplied is less than the number of bands of the destination image, the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band.

By default, the destination image bound, data type, and number of bands are the same as the source image.

If the result of the addition underflows or overflows the minimum or maximum value supported by the destination image, the value will be clamped to the minimum or maximum value, respectively.

Listing 6-8 shows a partial code sample of using the `AddConst` operation.

**Listing 6-8 Adding a Constant to an Image**

```
// Create the constant values.
RenderedImage im1, im2;
ParameterBlock pb;
double k0, k1, k2;

pb = new ParameterBlock();
pb.addSource(im1);
double[] constants = new double[3]; // or however many bands
                                    // in im1
constants[0] = k0;
constants[1] = k1;
constants[2] = k2;
pb.add(constants);

// Construct the AddConst operation.
RenderedImage addConstImage = JAI.create("addconst", pb, null);
```

## 6.5.3 Adding a Collection of Images

The `AddCollection` operation takes a collection of rendered images and adds every set of pixels, one from each source image of the corresponding position and band.

There's no restriction on the actual class type used to represent the source collection, but each element of the collection must be of the class `RenderedImages`. The number of images in the collection may vary from two to *n*, and is only limited by memory size. The source images may have different number of bands and data types.

By default, the destination image bound is the intersection of all the source image bounds. If any of the two sources don't intersect, the destination will have a width and a height of 0. The number of bands of the destination image is the same as the least number of bands of all the sources, and the data type is the biggest data type of all the sources.

The destination pixel values are calculated as:

```
dst[x][y][b] = 0;
for (int i = 0; i < numSources; i++) {
        dst[x][y][b] += srcs[i][x][y][b];
    }
```

If the result of the operation underflows or overflows the minimum or maximum value supported by the destination data type, the value will be clamped to the minimum or maximum value, respectively.

The `AddCollection` operation takes a collection of source images and no parameters.

### 6.5.4 Adding Constants to a Collection of Rendered Images

The `AddConstToCollection` operation takes a collection of rendered images and an array of double constants, and for each rendered image in the collection adds a constant to every pixel of its corresponding band.

The operation will attempt to store the result images in the same collection class as that of the source images. If a new instance of the source collection class can not be created, the operation will store the result images in a `java.util.Vector`. The output collection will contain the same number of images as in the source collection.

The `AddConstToCollection` operation takes a collection of rendered images and one parameter.

| Parameter | Type | Description |
|---|---|---|
| `constants` | `double` | The constants to be added. |

If the number of constants supplied is less than the number of bands of the source image, the same constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band.

### 6.5.5 Subtracting Two Source Images

The `Subtract` operation takes two rendered or renderable images, and for every pair of pixels, one from each source image of the corresponding position and band, subtracts the pixel from the second source from the pixel from the first source.

The two source images may have different numbers of bands and data types. By default, the destination image bounds are the intersection of the two source image bounds. If the sources don’t intersect, the destination will have a width and height of 0.

The default number of bands of the destination image is equal to the smallest number of bands of the source images, and the data type is the smallest data type with sufficient range to cover the range of both source data types (not necessarily the range of their sums).

As a special case, if one of the source images has *N* bands (where *N* is greater than one), the other source has one band, and an `ImageLayout` hint is provided containing a destination `SampleModel` with *K* bands ($1 < K \leq N$), then the single band of the one-banded source is subtracted from or into each of the first *K* bands of the *N*-band source.

The destination pixel values are defined by the following pseudocode:

```
dst[x][y][dstBand] = clamp(srcs[0][x][y][src0Band] -
                           srcs[1][x][y][src1Band]);
```

If the result of the subtraction underflows or overflows the minimum or maximum value supported by the destination image, the value will be clamped to the minimum or maximum value respectively.

The `Subtract` operation takes two rendered or renderable source images and no parameters.

### 6.5.6 Subtracting a Constant from an Image

The `Subtract`Const operation takes one rendered or renderable image and an array of double constants, and subtracts every pixel of the same band of the source from the constant from the corresponding array entry. If the number of constants supplied is less than the number of bands of the destination, the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band.

By default, the destination image bound, data type, and number of bands are the same as the source image.

The destination pixel values are defined by the following pseudocode:

```
if (constants.length < dstNumBands) {
    dst[x][y][b] = constants[0] - src[x][y][b];
} else {
    dst[x][y][b] = constants[b] - src[x][y][b];
}
```

The SubtractConst operation takes rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| constants | double | The per-band constants to be subtracted. |

If the result of the subtraction underflows or overflows the minimum or maximum value supported by the destination image, the value will be clamped to the minimum or maximum value respectively.

## 6.5.7 Subtracting an Image from a Constant

The SubtractFromConst operation takes one rendered or renderable source image and an array of double constants, and subtracts a constant from every pixel of its corresponding band of the source image. If the number of constants supplied is less than the number of bands of the destination, the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band. By default, the destination image bounds, data type, and number of bands are the same as the source image.

The destination pixel values are defined by the following pseudocode:

```
if (constants.length < dstNumBands) {
    dst[x][y][b] = src[x][y][b] - constants[0];
} else {
    dst[x][y][b] = src[x][y][b] - constants[b];
}
```

The SubtractFromConst operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| constants | double | The constants to be subtracted. |

If the result of the subtraction underflows or overflows the minimum or maximum value supported by the destination image, the value will be clamped to the minimum or maximum value respectively.

## 6.5.8 Dividing One Image by Another Image

The Divide operation takes two rendered or renderable images, and for every pair of pixels, one from each source image of the corresponding position and band, divides the pixel from the first source by the pixel from the second source.

In case of division by 0, if the numerator is 0, the result is set to 0; otherwise, the result is set to the maximum value supported by the destination data type.

The `Divide` operation does not require any parameters.

The two source images may have different number of bands and data types. By default, the destination image bound is the intersection of the two source image bounds. If the two sources don't intersect, the destination will have a width and a height of 0. The default number of bands of the destination image is the same as the least number of bands of the source images, and the data type is the biggest data type of the sources.

As a special case, if one of the source images has *N* bands (where *N* is greater than one), the other source has one band, and an `ImageLayout` hint is provided containing a destination `SampleModel` with *K* bands ($1 < K \leq N$), then the single band of the one-banded source will be divided by or into to each of the first *K* bands of the *N*-band source.

If the result of the operation underflows or overflows the minimum or maximum value supported by the destination data type, it will be clamped to the minimum or maximum value respectively.

The `Divide` operation takes two rendered or renderable source images and no parameters.

## 6.5.9 Dividing an Image by a Constant

The `DivideByConst` operation takes one rendered or renderable source image and an array of double constants, and divides every pixel of the same band of the source by the constant from the corresponding array entry. If the number of constants supplied is less than the number of bands of the destination, the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band.

In case of division by 0, if the numerator is 0, the result is set to 0. Otherwise, the result is set to the maximum value supported by the destination data type. By default, the destination image bound, data type, and number of bands are the same as the source image.

The destination pixel values are defined by the following pseudocode:

```
if (constants.length < dstNumBands) {
    dst[x][y][b] = srcs[x][y][b]/constants[0];
} else {
    dst[x][y][b] = srcs[x][y][b]/constants[b];
}
```

The `DivideByConst` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `constants` | `double` | The per-band constants to divide by. |

If the result of the division underflows or overflows the minimum or maximum value supported by the destination image, the value will be clamped to the minimum or maximum value, respectively.

## 6.5.10 Dividing an Image into a Constant

The `DivideIntoConst` operation takes one rendered or renderable image and an array of double constants, and divides every pixel of the same band of the source into the constant from the corresponding array entry. If the number of constants supplied is less than the number of bands of the destination, the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band.

In case of division by 0, if the numerator is 0, the result is set to 0. Otherwise, the result is set to the maximum value supported by the destination data type.

By default, the destination image bound, data type, and number of bands are the same as the source image.

The destination pixel values are defined by the following pseudocode:

```
if (constants.length < dstNumBands) {
    dst[x][y][b] = constants[0]/src[x][y][b];
} else {
    dst[x][y][b] = constants[b]/src[x][y][b];
}
```

The `DivideIntoConst` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `constants` | `double` | The per-band constants to be divided into. |

If the result of the division underflows or overflows the minimum or maximum value supported by the destination image, the value will be clamped to the minimum or maximum value, respectively.

### 6.5.11 Dividing Complex Images

The `DivideComplex` operation divides two images representing complex data. The source images must each contain an even number of bands with the even-indexed bands (0, 2, etc.) representing the real and the odd-indexed bands (1, 3, etc.) the imaginary parts of each pixel. The destination image similarly contains an even number of bands with the same interpretation and with contents defined by:

```
a = src0[x][y][2k];
b = src0[x][y][2k + 1];
c = src1[x][y][2k];
d = src1[x][y][2k + 1];

dst[x][y][2k] = (a*c + b*d)/(c2 + d2)
dst[x][y][2k + 1] = (b*c - a*d)/(c2 + d2)
```

where $0 \le k < \frac{\text{numBands}}{2}$

With one exception, the number of bands of the destination image is the same as the minimum of the number of bands of the two sources, and the data type is the biggest data type of the sources. The exception occurs when one of the source images has two bands, the other source image has $N = 2K$ bands where $K$ is greater than one, and an `ImageLayout` hint is provided containing a destination `SampleModel` that specifies $M = 2L$ bands for the destination image where $L$ is greater than one and $L \le K$. In this special case if the first source has two bands, its single complex component will be divided by each of the first $L$ complex components of the second source. If the second source has two bands, its single complex component will divide each of the $L$ complex components of the first source.

If the result of the operation underflows or overflows the minimum or /maximum value supported by the destination data type, it will be clamped to the minimum or maximum value, respectively.

The `DivideComplex` operation takes two rendered or renderable source images representing complex data and no parameters.

### 6.5.12 Multiplying Two Images

The `Multiply` operation takes two rendered or renderable images, and multiplies every pair of pixels, one from each source image of the corresponding position and band.

The two source images may have different number of bands and data types. By default, the destination image bound is the intersection of the two source image bounds. If the two source images don't intersect, the destination will have a width and a height of 0.

The default number of bands of the destination image is the same as the least number of bands of the source images, and the data type is the biggest data type of the source images. A special case may occur if one of the source images has *N* bands where *N* is greater than one, the other source has one band, and an `ImageLayout` hint is provided containing a destination `SampleModel`. If the `SampleModel` hint specifies *K* bands for the destination image where *K* is greater than one and $K \leq N$, each of the first *K* bands of the *N*-band source is multiplied by the single band of the one-band source.

In the default case the destination pixel values are calculated as:

```
for (int h = 0; h < dstHeight; h++) {
     for (int w = 0; w < dstWidth; w++) {
          for (int b = 0; b < dstNumBands; b++) {
               dst[h][w][b] = src1[h][w][b] * src2[h][w][b];
          }
     }
}
```

The `Multiply` operation takes two rendered or renderable source images and no parameters.

If the result of the multiplication underflows or overflows the minimum or maximum value supported by the destination image, the value will be clamped to the minimum or maximum value, respectively.

## 6.5.13 Multiplying an Image by a Constant

The `MultiplyConst` operation takes one rendered or renderable image and an array of double constants, and multiplies every pixel of the same band of the source by the constant from the corresponding array entry. If the number of constants supplied is less than the number of bands of the destination, the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band. By default, the destination image bound, data type, and number of bands are the same as the source image.

The destination pixel values are calculated as:

```
if (constants.length < dstNumBands) {
    dst[x][y][b] = srcs[x][y][b]*constants[0];
} else {
        dst[x][y][b] = srcs[x][y][b]*constants[b];
    }
```

The `MultiplyConst` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `constants` | `double` | The per-band constants to multiply by. |

If the result of the multiplication underflows or overflows the minimum or maximum value supported by the destination image, the value will be clamped to the minimum or maximum value respectively.

## 6.5.14 Multiplying Two Complex Images

The `MultiplyComplex` operation multiplies two images representing complex data. The source images must each contain an even number of bands, with the with the even-indexed bands (0, 2, etc.) representing the real and the odd-indexed bands (1, 3, etc.) the imaginary parts of each pixel. The destination image similarly contains an even number of bands with the same interpretation and with contents defined by:

$a$ = src0[$x$][$y$][2$k$];
$b$ = src0[$x$][$y$][2$k$ + 1];
$c$ = src1[$x$][$y$][2$k$];
$d$ = src1[$x$][$y$][2$k$ + 1];

dst[$x$][$y$][2$k$] = $a*c - b*d$;
dst[$x$][$y$][2$k$ + 1] = $a*d + b*c$;

$$\text{where } 0 \le k < \frac{\text{numBands}}{2}$$

With one exception, the number of bands of the destination image is the same as the minimum of the number of bands of the two source images, and the data type is the biggest data type of the sources. The exception occurs when one of the source images has two bands, the other source image has $N = 2K$ bands where $K$ is greater than one, and an `ImageLayout` hint is provided containing a destination `SampleModel` that specifies $M = 2L$ bands for the destination image where $L$ is greater than one and $L \le K$. In this special case each of the first $L$ complex components in the $N$-band source will be multiplied by the single complex component in the one-band source.

If the result of the operation underflows or overflows the minimum or maximum value supported by the destination data type, it will be clamped to the minimum or maximum value, respectively.

The `MultiplyComplex` operation takes two rendered source images representing complex data and no parameters.

### 6.5.15 Finding the Absolute Value of Pixels

Images with signed integer pixels have an asymmetrical range of values from –32,768 to 32,767, which is not very useful for many imaging operations. The `Absolute` operation takes a single rendered or renderable source image, and computes the mathematical absolute value of each pixel:

```
if (src[x][y][b] < 0) {
     dst[x][y][b] = -src[x][y][b];
 } else {
     dst[x][y][b] = src[x][y][b];
 }
```

For signed integral data types, the smallest value of the data type does not have a positive counterpart; such values will be left unchanged. This behavior parallels that of the Java unary minus operator.

The `Absolute` operation takes one rendered or renderable source image and no parameters

### 6.5.16 Taking the Exponent of an Image

The `Exp` operation takes the exponential of the pixel values of an image. The pixel values of the destination image are defined by the following pseudocode:

```
dst[x][y][b] = java.lang.Math.exp(src[x][y][b])
```

For integral image datatypes, the result will be rounded and clamped as needed.

The `Exp` operation takes one rendered or renderable source image and no parameters.

Listing 6-9 shows a partial code sample of using the `Exp` operation to take the exponent of an image.

**Listing 6-9 Taking the Exponent of an Image**

```
// Create a ParameterBlock with the source image.
pb = new ParameterBlock();
pb.addSource(src);
```

**Listing 6-9 Taking the Exponent of an Image (Continued)**

```
   // Perform the Exp operation
   RenderedImage dst = JAI.create("exp", pb);
```

## 6.6 Dithering an Image

The display of a 24-bit color image on an 8-bit frame buffer requires an operation known as *dithering*. The dithering operation compresses the three bands of an RGB image to a single-banded byte image.

The dithering operation uses a lookup table through which the source image is passed to produce the destination image. The most-common use for the dithering operation is to convert true-color (three-band byte) images to pseudo-color (single-band byte) images.

JAI offers two operations for dithering an image: ordered dither and error-diffusion dither. The choice of dithering operation depends on desired speed and image quality, as shown in Table 6-1.

**Table 6-1 Dithering Choices**

| Dither Type | Relative Speed | Relative Quality |
|---|---|---|
| Ordered | Medium | Medium |
| Error diffusion | Slowest | Best |

### 6.6.1 Ordered Dither

The ordered dithering operation is somewhat faster than the error-diffusion dither and produces a somewhat better destination image quality than the error-diffusion dither. The `OrderedDither` operation also differs from error-diffusion dither in that it (`OrderedDither`) uses a color cube rather than a general lookup table.

The `OrderedDither` operation performs color quantization by finding the nearest color to each pixel in a supplied color cube lookup table and “shifting” the resulting index value by a pseudo-random amount determined by the values of a supplied *dither mask*.

The `OrderedDither` operation takes one rendered source image and two parameters:

| Parameter | Type | Description |
|---|---|---|
| `colorMap` | `ColorCube` | The color cube. See Section 6.6.1.1, "Color Map Parameter." |
| `ditherMask` | `KernelJAI[]` | The dither mask. See Section 6.6.1.2, "Dither Mask Parameter." |

### 6.6.1.1 Color Map Parameter

The `colorMap` parameter can be either one of the predefined `ColorCubes`, or a custom color map can be created as a `ColorCube` object. To create a custom color map, see Section 7.6.1.3, "Creating a Color-cube Lookup Table."

The predefined color maps are:

| colorMap | Description |
|---|---|
| `BYTE_496` | A `ColorCube` with dimensions 4:9:6, useful for dithering RGB images into 216 colors. The offset of this ColorCube is 38. This color cube dithers blue values in the source image to one of four blue levels, green values to one of nine green levels, and red values to one of six red levels. This is the default color cube for the ordered dither operation. |
| `BYTE_855` | A `ColorCube` with dimensions 8:5:5, useful for dithering YCbCr images into 200 colors. The offset of this ColorCube is 54. This color cube dithers blue values in the source image to one of eight blue levels, green values to one of five green levels, and red values to one of five red levels. |

### 6.6.1.2 Dither Mask Parameter

The dither mask is a three-dimensional array of floating point values, the depth of which equals the number of bands in the image. The dither mask is supplied as an array of `KernelJAI` objects. Each element of the array is a `KernelJAI` object that represents the dither mask matrix for the corresponding band. All `KernelJAI` objects in the array must have the same dimensions and contain floating point values greater than or equal to 0.0 and less than or equal to 1.0.

The `ditherMask` parameter may either be one of the predefined dither masks or a custom mask may be created. To create a custom dither mask, see Section 6.9, "Constructing a Kernel."

The predefined dither masks are (see Figure 6-1):

| **ditherMask** | **Description** |
|---|---|
| `DITHER_MASK_441` | A 4 × 4 × 1 mask useful for dithering eight-bit grayscale images to one-bit images |
| `DITHER_MASK_443` | A 4 × 4 × 3 mask useful for dithering 24-bit color images to eight-bit pseudocolor images. This is the default dither mask for the `OrderedDither` operation. |

| | | | |
|---|---|---|---|
| 0.9375 | 0.4375 | 0.8125 | 0.3125 |
| 0.1875 | 0.6875 | 0.0625 | 0.5625 |
| 0.7500 | 0.2500 | 0.8750 | 0.3750 |
| 0.0000 | 0.5000 | 0.1250 | 0.6250 |

4 X 4 X 1 dither mask
(`DITHER_MASK_441`)

| | | | |
|---|---|---|---|
| 0.0000 | 0.5000 | 0.1250 | 0.6250 |
| 0.7500 | 0.2500 | 0.8750 | 0.3750 |
| 0.1875 | 0.6875 | 0.0625 | 0.5625 |
| 0.9375 | 0.4375 | 0.8125 | 0.3125 |

| | | | |
|---|---|---|---|
| 0.6250 | 0.1250 | 0.5000 | 0.0000 |
| 0.3750 | 0.8750 | 0.2500 | 0.7500 |
| 0.5625 | 0.0625 | 0.6875 | 0.1875 |
| 0.3125 | 0.8125 | 0.4375 | 0.9375 |

| | | | |
|---|---|---|---|
| 0.9375 | 0.4375 | 0.8125 | 0.3125 |
| 0.1875 | 0.6875 | 0.0625 | 0.5625 |
| 0.7500 | 0.2500 | 0.8750 | 0.3750 |
| 0.0000 | 0.5000 | 0.1250 | 0.6250 |

4 X 4 X 3 dither mask
(`DITHER_MASK_443`)

**Figure 6-1 Ordered Dither Masks**

#### 6.6.1.3 OrderedDither Example

Listing 6-10 shows a partial code sample of using the `OrderedDither` operation.

**Listing 6-10 Ordered Dither Example**

```
    // Create the color cube.
    ColorCube colorMap =
        srcRescale.getSampleModel().getTransferType() ==
                   DataBuffer.TYPE_BYTE ?
        ColorCube.BYTE_496 :
       ColorCube.createColorCube(dataType, 38, new int[] {4, 9, 6});

    // Set the dither mask to the pre-defined 4x4x3 mask.
    KernelJAI[] ditherMask = KernelJAI.DITHER_MASK_443;

    // Create a new ParameterBlock.
    ParameterBlock pb = new ParameterBlock();
    pb.addSource(srcRescale).add(colorMap).add(ditherMask);

    // Create a gray scale color model.
    ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);
    int bits[] = new int[] {8};
    ColorModel cm = new ComponentColorModel(cs, bits, false, false,
                                            Transparency.OPAQUE,
                                            DataBuffer.TYPE_BYTE);

    // Create a tiled layout with the requested ColorModel.
    layout = new ImageLayout();
    layout.setTileWidth(TILE_WIDTH).setTileHeight(TILE_HEIGHT);
    layout.setColorModel(cm);

    // Create RenderingHints for the ImageLayout.
    rh = new RenderingHints(JAI.KEY_IMAGE_LAYOUT, layout);

    // Create the ordered dither OpImage.
    PlanarImage image = (PlanarImage)JAI.create("ordereddither",
                                                pb, rh);
```

### 6.6.2 Error-diffusion Dither

The error-diffusion dithering operation produces the most accurate destination image, but is more complex and thus takes longer than the ordered dither.

The `ErrorDiffusion` operation performs color quantization by finding the nearest color to each pixel in a supplied lookup table, called a color map, and “diffusing” the color quantization error below and to the right of the pixel.

The source image and the color map must have the same data type and number of bands. Also, the color map must have the same offset in all bands. The resulting image is single-banded.

The `ErrorDiffusion` operation takes one rendered source image and two parameters:

| Parameter | Type | Description |
|---|---|---|
| `colorMap` | `LookupTableJAI` | The color map. A `LookupTableJAI` (see Section 7.6.1, "Creating the Lookup Table") or a `ColorCube` (see Section 6.6.1.1, "Color Map Parameter"). |
| `errorKernel` | `KernelJAI` | The error filter kernel. See Section 6.6.2.1, "Error Filter Kernel." |

### 6.6.2.1 Error Filter Kernel

The `errorKernel` parameter can be one of three predefined error filters or you can create your own. To create your own, see Section 6.9, "Constructing a Kernel."

The predefined kernels are (see Figure 6-2):

| errorKernel | Description |
|---|---|
| `ERROR_FILTER_FLOYD_STEINBERG` | Based on the Floyd-Steinberg filter model (the default if none is specified). |
| `ERROR_FILTER_JARVIS` | Based on the Jarvis-Judice-Ninke filter model. |
| `ERROR_FILTER_STUCKI` | Based on the Stucki filter model |

The error filter kernel, also known as the *error distribution filter*, diffuses the color quantization error below and to the right of the pixel. The elements of the error filter kernel that are in the same row and to the right of the key element or are in a row below that of the key element must be between 0.0 and 1.0 and must sum to approximately 1.0. The other elements of the error filter kernel are ignored.

In operation, the filter is laid on top of the source image so that its origin aligns with the pixel to be passed through the lookup table. Figure 6-3 shows an example using the Floyd-Steinberg filter. The diffusion operation then:

- Sets the pixel at 0,2 to 214 + (5 × [7/16])
- Sets the pixel at 1,0 to 128 + (5 × [3/16])
- Sets the pixel at 1,1 to 255 + (5 × [5/16])
- Sets the pixel at 1,2 to 104 + (5 × [1/16])

The filter is then moved to the next pixel and the process is repeated. The result of this process is an averaging that produces a smoother dithered image with little or no contouring.

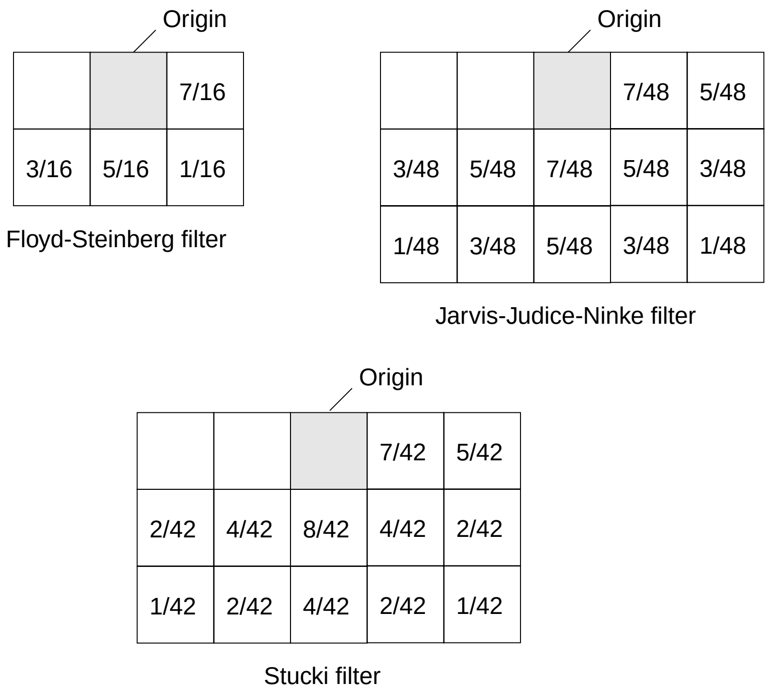


**Figure 6-2 Error Diffusion Dither Filters**

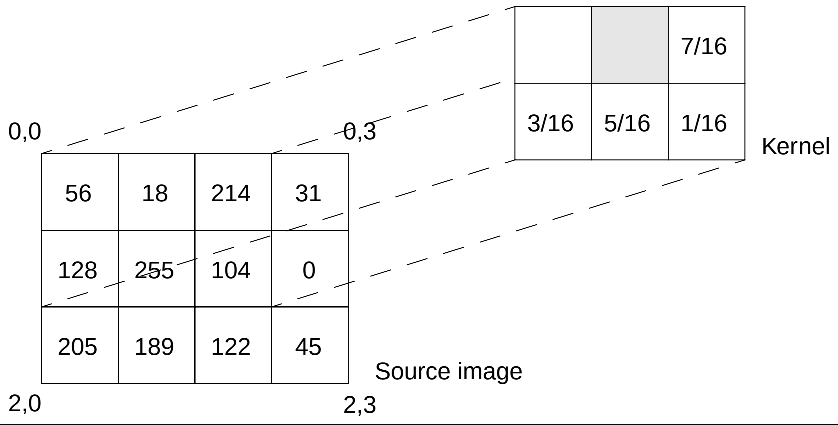


**Figure 6-3 Error Diffusion Operation**

#### 6.6.2.2 ErrorDiffusion Example

Listing 6-11 shows a partial code sample of using the `ErrorDiffusion` operation.

**Listing 6-11 Error Diffusion Example**

```
// Create a color map with the 4-9-6 color cube and the
// Floyd-Steinberg error kernel.
ParameterBlock pb;
pb.addSource(src);
pb.add(ColorCube.BYTE_496);
pb.add(KernelJAI.ERROR_FILTER_FLOYD_STEINBERG);

// Perform the error diffusion operation.
dst = (PlanarImage)JAI.create("errordiffusion", pb, null);
```

## 6.7 Clamping Pixel Values

The `clamp` operation restricts the range of pixel values for a source image by constraining the range of pixels to defined “low” and “high” values. The operation takes one rendered or renderable source image, and sets all the pixels whose value is below a low value to that low value and all the pixels whose value is above a high value to that high value. The pixels whose value is between the low value and the high value are left unchanged.

A different set of low and high values may be applied to each band of the source image, or the same set of low and high values may be applied to all bands of the source. If the number of low and high values supplied is less than the number of bands of the source, the values from entry 0 are applied to all the bands. Each low value must be less than or equal to its corresponding high value.

The pixel values of the destination image are defined by the following pseudocode:

```
lowVal = (low.length < dstNumBands) ?
         low[0] : low[b];
highVal = (high.length < dstNumBands) ?
          high[0] : high[b];

if (src[x][y][b] < lowVal) {
    dst[x][y][b] = lowVal;
} else if (src[x][y][b] > highVal) {
    dst[x][y][b] = highVal;
} else {
    dst[x][y][b] = src[x][y][b];
    }
```

The `clamp` operation takes one rendered or renderable source image and two parameters:

| Parameter | Type | Description |
|---|---|---|
| `low` | `Double` | The lower boundary for each band. |
| `high` | `Double` | The upper boundary for each band. |

Listing 6-12 shows a partial code sample of using the `Clamp` operation to clamp pixels values to between 5 and 250.

**Listing 6-12 Clamp Operation**

```
    // Get the source image width, height, and SampleModel.
    int w = src.getWidth();
    int h = src.getHeight();
    int b = src.getSampleModel().getNumBands();

    // Set the low and high clamp values.
    double[] low, high;

    low  = new double[b];
    high = new double[b];

    for (int i=0; i<b; i++) {
         low[i]  = 5;              // The low clamp value
         high[i] = 250;            // The high clamp value
    }

    // Create the ParameterBlock with the source and parameters.
    pb = new ParameterBlock();
    pb.addSource(src);
    pb.add(low);
    pb.add(high);

    // Perform the operation.
    RenderedImage dst = JAI.create("clamp", pb);
```

## 6.8 Band Copying

The `BandSelect` operation chooses *N* bands from a rendered or renderable source image and copies the pixel data of these bands to the destination image in the order specified. The `bandIndices` parameter specifies the source band indices, and its size (*bandIndices.length*) determines the number of bands of the destination image. The destination image may have ay number of bands, and a particular band of the source image may be repeated in the destination image by specifying it multiple times in the `bandIndices` parameter.

Each of the `bandIndices` value should be a valid band index number of the source image. For example, if the source only has two bands, 1 is a valid band index, but 3 is not. The first band is numbered 0.

The destination pixel values are defined by the following pseudocode:

```
dst[x][y][b] = src[x][y][bandIndices[b]];
```

The `bandselect` operation takes one rendered or renderable source image and one parameter:

| **Parameter** | **Type** | **Description** |
|---|---|---|
| `bandIndices` | `int[]` | The indices of the selected bands of the image. |

Listing 6-13 shows a partial code sample of using the `BandSelect` operation.

**Listing 6-13 BandSelect Operation**

```
// Set the indices of three bands of the image.
int[] bandIndices;
bandIndices = new int[3];
bandIndices[0] = 0;
bandIndices[1] = 2;
bandIndices[2] = 2;

// Construct the ParameterBlock.
pb = new ParameterBlock();
pb.addSource(src);
pb.add(bandIndices);

// Perform the operation
RenderedImage dst = (RenderedImage)JAI.create("bandSelect",
                                               pb);
```

## 6.9 Constructing a Kernel

The `KernelJAI` class is an auxiliary class used with the convolve, ordered dither, error diffusion dither, dilate, and erode operations. A `KernelJAI` is characterized by its width, height, and key element (origin) position. The key element is the element that is placed over the current source pixel to perform convolution or error diffusion.

For the `OrderedDither` operation (see Section 6.6.1, “Ordered Dither”), an array of `KernelJAI` objects is actually required with there being one `KernelJAI` per band of the image to be dithered. The location of the key element is in fact irrelevant to the `OrderedDither` operation.

There are four constructors for creating a `KernelJAI`. The following constructor constructs a `KernelJAI` object with the given parameters.

```
KernelJAI(int width, int height, float[] data)
```

The `width` and `height` parameters determine the kernel size. The `data` parameter is a pointer to the floating point values stored in a data array. The key element is set to

$$\text{trunc}\left(\frac{\text{width}}{2}\right), \text{trunc}\left(\frac{\text{height}}{2}\right)$$

The following constructor constructs a `KernelJAI` object with the given parameters.

```
KernelJAI(int width, int height, int xOrigin, int yOrigin,
             float[] data)
```

The `xOrigin` and `yOrigin` parameters determine the key element’s origin.

The following constructor constructs a separable `KernelJAI` object from two float arrays.

```
KernelJAI(int width, int height, int xOrigin, int yOrigin,
             float[] dataH, float[] dataV)
```

The `dataH` and `dataV` parameters specify the float data for the horizontal and vertical directions, respectively.

The following constructor constructs a `KernelJAI` object from a `java.awt.image.Kernel` object.

```
KernelJAI(java.awt.image.Kernel k)
```

Listing 6-14 shows a partial code sample for creating a simple $3 \times 3$ kernel with the key element located at coordinates 1,1, as shown in Figure 6-4.

**Listing 6-14 Constructing a KernelJAI**

```
kernel = new KernelJAI;
float[] kernelData = {
    0.0F,        1.0F,        0.0F,
    1.0F,        1.0F,        1.0F,
    0.0F,        1.0F,        0.0F
};
kernel = new KernelJAI(3, 3, 1, 1, kernelData);
```

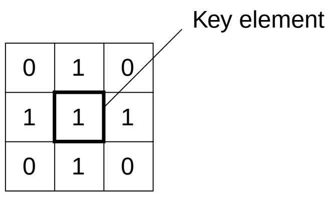


**Figure 6-4 Example Kernel**

The Java Advanced Imaging API provides a shorthand method for creating several commonly-used kernels, listed in Table 6-2, which can simply be called by name. These kernels and their use are described in more detail in Section 6.6.1, “Ordered Dither,” Section 6.6.2, “Error-diffusion Dither,” and Section 9.5, “Edge Detection.”

**Table 6-2 Named Kernels**

| Kernel Name | Description and Use |
|---|---|
| `DITHER_MASK_441` | Ordered dither filter. A 4 × 4 × 1 mask useful for dithering 8-bit grayscale images to 1-bit images |
| `DITHER_MASK_443` | Ordered dither filter. A 4 × 4 × 3 mask useful for dithering 24-bit color images to 8-bit pseudocolor images. |
| `ERROR_FILTER_FLOYD_STEINBERG` | Error diffusion filter, based on the Floyd-Steinberg model. |
| `ERROR_FILTER_JARVIS` | Error diffusion filter, based on the Jarvis-Judice-Ninke model. |
| `ERROR_FILTER_STUCKI` | Error diffusion filter, based on the Stucki model |
| `GRADIENT_MASK_SOBEL_HORIZONTAL` | The horizontal gradient filter mask for the `Gradient` operation. |
| `GRADIENT_MASK_SOBEL_VERTICAL` | The vertical gradient filter mask for the `Gradient` operation. |

The following code sample shows the format for creating a named kernel:

```
KernelJAI kernel = KernelJAI.ERROR_FILTER_FLOYD_STEINBERG;
```

**API:** `javax.media.jai.KernelJAI`

-
  ```
  public KernelJAI(int width, int height, int xOrigin,
      int yOrigin, float[] data)
  ```
  constructs a `KernelJAI` with the given parameters. The data array is copied.

| | | |
|---|---|---|
| *Parameters*: | `width` | The width of the kernel. |
| | `height` | The height of the kernel |
| | `xOrigin` | The *x* coordinate of the key kernel element. |
| | `yOrigin` | The *y* coordinate of the key kernel element. |
| | `data` | The float data in row-major format. |

-
  ```
  public KernelJAI(int width, int height, int xOrigin,
      int yOrigin, float[] dataH, float[] dataV)
  ```
  constructs a separable `KernelJAI` from two float arrays. The data arrays are copied.

| | | |
|---|---|---|
| *Parameters*: | `dataH` | The float data for the horizontal direction. |
| | `dataV` | The float data for the vertical direction. |

-
  ```
  public KernelJAI(int width, int height, float[] data)
  ```
  constructs a `KernelJAI` with the given parameters. The data array is copied. The key element is set to (trunc(width/2), trunc(height/2)).

| | | |
|---|---|---|
| *Parameters*: | `data` | The float data in row-major format. |

-
  ```
  public KernelJAI(Kernel k)
  ```
  constructs a `KernelJAI` from a `java.awt.image.Kernel` object.

CHAPTER 7

# Image Enhancement

THIS chapter describes the basics of improving the visual appearance of images through enhancement operations.

## 7.1 Introduction

The JAI API image enhancement operations include:

- Adding borders
- Cropping an image
- Amplitude rescaling
- Histogram equalization
- Lookup table modification
- Convolution filtering
- Median filtering
- Frequency domain processing
- Pixel point processing
- Thresholding (binary contrast enhancement)

## 7.2 Adding Borders to Images

JAI provides two different ways of adding a border to an image. These two ways are described in the following paragraphs.

### 7.2.1 The Border Operation

The `Border` operation allows you to add a simple filled border around a source image. The border extends the source image's boundaries by a specified number of pixels.The amount of extension may be specified separately for the top, bottom, and left and right sides. The following types of border fill may be specified:

- Zero fill – the border area is extended with zeros (`BORDER_ZERO_FILL`).
- Constant fill – the border area is extended with a specified constant value (`BORDER_CONST_FILL`). An array of constants must be supplied. The array must have at least one element, in which case this same constant is applied to all destination image bands. Or, it may have a different constant entry for each corresponding band. For all other border types, this `constants` parameter may be `null`.
- Extend – the border area is created by copying the edge and corner pixels (`BORDER_COPY`).
- Reflection – the border area is created by reflection of the image's outer edge (`BORDER_REFLECT`).
- Wrap – the border area is extended by "wrapping" the image plane toroidally, that is, joining opposite edges of the image (`BORDER_WRAP`).

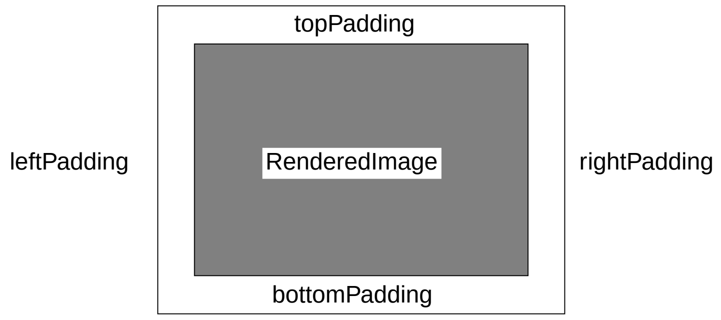


**Figure 7-1 Image Borders**

The image layout (tile width, height, and offsets; `SampleModel` and `ColorModel`) is copied from the source. The `Border` operation takes one rendered source image and six parameters:

| Parameters | Type | Description |
|---|---|---|
| leftPad | Integer | The image's left padding. |
| rightPad | Integer | The image's right padding. |
| topPad | Integer | The image's top padding. |
| bottomPad | Integer | The image's bottom padding. |
| type | Integer | The border type. One of BORDER_ZERO, BORDER_CONST_FILL, BORDER_COPY, BORDER_REFLECT, or BORDER_WRAP. The default is BORDER_ZERO. |
| constant | double | The constants used by the BORDER_CONST_FILL. |

## 7.2.2 Extending the Edge of an Image

Some area operations, such as convolve, scale, and rotate, benefit from the addition of an extended border around the source image. The extended border comes into play when the convolution kernel overlaps the source image as the key value is scanned over it.

A BorderExtender may be applied to an operation using a suitable hint. The hints are defined in Table 7-1.

**Table 7-1 BorderExtender Hints**

| Name | Description |
|---|---|
| BorderExtenderZero | Extends an image's border by filling all pixels outside the image bounds with zeros. See Section 7.2.2.1, "BorderExtenderZero." |
| BorderExtenderConstant | Extends an image's border by filling all pixels outside the image bounds with constant values. See Section 7.2.2.2, "BorderExtenderConstant." |
| BorderExtenderCopy | Extends an image's border by filling all pixels outside the image bounds with copies of the edge pixels. Useful as a way of padding source images prior to area or geometric operations, such as convolution, scaling, or rotation. See Section 7.2.2.3, "BorderExtenderCopy." |
| BorderExtenderWrap | Extends an image's border by filling all pixels outside the image bounds with copies of the whole image. This form of extension is appropriate for data that is inherently periodic, such as the Fourier transform of an image, or a wallpaper pattern. See Section 7.2.2.4, "BorderExtenderWrap." |
| BorderExtenderReflect | Extends an image's border by filling all pixels outside the image bounds with copies of the whole image. This form of extension avoids discontinuities around the edges of the image. See Section 7.2.2.5, "BorderExtenderReflect." |

The BorderExtender class is the superclass for four classes that extend a WritableRaster with additional pixel data taken from a PlanarImage. Instances of `BorderExtender` are used by the `PlanarImage.getExtendedData` and `PlanarImage.copyExtendedData` methods.

The `PlanarImage.getExtendedData` method returns a copy of an arbitrary rectangular region of the image in a `Raster`. The portion of the rectangle of interest outside the bounds of the image will be computed by calling the given `BorderExtender`. If the region falls entirely within the image, the extender will not be used. Thus it is possible to use a `null` value for the `extender` parameter when it is known that no actual extension will be required. The returned `Raster` should be considered non-writable. The `copyExtendedData` method should be used if the returned `Raster` is to be modified.

The `PlanarImage.copyExtendedData` method copies an arbitrary rectangular region of the `RenderedImage` into a caller-supplied `WritableRaster`. The portion of the supplied `WritableRaster` that lies outside the bounds of the image is computed by calling the given `BorderExtender`. The supplied `WritableRaster` must have a `SampleModel` that is compatible with that of the image.

Each instance of `BorderExtender` has an `extend` method that takes a `WritableRaster` and a `PlanarImage`. The portion of the raster that intersects the bounds of the image will already contain a copy of the image data. The remaining area is to be filled in according to the policy of the `BorderImage` subclass. The subclasses are described in Table 7-1.

**API:** `javax.media.jai.Planarimage`

-
  ```
  Raster getExtendedData(Rectangle region,
      BorderExtender extender)
  ```
  returns a copy of an arbitrary rectangular region of this image in a Raster.

| | | |
|---|---|---|
| *Parameters*: | `region` | The region of the image to be returned. |
| | `extender` | An instance of `BorderExtender`, used only if the region exceeds the image bounds. |

- `void copyExtendedData(WritableRaster dest,`
  `BorderExtender extender)`

  copies an arbitrary rectangular region of the `RenderedImage` into a caller-supplied `WritableRaster`.

| *Parameters*: | | |
|---|---|---|
| | `dest` | A `WritableRaster` to hold the returned portion of the image. |
| | `extender` | An instance of `BorderExtender`. |

**API:** `javax.media.jai.BorderExtender`

- `static BorderExtender createInstance(int extenderType)`

  returns an instance of `BorderExtender` that implements a given extension policy. The policies understood by this method are:

| **Policy** | **Description** |
|---|---|
| `BORDER_ZERO` | Set sample values to zero. |
| `BORDER_COPY` | Set sample values to copies of the nearest valid pixel. For example, pixels to the left of the valid rectangle will take on the value of the valid edge pixel in the same row. Pixels both above and to the left of the valid rectangle will take on the value of the upper-left pixel. |
| `BORDER_REFLECT` | The output image is defined as if mirrors were placed along the edges of the source image. Thus if the left edge of the valid rectangle lies at $x = 10$, pixel $(9, y)$ will be a copy of pixel $(10, y)$; pixel $(6, y)$ will be a copy of pixel $(13, y)$. |
| `BORDER_WRAP` | The source image is tiled repeatedly in the plane. |

- `abstract void extend(WritableRaster raster, PlanarImage im)`

  fills in the portions of a given `Raster` that lie outside the bounds of a given `PlanarImage` with data derived from that `PlanarImage`.

#### 7.2.2.1 BorderExtenderZero

The `BorderExtenderZero` class is a subclass of `BorderExtender` that implements border extension by filling all pixels outside of the image bounds with zeros. For example, Figure 7-2 shows the result of using `BorderExtenderZero` to extend an image by adding two extra rows to the top and bottom and two extra columns on the left and right sides.

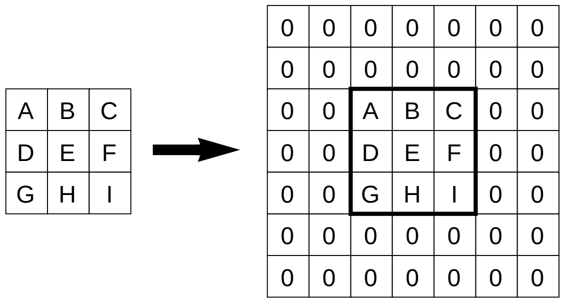


**Figure 7-2 BorderExtenderZero Example**

**API:** `javax.media.jai.BorderExtenderZero`

- `final void extend(WritableRaster raster, PlanarImage im)`

  fills in the portions of a given `Raster` that lie outside the bounds of a given `PlanarImage` with zeros. The portion of Raster that lies within `im.getBounds` is not altered.

#### 7.2.2.2 BorderExtenderConstant

The `BorderExtenderConstant` class is a subclass of `BorderExtender` that implements border extension by filling all pixels outside of the image bounds with constant values. For example, Figure 7-3 shows the result of using `BorderExtenderConstant` to extend an image by adding two extra rows to the top and bottom and two extra columns on the left and right sides.

In the figure, `X` is the constant fill value. The set of constants is clamped to the range and precision of the data type of the `Raster` being filled. The number of constants used is given by the number of bands of the `Raster`. If the `Raster` has $b$ bands, and there are $c$ constants, constants 0 through $b - 1$ are used when $b \leq c$. If $b > c$, zeros are used to fill out the constants array.

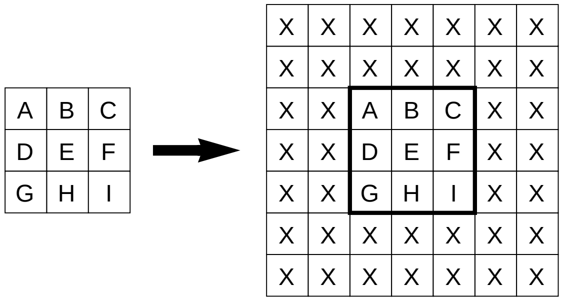


**Figure 7-3 BorderExtenderConstant Example**

**API:** `javax.media.jai.BorderExtenderConstant`

- `BorderExtenderConstant(double[] constants)`

  constructs an instance of `BorderExtenderConstant` with a given set of constants. The constants are specified as an array of `double`s.

- `final void extend(WritableRaster raster, PlanarImage im)`

  fills in the portions of a given `Raster` that lie outside the bounds of a given `PlanarImage` with constant values. The portion of `Raster` that lies within `im.getBounds` is not altered.

### 7.2.2.3 BorderExtenderCopy

The `BorderExtenderCopy` class is a subclass of `BorderExtender` that implements border extension by filling all pixels outside of the image bounds with copies of the edge pixels. For example, Figure 7-4 shows the result of using `BorderExtenderCopy` to extend an image by adding two extra rows to the top and bottom and two extra columns on the left and right sides.

Although this type of extension is not particularly visually appealing, it is useful as a way of padding source images prior to area or geometric operations, such as convolution, scaling, or rotation.

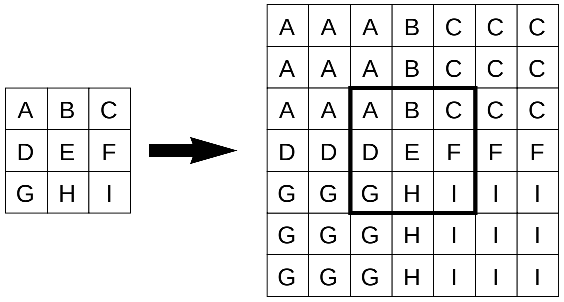


**Figure 7-4 BorderExtenderCopy Example**

**API:** `javax.media.jai.BorderExtenderCopy`

- `final void extend(WritableRaster raster, PlanarImage im)`

  fills in the portions of a given `Raster` that lie outside the bounds of a given `PlanarImage` with copies of the edge pixels of the image. The portion of `Raster` that lies within `im.getBounds` is not altered.

#### 7.2.2.4 BorderExtenderWrap

The `BorderExtenderWrap` class is a subclass of `BorderExtender` that implements border extension by filling all pixels outside of the image bounds with copies of the whole image. For example, Figure 7-5 shows the result of using `BorderExtenderWrap` to extend an image by adding two extra rows to the top and bottom and two extra columns on the left and right sides.

This form of extension is appropriate for data that is inherently periodic, such as the Fourier transform of an image or a wallpaper pattern.

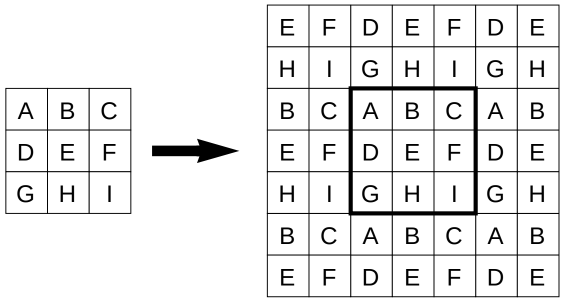


**Figure 7-5 BorderExtenderWrap Example**

**API:** `javax.media.jai.BorderExtenderWrap`

- `final void extend(WritableRaster raster, PlanarImage im)`

  Fills in the portions of a given `Raster` that lie outside the bounds of a given `PlanarImage` with copies of the entire image. The portion of `Raster` that lies within `im.getBounds` is not altered.

#### 7.2.2.5 BorderExtenderReflect

The `BorderExtenderReflect` class is a subclass of `BorderExtender` that implements border extension by filling all pixels outside the image bounds with reflected copies of the whole image. For example, Figure 7-6 shows the result of using `BorderExtenderReflect` to extend an image by adding two extra rows to the top and bottom and one extra column on the left and right sides.

This form of extension avoids discontinuities around the edges of the image.

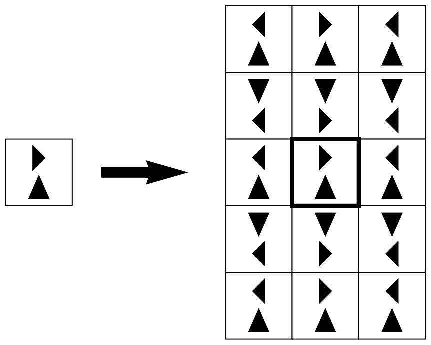


**Figure 7-6 BorderExtenderReflect Example**

**API:** `javax.media.jai.BorderExtenderReflect`

- `final void extend(WritableRaster raster, PlanarImage im)`

  Fills in the portions of a given `Raster` that lie outside the bounds of a given `PlanarImage` with suitably reflected copies of the entire image. The portion of `Raster` that lies within `im.getBounds` is not altered.

## 7.3 Cropping an Image

The `Crop` operation crops a rendered or renderable image to a specified rectangular area. The *x*, *y*, width, and height values are clipped to the source image's bounding box. These values are rounded to type `int` for rendered images.

The `Crop` operation takes one rendered or renderable source image and four parameters. None of the parameters have default values; all must be supplied.

| Parameter | Type | Description |
|---|---|---|
| `x` | `Float` | The *x* origin for each band. |
| `y` | `Float` | The *y* origin for each band. |
| `width` | `Float` | The width for each band. |
| `height` | `Float` | The height for each band. |

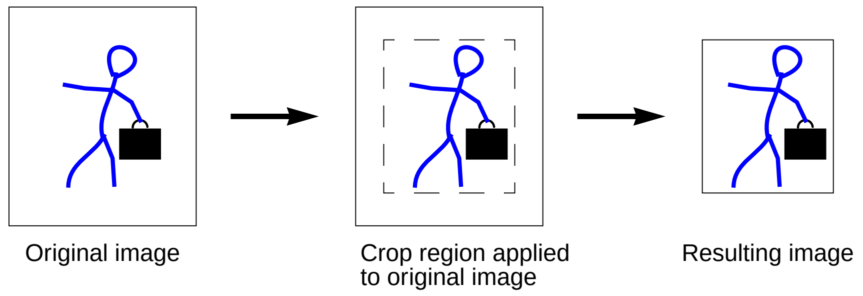


**Figure 7-7 Crop Operation**

## 7.4 Amplitude Rescaling

Amplitude rescaling provides a linear amplitude transformation of input pixel values to output pixel values. Amplitude rescaling can be used to enhance images that have insufficient contrast between the lightest and darkest values, such as caused by underexposure or overexposure of the original image.

The full dynamic range of one band of an eight-bit image is 0 to 255. An underexposed image may only contain pixel values from 10 to 180, resulting in an image that does not fully use the dynamic range of the display. Such an image can be greatly improved by linearly stretching the contrast range; mapping the lowest values to 0 and the highest values to 255.

The `rescale` operation takes a rendered or renderable source image and maps the pixel values of the image from one range to another range by multiplying each pixel value by one of a set of constants and then adding another constant to the result of the multiplication. If the number of constants supplied is less than the number of bands of the destination, the constant from entry 0 is applied to all the bands. Otherwise, a constant from a different entry is applied to each band. There must be at least one entry in each of the constants and offsets arrays.

The pixel values of the destination image are defined by the following pseudocode:

```
constant = (constants.length < dstNumBands) ?
              constants[0] : constants[b];
offset = (offsets.length < dstNumBands) ?
            offsets[0] : offsets[b];

dst[x][y][b] = src[x][y][b]*constant + offset;
```

The pixel arithmetic is performed using the data type of the destination image. By default, the destination will have the same data type as the source image unless an `ImageLayout` containing a `SampleModel` with a different data type is supplied as a rendering hint.

The values of the lowest and highest pixel amplitudes must be known. This information can be acquired through the `Extrema` operation (see Section 9.3, "Finding the Extrema of an Image").

The following equations show the relationships between the extrema and the scale and offset factors.

$$\text{scale}(b) = \frac{255}{max(b) - min(b)} \tag{7.1}$$

$$\text{offset}(b) = \frac{255 \times min(b)}{min(b) - max(b)} \tag{7.2}$$

where *max*(*b*) and *min*(*b*) are the largest and smallest pixel values in the band, respectively.

The `rescale` operation takes one rendered or renderable source image and two parameters:

| **Parameter** | **Type** | **Description** |
|---|---|---|
| `constants` | `double` | The per-band constants to multiply by. |
| `offsets` | `double` | The per-band offsets to be added. |

## 7.5 Histogram Equalization

An image histogram is an analytic tool used to measure the amplitude distribution of pixels within an image. For example, a histogram can be used to provide a count of the number of pixels at amplitude 0, the number at amplitude 1, and so on. By analyzing the distribution of pixel amplitudes, you can gain some information about the visual appearance of an image. A high-contrast image contains a wide distribution of pixel counts covering the entire amplitude range. A low contrast image has most of the pixel amplitudes congregated in a relatively narrow range.

See Section 9.4, "Histogram Generation," for information on how to generate a histogram for an image. The next two sections describe JAI operations that use an image histogram to enhance an image's appearance.

### 7.5.1 Piecewise Linear Mapping

The `Piecewise` operation performs a piecewise linear mapping of an image's pixel values. The piecewise linear mapping is described by a set of breakpoints that are provided as an array of the form:

```
float breakPoints[N][2][numBreakPoints]
```

where the value of *N* may be either unity or the number of bands in the source image.

If *N* is unity, the same set of breakpoints will be applied to all bands in the image. The abscissas of the supplied breakpoints must be monotonically increasing.

The pixel values of the destination image are defined by the following pseudocode:

```
if(src[x][y][b] < breakPoints[b][0][0])
    dst[x][y][b] = breakPoints[b][1][0]);
} else if(src[x][y][b] > breakPoints[b][0][numBreakPoints-1]) {
    dst[x][y][b] = breakPoints[b][1][numBreakPoints-1]);
} else {
    int i = 0;
    while(breakPoints[b][0][i+1] < src[x][y][b]) {
        i++;
    }
    dst[x][y][b] = breakPoints[b][1][i] +
                        (src[x][y][b] - breakPoints[b][0][i])*
                   (breakPoints[b][1][i+1] - breakPoints[b][1][i])/
                   (breakPoints[b][0][i+1] - breakPoints[b][0][i]);
```

The Piecewise operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| breakPoints | Float | The breakpoint array. |

Listing 7-1 shows a code sample of a Piecewise operation, showing only the construction of the piecewise-mapped image and the operation. The generation of the source image, fmt, is not shown.

**Listing 7-1 Example Piecewise Operation**

```
// Create a piecewise-mapped image emphasizing low values.
float[][][] bp = new float[numBands][2][];
for(int b = 0; b < numBands; b++) {
    bp[b][0] = new float[] {0.0F, 32.0F, 64.0F, 255.0F};
    bp[b][1] = new float[] {0.0F, 64.0F, 112.0F, 255.0F};
}

// Create the Piecewise operation.
RenderedOp pw = JAI.create("piecewise", fmt, bp);
```

## 7.5.2 Histogram Matching

It is sometimes desirable to transform an image so that its histogram matches that of a specified functional form. The MatchCDF operation performs a piecewise linear mapping of the pixel values of an image such that the cumulative distribution function (CDF) of the destination image matches as closely as possible a specified cumulative distribution function.

The CDF of an image is its area-normalized threshold area function. The desired CDF for the MatchCDF operation is described by an array of the form:

```
float CDF[numBands][numBins[b]]
```

where numBins denotes the number of bins in the histogram of the source image for band *b*.

Each element in the array CDF[b] must be non-negative, the array must represent a non-decreasing sequence, and the last element of the array must be 1.0F. The source image must have a Histogram object available via its getProperty method.

The `MatchCDF` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `CDF` | `Float` | The desired cumulative distribution function. |

The operation requires that the image histogram be available.

Listing 7-2 shows a code sample of a `MatchCDF` operation, showing only the histogram operation, construction of two different CDFs, and the operations that use them.

**Listing 7-2 Example MatchCDF Operation**

```
// Retrieves a histogram for the image.
private static Histogram getHistogram(RenderedOp img,
                                      int binCount) {

    // Get the band count.
    int numBands = img.getSampleModel().getNumBands();

    // Allocate histogram memory.
    int[] numBins = new int[numBands];
    double[] lowValue = new double[numBands];
    double[] highValue = new double[numBands];
    for(int i = 0; i < numBands; i++) {
        numBins[i] = binCount;
        lowValue[i] = 0.0;
        highValue[i] = 255.0;
    }

    // Create the Histogram object.
   Histogram hist = new Histogram(numBins, lowValue, highValue);

    // Set the ROI to the entire image.
    ROIShape roi = new ROIShape(img.getBounds());

    // Create the histogram op.
    RenderedOp histImage =
        JAI.create("histogram", img,
                   hist, roi, new Integer(1), new Integer(1));

    // Retrieve the histogram.
    hist = (Histogram)histImage.getProperty("histogram");

    return hist;
}
```

**Listing 7-2 Example MatchCDF Operation (Continued)**

```
// Create an equalization CDF.
float[][] CDFeq = new float[numBands][];
for(int b = 0; b < numBands; b++) {
    CDFeq[b] = new float[binCount];
    for(int i = 0; i < binCount; i++) {
        CDFeq[b][i] = (float)(i+1)/(float)binCount;
    }
}

// Create a normalization CDF.
double[] mean = new double[] {128.0, 128.0, 128.0};
double[] stDev = new double[] {64.0, 64.0, 64.0};
float[][] CDFnorm = new float[numBands][];
for(int b = 0; b < numBands; b++) {
    CDFnorm[b] = new float[binCount];
    double mu = mean[b];
    double twoSigmaSquared = 2.0*stDev[b]*stDev[b];
    CDFnorm[b][0] =
        (float)Math.exp(-mu*mu/twoSigmaSquared);
    for(int i = 1; i < binCount; i++) {
        double deviation = i - mu;
        CDFnorm[b][i] = CDFnorm[b][i-1] +
         (float)Math.exp(-deviation*deviation/twoSigmaSquared);
    }
}
for(int b = 0; b < numBands; b++) {
    double CDFnormLast = CDFnorm[b][binCount-1];
   for(int i = 0; i < binCount; i++) {
       CDFnorm[b][i] /= CDFnormLast;
   }
}

// Create a histogram-equalized image.
RenderedOp eq = JAI.create("matchcdf", fmt, CDFeq);

// Create a histogram-normalized image.
RenderedOp nm = JAI.create("matchcdf", fmt, CDFnorm);
```

## 7.6 Lookup Table Modification

The lookup table modification provides a non-linear amplitude transformation. Non-linear amplitude transformation is useful if you have a non-linear amplitude response difference between the sensor that captures the image data and the display.

The lookup table modification mechanism allows you to arbitrarily convert between the source image byte, short, or integer pixel value and one or more output values. The output value can be a byte, short, integer, float, or double image pixel.

The input pixel value acts as an address to the lookup table inputs, as shown in Figure 7-8. Each location in the lookup table stores the desired output value for that particular address.

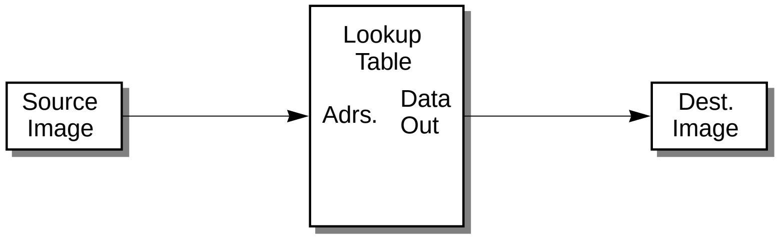


**Figure 7-8 Lookup Table**

The lookup table is first loaded with the necessary data. Table 7-2 shows a partial listing of an example lookup table. In this example, the input values range from 0 to 255. The output values provide a scaled square root transformation between the input and output, according to the following equation:

$$\text{output} = \sqrt{255 \times \text{input}}$$

**Table 7-2 Example Lookup Table**

| Input | Output |
|---|---|
| 0 | 0 |
| 1 | 16 |
| 2 | 23 |
| 3 | 28 |
| . | . |
| 253 | 254 |
| 254 | 255 |
| 255 | 255 |

This example provides a non-linear amplitude transformation between input and output pixel values, in which the smaller input amplitude values are amplified and the larger input values are attenuated. Other types of lookup values can be used to solve nearly any non-linear amplitude scaling problem.

## 7.6.1 Creating the Lookup Table

The `LookupTableJAI` object represents a single- or multi-banded table or a color cube of any supported data types. A single- or multi-banded source image of integer data types is passed through the table and transformed into a single- or multi-banded destination image of both integral and float or double data types.

The `LookupTableJAI` object is used for the `ErrorDiffusion` operation, where it describes a color map, and the `Lookup` operation, where it describes the lookup table. For the `Lookup` operation, the table data may cover only a subrange of the legal range of the input data type. The subrange is selected by means of an offset parameter that is to be subtracted from the input value before indexing into the table array.

The procedures for constructing a lookup table vary slightly, depending on whether the input image is single-banded or multi-banded. For a single-band input image, you construct a single lookup table. For a multi-band image, you construct a single lookup table with entries for each band.

### 7.6.1.1 Creating a Single-band Lookup Table

The single-banded lookup table contains data for a single channel or image component. To create a lookup table for a single-band input image, use one of the single-band constructors. The constructors take up to three parameters:

- A pointer to the data to be stored in the table. The data may be of type `Byte`, `Short`, `UShort`, `Int`, `Float`, or `Double`.
- The offset. The offset selects the lookup table subrange. The offset value is subtracted from the input value before indexing into the table array.
- A boolean flag that indicates whether Short data is of type Short or UShort.

Listing 7-3 shows an example of the construction of a single-band byte lookup table.

**Listing 7-3 Example Single-band Lookup Table**

```
byte[] tableData = new byte[0x10000];
for (int i = 0; i < 0x10000; i++) {
tableData[i] = (byte)(i >> 8);
}

// Create a LookupTableJAI object to be used with the
// "lookup" operator.
LookupTableJAI table = new LookupTableJAI(tableData);
```

**API:** `javax.media.jai.LookupTableJAI`

- `LookupTableJAI(byte[] data)`

  constructs a single-banded byte lookup table with an index offset of 0.

| | | |
|---|---|---|
| *Parameters*: | `data` | The single-banded byte data |

- `LookupTableJAI(byte[] data, int offset)`

  constructs a single-banded byte lookup table with an index offset.

| | | |
|---|---|---|
| *Parameters*: | `data` | The single-banded byte data |
| | `offset` | The offset |

- `LookupTableJAI(short[] data, boolean isUShort)`

  constructs a single-banded short or unsigned short lookup table with an index offset of 0.

| | | |
|---|---|---|
| *Parameters*: | `data` | The single-banded short data |
| | `isUShort` | True if the data type is `DataBuffer.TYPE_USHORT`; false if the data type is `DataBuffer.TYPE_SHORT`. |

- `LookupTableJAI(short[] data, int offset, boolean isUShort)`

  constructs a single-banded short or unsigned short lookup table with an index offset.

| | | |
|---|---|---|
| *Parameters*: | `data` | The single-banded short data |
| | `offset` | The offset |
| | `isUShort` | True if the data type is `DataBuffer.TYPE_USHORT`; false if the data type is `DataBuffer.TYPE_SHORT`. |

- `LookupTableJAI(int[] data)`
  constructs a single-banded int lookup table with an index offset

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The single-banded int data |

- `LookupTableJAI(int[] data, int offset)`
  constructs a single-banded int lookup table with an index offset

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The single-banded int data |
  | | `offset` | The offset |

- `LookupTableJAI(float[] data)`
  constructs a single-banded float lookup table with an index offset of 0

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The single-banded float data |

- `LookupTableJAI(float[] data, int offset)`
  constructs a single-banded float lookup table with an index offset

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The single-banded float data |
  | | `offset` | The offset |

- `LookupTableJAI(double[] data)`
  constructs a single-banded double lookup table with an index offset of 0

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The single-banded double data |

- `LookupTableJAI(double[] data, int offset)`
  constructs a single-banded double lookup table with an index offset

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The single-banded double data |
  | | `offset` | The offset |

#### 7.6.1.2 Creating a Multi-band Lookup Table

The multi-band lookup table contains data for more than one channels or image components, such as separate arrays for R, G, and B. To create a lookup table for a multi-band input image, use one of the multi-band constructors. Like the single-band constructors, the multi-band constructors take up to three parameters:

- A pointer to the data to be stored in the table. The data may be of type Byte, Short, UShort, Int, Float, or Double.
- The offset. The offset selects the lookup table subrange. The offset value is subtracted from the input value before indexing into the table array. The constructors allow you to specify one offset for all of the bands or separate offsets for each band.
- A boolean flag that indicates whether Short data is of type Short or UShort.

Listing 7-4 shows an example of the construction of a multi-banded byte lookup table.

**Listing 7-4 Example Multi-band Lookup Table**

```
// Create the table data.
byte[][] tableData = new byte[3][0x10000];
for (int i = 0; i < 0x10000; i++) {
tableData[0][i] = (byte)(i >> 8); // this may be different
tableData[1][i] = (byte)(i >> 8); // for each band
tableData[2][i] = (byte)(i >> 8);
}

// Create a LookupTableJAI object to be used with the
// "lookup" operator.
LookupTableJAI table = new LookupTableJAI(tableData);
```

**API:** `javax.media.jai.LookupTableJAI`

- `LookupTableJAI(byte[][] data)`

  constructs a multi-banded byte lookup table with an index offset for each band of 0.

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded byte data in [band][index] format |

- `LookupTableJAI(byte[][] data, int offset)`

  constructs a multi-banded byte lookup table where all bands have the same index offset.

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded byte data in [band][index] format |
| | `offset` | The common offset for all bands |

- `LookupTableJAI(byte[][] data, int[] offsets)`

  constructs a multi-banded byte lookup table where each band has a different index offset.

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded byte data in [band][index] format |
| | `offsets` | The offsets for the bands |

- `LookupTableJAI(short[][] data, boolean isUShort)`

  constructs a multi-banded short or unsigned short lookup table. The index offset for each band is 0

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded short data in [band][index] format. |
| | `isUShort` | True if the data type is `DataBuffer.TYPE_USHORT`; false if the data type is `DataBuffer.TYPE_SHORT`. |

- `LookupTableJAI(short[][] data, int offset, boolean isUShort)`

  constructs a multi-banded short or unsigned short lookup table where all bands have the same index offset

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded short data in [band][index] format |
| | `offset` | The common offset for all bands |
| | `isUShort` | True if the data type is `DataBuffer.TYPE_USHORT`; false if the data type is `DataBuffer.TYPE_SHORT`. |

- `LookupTableJAI(short[][] data, int[] offsets, boolean isUShort)`

  constructs a multi-banded short or unsigned short lookup table where each band has a different index offset

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded short data in [band][index] format |
| | `offset` | The offsets for the bands |
| | `isUShort` | True if the data type is `DataBuffer.TYPE_USHORT`; false if the data type is `DataBuffer.TYPE_SHORT`. |

- `LookupTableJAI(int[][] data)`

  constructs a multi-banded int lookup table. The index offset for each band is 0

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The multi-banded int data in [band][index] format |

- `LookupTableJAI(int[][] data, int offset)`

  constructs a multi-banded int lookup table where all bands have the same index offset

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The multi-banded int data in [band][index] format |
  | | `offset` | The common offset for all bands |

- `LookupTableJAI(int[][] data, int[] offsets)`

  constructs a multi-banded int lookup table where each band has a different index offset

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The multi-banded int data in [band][index] format |
  | | `offset` | The offsets for the bands |

- `LookupTableJAI(float[][] data)`

  constructs a multi-banded float lookup table. The index offset for each band is 0

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The multi-banded float data in [band][index] format |

- `LookupTableJAI(float[][] data, int offset)`

  constructs a multi-banded float lookup table where all bands have the same index offset

  | | | |
  |---|---|---|
  | *Parameters*: | `data` | The multi-banded float data in [band][index] format |
  | | `offset` | The common offset for all bands |

- `LookupTableJAI(float[][] data, int[] offsets)`

  constructs a multi-banded float lookup table where each band has a different index offset

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded float data in [band][index] format |
| | `offset` | The offsets for the bands |

- `LookupTableJAI(double[][] data)`

  constructs a multi-banded double lookup table. The index offset for each band is 0

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded double data in [band][index] format |

- `LookupTableJAI(double[][] data, int offset)`

  constructs a multi-banded double lookup table where all bands have the same index offset

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded double data in [band][index] format |
| | `offset` | The common offset for all bands |

- `LookupTableJAI(double[][] data, int[] offsets)`

  constructs a multi-banded double lookup table where each band has a different index offset

| | | |
|---|---|---|
| *Parameters*: | `data` | The multi-banded double data in [band][index] format |
| | `offsets` | The offsets for the bands |

#### 7.6.1.3 Creating a Color-cube Lookup Table

Dithering operations that use a color cube are considerably faster than those that use a generic lookup table. However, the color cube provides less control over the exact contents of the lookup table.

The `ColorCube` class is a subclass of `LookupTableJAI` and represents a color cube lookup table. You create a colorcube using one of the `ColorCube.createColorCube` methods. Rather than specifying the data to be loaded into the lookup table, you provide an array of `dimensions`. The dimensions parameter specifies the size (or number of levels) of each band of the image.

Although a color cube implies three dimensions, that is not always the case. The color cube has the same number of dimensions as the image has bands. For example, a monochrome image requires only one dimension parameter.

The values in the dimensions parameter are signed. A positive value indicates that the corresponding color ramp increases. A negative value indicates that the ramp decreases.

JAI provides two predefined color cubes, which can be used for the ordered dither operation (see Section 6.6.1, "Ordered Dither"):

| ColorCube | Description |
|---|---|
| BYTE_496 | A ColorCube with dimensions 4:9:6, useful for dithering RGB images into 216 colors. The offset of this ColorCube is 38. This color cube dithers blue values in the source image to one of 4 blue levels, green values to one of 9 green levels, and red values to one of 6 red levels. This is the default color cube for the ordered dither operation. |
| BYTE_855 | A ColorCube with dimensions 8:5:5, useful for dithering $YC_bC_r$ images into 200 colors. The offset of this ColorCube is 54. This color cube dithers blue values in the source image to one of 8 blue levels, green values to one of 5 green levels, and red values to one of 5 red levels. |

These color cubes are specified by the colorMap parameter that is required by the OrderedDither operation.

**API:** javax.media.jai.ColorCube

```
• static ColorCube createColorCube(int dataType, int offset,
      int[] dimensions)
```

creates a multi-banded ColorCube of a specified data type.

| *Parameters*: | | |
|---|---|---|
| | dataType | The data type of the ColorCube. One of DataBuffer.TYPE_BYTE, DataBuffer.TYPE_SHORT, DataBuffer.TYPE_USHORT, DataBuffer.TYPE_INT, DataBuffer.TYPE_FLOAT, or DataBuffer.TYPE_DOUBLE. |
| | offset | The common offset for all bands. |
| | dimensions | The signed dimensions for each band. |

- `static ColorCube createColorCube(int dataType,`
  `    int[] dimensions)`

  create a multi-banded `ColorCube` of a specified data type with zero offset for all bands.

| | | |
|---|---|---|
| *Parameters*: | `dataType` | The data type of the `ColorCube`. One of `DataBuffer.TYPE_BYTE`, `DataBuffer.TYPE_SHORT`, `DataBuffer.TYPE_USHORT`, `DataBuffer.TYPE_INT`, `DataBuffer.TYPE_FLOAT`, or `DataBuffer.TYPE_DOUBLE`. |
| | `dimensions` | The signed dimensions for each band. |

- `static ColorCube createColorCubeByte(int[] dimensions)`

  constructs a multi-banded byte `ColorCube`.

| | | |
|---|---|---|
| *Parameters*: | `dimensions` | A list of signed sizes of each side of the color cube. |

- `static ColorCube createColorCubeByte(int offset,`
  `    int[] dimensions)`

  constructs a multi-banded byte ColorCube with an index offset common to all bands.

| | | |
|---|---|---|
| *Parameters*: | `offset` | The common offset for all bands. |
| | `dimensions` | A list of signed sizes of each side of the color cube. |

- `static ColorCube createColorCubeShort(int[] dimensions)`

  constructs a multi-banded short `ColorCube`.

- `static ColorCube createColorCubeShort(int offset,`
  `    int[] dimensions)`

  constructs a multi-banded short `ColorCube` with an index offset common to all bands.

- `static ColorCube createColorCubeUShort(int[] dimensions)`

  constructs a multi-banded unsigned short `ColorCube`.

-
  ```
  static ColorCube createColorCubeUShort(int offset,
      int[] dimensions)
  ```
  constructs a multi-banded unsigned short `ColorCube` with an index offset common to all bands.

-
  ```
  static ColorCube createColorCubeInt(int[] dimensions)
  ```
  constructs a multi-banded int `ColorCube`.

-
  ```
  static ColorCube createColorCubeInt(int offset,
      int[] dimensions)
  ```
  constructs a multi-banded int `ColorCube` with an index offset common to all bands.

-
  ```
  static ColorCube createColorCubeFloat(int[] dimensions)
  ```
  constructs a multi-banded float `ColorCube`.

-
  ```
  static ColorCube createColorCubeFloat(int offset,
      int[] dimensions)
  ```
  constructs a multi-banded float ColorCube with an index offset common to all bands.

-
  ```
  static ColorCube createColorCubeDouble(int[] dimensions)
  ```
  constructs a multi-banded double `ColorCube` with an index offset common to all bands.

-
  ```
  static ColorCube createColorCubeDouble(int offset,
      int[] dimensions)
  ```
  constructs a multi-banded double `ColorCube` with an index offset common to all bands.

### 7.6.2 Performing the Lookup

The `lookup` operation performs a general table lookup on a rendered or renderable image. The destination image is obtained by passing the source image through the lookup table. The source image may be single- or multi-banded of data types `byte`, `ushort`, `short`, or `int`. The lookup table may be single- or multi-banded of any JAI-supported data types.

The destination image must have the same data type as the lookup table, and its number of bands is determined based on the number of bands of the source and the table. If the source is single-banded, the destination has the same number of bands as the lookup table; otherwise, the destination has the same number of bands as the source.

If either the source or the table is single-banded and the other one is multi-banded, the single band is applied to every band of the multi-banded object. If both are multi-banded, their corresponding bands are matched up.

The table may have a set of offset values, one for each band. This value is subtracted from the source pixel values before indexing into the table data array.

It is the user's responsibility to make certain the lookup table supplied is suitable for the source image. Specifically, the table data must cover the entire range of the source data. Otherwise, the result of this operation is undefined.

By the nature of this operation, the destination may have a different number of bands and/or data type from the source. The `SampleModel` of the destination is created in accordance with the actual lookup table used in a specific case.

There are three specific cases of table lookup that determine the pixel values of the destination image:

- If the source image is single-banded and the lookup table is single- or multi-banded, the destination image has the same number of bands as the lookup table:

```
for (int h = 0; h < dstHeight; h++) {
    for (int w = 0; w < dstWidth; w++) {
        for (int b = 0; b < dstNumBands; b++) {
         dst[h][w][b] = table[b][src[h][w][0] - offsets[b]]
        }
    }
}
```

- If the source image is multi-banded and the lookup table is single-banded, the destination image has the same number of bands as the source image:

```
for (int h = 0; h < dstHeight; h++) {
    for (int w = 0; w < dstWidth; w++) {
        for (int b = 0; b < dstNumBands; b++) {
         dst[h][w][b] = table[0][src[h][w][b] - offsets[0]]
       }
     }
}
```

- If the source image is multi-banded and the lookup table is multi-banded, with the same number of bands as the source image, the destination image will have the same number of bands as the source image:

```
for (int h = 0; h < dstHeight; h++) {
    for (int w = 0; w < dstWidth; w++) {
        for (int b = 0; b < dstNumBands; b++) {
                dst[h][w][b] = table[b][src[h][w][b] - offsets[b]]
              }
          }
      }
```

The `lookup` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `table` | `LookupTableJAI` | The lookup table through which the source image is passed. |

See Section 7.6.1, "Creating the Lookup Table" for more information.

For a complete example of the `Lookup` operation, see Listing A-1 on page 417.

## 7.6.3 Other Lookup Table Operations

### 7.6.3.1 Reading the Table Data

Several methods are available to read the current contents of the lookup table. The choice of method depends on the data format: byte, short, integer, floating-point, or double floating-point.

**API:** `javax.media.jai.LookupTableJAI`

- `java.awt.image.DataBuffer getData()`
  returns the table data as a `DataBuffer`.
- `byte[][] getByteData()`
  returns the byte table data in array format.
- `byte[] getByteData(int band)`
  returns the byte table data of a specific band in array format.
- `short[][] getShortData()`
  returns the short table data in array format.
- `short[] getShortData(int band)`
  returns the short table data of a specific band in array format.
- `int[][] getIntData()`
  returns the integer table data in array format.

- `int[] getIntData(int band)`
  returns the integer table data of a specific band in array format.

- `float[][] getFloatData()`
  returns the float table data in array format.

- `float[] getFloatData(int band)`
  returns the float table data of a specific band in array format.

- `double[][] getDoubleData()`
  returns the double table data in array format.

- `double[] getDoubleData(int band)`
  returns the double table data of a specific band in array format.

#### 7.6.3.2 Reading the Table Offsets

There are three methods for reading the offset values within the current lookup table.

**API:** `javax.media.jai.LookupTableJAI`

- `int[] getOffsets()`
  returns the index offsets of entry 0 for all bands.

- `int getOffset()`
  returns the index offset of entry 0 for the default band.

- `int getOffset(int band)`
  returns the index offset of entry 0 for a specific band.

| | | |
|---|---|---|
| *Parameters*: | `band` | The band to read |

#### 7.6.3.3 Reading the Number of Bands

A single method is used to read the number of bands in the lookup table.

**API:** `javax.media.jai.LookupTableJAI`

- `int getNumBands()`
  returns the number of bands of the table.

#### 7.6.3.4 Reading the Number of Entries Per Band

A single method is used to read the number of entries per band in the lookup table.

**API:** `javax.media.jai.LookupTableJAI`

- `int getNumEntries()`

  returns the number of entries per band of the table.

#### 7.6.3.5 Reading the Data Type

A single method is used to read the data type of the lookup table.

**API:** `javax.media.jai.LookupTableJAI`

- `int getDataType()`

  returns the data type of the table data.

#### 7.6.3.6 Reading the Destination Bands and SampleModel

**API:** `javax.media.jai.LookupTableJAI`

- `int getDestNumBands(int sourceNumBands)`

  returns the number of bands of the destination image, based on the number of bands of the source image and lookup table.

  | | | |
  |---|---|---|
  | *Parameters*: | `sourceNum-Bands` | The number of bands of the source image. |

-
  ```
  java.awt.image.SampleModel
      getDestSampleModel(java.awt.image.SampleModel
      srcSampleModel)
  ```

  returns a `SampleModel` suitable for holding the output of a lookup operation on the source data described by a given `SampleModel` with this table. The width and height of the destination `SampleModel` are the same as that of the source. This method returns null if the source `SampleModel` has a non-integral data type.

  | | | |
  |---|---|---|
  | *Parameters*: | `srcSample-Model` | The `SampleModel` of the source image. |

-
  ```
  java.awt.image.SampleModel
      getDestSampleModel(java.awt.image.SampleModel
      srcSampleModel, int width, int height)
  ```

  returns a `SampleModel` suitable for holding the output of a lookup operation on the source data described by a given `SampleModel` with this table. This method will return null if the source `SampleModel` has a non-integral data type.

| | | |
|---|---|---|
| *Parameters*: | `srcSample-Model` | The `SampleModel` of the source image. |
| | `width` | The width of the destination `SampleModel`. |
| | `height` | The height of the destination `SampleModel`. |

## 7.7 Convolution Filtering

Convolution filtering is often used to reduce the effects of noise in images or to sharpen the detail in blurred images. Convolution filtering is a form of *spatial filtering* that computes each output sample by multiplying elements of a kernel with the samples surrounding a particular source sample.

Convolution filtering operates on a group of input pixels surrounding a center pixel. The adjoining pixels provide important information about brightness trends in the area of the pixel being processed.

Convolution filtering moves across the source image, pixel by pixel, placing resulting pixels into the destination image. The resulting brightness of each source pixel depends on the group of pixels surrounding the source pixel. Using the brightness information of the source pixel's neighbors, the convolution process calculates the spatial frequency activity in the area, making it possible to filter the brightness based on the spatial frequency of the area.

Convolution filtering uses a *convolve kernel*, containing an array of convolution coefficient values, called *key elements*, as shown in Figure 7-9. The array is not restricted to any particular size, and does not even have to be square. The kernel can be $1 \times 1$, $3 \times 3$, $5 \times 5$, $M \times N$, and so on. A larger kernel size affords a more precise filtering operation by increasing the number of neighboring pixels used in the calculation. However, the kernel cannot be bigger in any dimension than the image data. Also, the larger the kernel, the more computations that are required to be performed. For example, given a $640 \times 480$ image and a $3 \times 3$ kernel, the convolve operation requires over five million total multiplications and additions.

The convolution filtering operation computes each output sample by multiplying the key elements of the kernel with the samples surrounding a particular source pixel. For each destination pixel, the kernel is rotated 180 degrees and its key element is placed over the source pixel corresponding with the destination pixel. The key elements are multiplied with the source pixels under them, and the resulting products are summed together to produce the destination sample value.

The selection of the weights for the key elements determines the nature of the filtering action, such as *high-pass* or *low-pass*. If the values of the key elements are the reciprocal of the number of key elements in the kernel (for example, 1/9 for a 3 × 3 kernel), the result is a conventional low-pass averaging process. If the weights are altered, certain pixels in the kernel will have an increased or decreased influence in the average. Figure 7-10 shows three example convolve filters, low-pass, high-pass, and Laplacian.

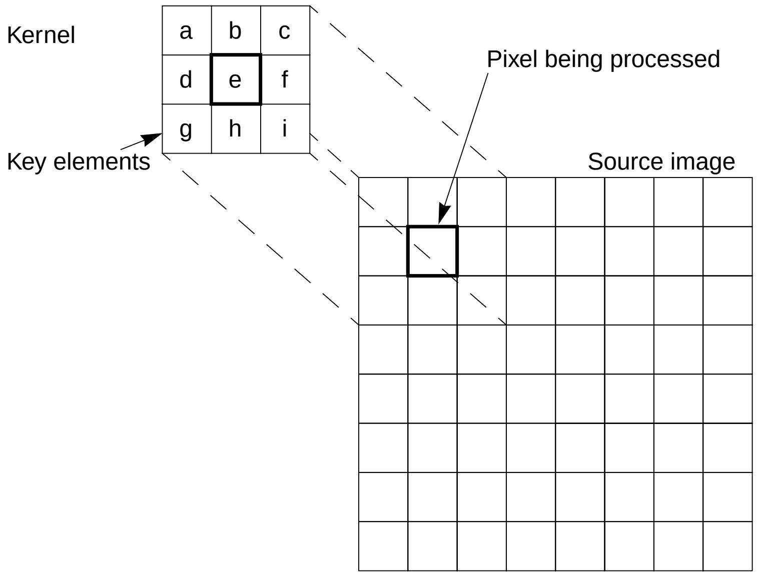


**Figure 7-9 Convolve Kernel**

| | | |
|---|---|---|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

Example low-pass filter

| | | |
|---|---|---|
| –1 | –1 | –1 |
| –1 | 9 | –1 |
| –1 | –1 | –1 |

Example high-pass filter

| | | |
|---|---|---|
| 1 | –2 | 1 |
| –2 | 5 | –2 |
| 1 | –2 | 1 |

Example Laplacian filter

**Figure 7-10 Convolve Filter Samples**

The low-pass filter, also known as a *box filter*, attenuates the high-spatial frequency components of an image and has little affect on the low-frequency components. The effect of passing an image through a low-pass filter is a slight blurring. This is caused by attenuating the sharp brightness transitions between edges and makes the image appear to have less detail. See also Section 7.7.2, “Box Filter.”

The high-pass filter has the opposite effect of the low-pass filter, accentuating the high-frequency components and offering little affect on the low-frequency components. The effect of passing an image through a high-pass filter is a sharper image with increased detail in the areas of brightness transition.

The Laplacian filter is another image detail sharpening filter that works well for noise-free images. This filter subtracts the brightness values of the four neighboring pixels from the central pixel. The result of applying this filter is to reduce the gray level to zero.

## 7.7.1 Performing the Convolve Operation

The following example code shows a `convolve` operation on a single sample `dst[x][y]`, which assumes that the kernel is of size `M` × `N` and has already been rotated through 180 degrees. The kernel’s key element is located at position (`xKey`, `yKey`).

```
dst[x][y] = 0;
 for (int i = -xOrigin; i < -xOrigin + width; i++) {
 for (int j = -yOrigin; j < -yOrigin + height; j++) {
  dst[x][y] += src[x + i][y + j]*kernel[xOrigin + i][yOrigin + j];
   }
}
```

Convolution, or any neighborhood operation, leaves a band of pixels around the edges undefined. For example, for a 3 × 3 kernel, only four kernel elements and four source pixels contribute to the destination pixel located at (0,0). Such pixels are not included in the destination image. A border extension may be added via the `BorderExtender` class. The type of border extension can be specified as a `RenderingHint` to the `JAI.create` method. If no border extension type is provided, a default extension of `BorderExtender.BORDER_COPY` will be used to perform the extension. See Section 3.7.3, "Rendering Hints."

The `convolve` operation takes one rendered source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `kernel` | `KernelJAI` | The convolution kernel. See Section 6.9, "Constructing a Kernel." |

The default `kernel` is `null`.

Listing 7-5 shows a code sample for a `Convolve` operation.

**Listing 7-5 Example Convolve Operation**

```
// Create the kernel.
kernel = new KernelJAI
float[] = {  0.0F, -1.0F,  0.0F,
            -1.0F,  5.0F, -1.0F,
             0.0F, -1.0F,  0.0F };

// Create the convolve operation.
im1 = JAI.create("convolve", im, kernel);
```

## 7.7.2 Box Filter

The `BoxFilter` operation is a special case of convolve operation in which each source pixel contributes the same weight to the destination pixel. The box filter operation determines the intensity of a pixel in an image by averaging the source pixels within a rectangular area around the pixel. The pixel values of the destination image are defined by the following pseudocode:

```
int count = width * height; // # of pixels in the box
for (int b = 0; b < numBands; b++) {
     int total = 0;
     for (int j = -yKey; j < -yKey + height; j++) {
          for (int i = -xKey; i < -xKey + width; i++) {
              total += src[x+i][y+j][b];
           }
      }
     dst[x][y][b] = (total + count/2) / count; // round
}
```

The `BoxFilter` operation uses a low-pass filter that passes (leaves untouched) the low spatial frequency components of the image and attenuates the high-frequency components. In an area of the image that has constant brightness, the brightness values are passed untouched. When the filter passes over an area of sharp black to white transitions, the brightness range is greatly attenuated.

Convolution, like any neighborhood operation, leaves a band of pixels around the edges undefined. For example, for a $3 \times 3$ kernel, only four kernel elements and four source pixels contribute to the convolution pixel at the corners of the source image. Pixels that do not allow the full kernel to be applied to the source are not included in the destination image. A `Border` operation (see Section 7.2, "Adding Borders to Images") may be used to add an appropriate border to the source image to avoid shrinkage of the image boundaries.

The kernel may not be bigger in any dimension than the image data.

The `BoxFilter` operation takes one rendered source image and four parameters:

| Parameter | Type | Description |
|---|---|---|
| `width` | `Integer` | The width of the box. |
| `height` | `Integer` | The height of the box. |
| `xKey` | `Integer` | The *x* position of the key element. |
| `yKey` | `Integer` | The *y* position of the key element. |

The `width` parameter is required. The remaining parameters may be `null` and, if not supplied, default to the following values:

$$\mathtt{height} = \mathtt{width}$$

$$\mathtt{xKey} = \frac{\mathtt{width}}{2}$$

$$\mathtt{yKey} = \frac{\mathtt{height}}{2}$$

Listing 7-6 shows a code sample for a `BoxFilter` operation.

**Listing 7-6 Example BoxFilter Operation**

```
    // Read the arguments.
    String fileName = args.length > 0 ? args[0] : DEFAULT_FILE;
    int width = args.length > 1 ?
        Integer.decode(args[1]).intValue() : DEFAULT_SIZE;
    int height = args.length > 2 ?
        Integer.decode(args[2]).intValue() : width;
```

**Listing 7-6 Example BoxFilter Operation (Continued)**

```
    new BoxFilterExample(fileName, width, height);
    }

    public BoxFilterExample(String fileName, int width, int height)

    // Load the image.
    RenderedOp src =  JAI.create("fileload", fileName);

    // Create the BoxFilter operation.
    RenderedOp dst = JAI.create("boxfilter", src,
                                width, height,
                                width/2, height/2);
```

## 7.8 Median Filtering

A median filter is used to remove impulse noise spikes from an image and thus smoothing the image. Impulse noise spikes appear as bright or dark pixels randomly distributed throughout the image. Noise spikes are normally significantly brighter or darker than their neighboring pixels and can easily be found by comparing the median value of a group of input pixels.

The median filter is a neighborhood-based ranking filter in which the pixels in the neighborhood are ranked in the order of their levels. The median value of the group is then stored in the output pixel. The resulting image is then free of pixel brightnesses that are at the extremes in each input group of pixels.

The noise-reducing effect that the median filter has on an image depends on two related things: the spatial extent of the neighborhood (the mask) and the number of pixels involved in the computation. The `MedianFilter` operation supports three different mask shapes, a square, a plus, and an X-shape, as shown in Figure 7-11.

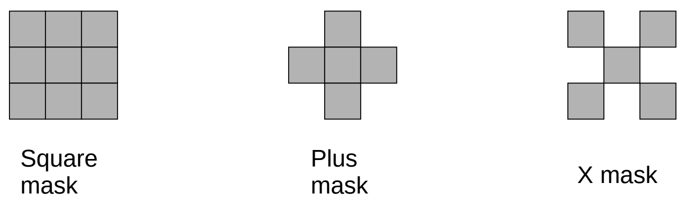


**Figure 7-11 Median Filter Masks**

The `MedianFilter` operation may also be used to compute the *separable median* of a 3 × 3 or 5 × 5 region of pixels. The separable median is defined as the median of the medians of each row. For example, if the pixel values in a 3 × 3 window are as follows:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 5 | 6 | 7 |
| 4 | 8 | 9 |

the overall (non-separable) median value is 5, while the separable median is equal to the median of the three row medians: median(1, 2, 3) = 2, median(5, 6, 7) = 6, and median(4, 8, 9) = 8, yielding an overall median of 6. The separable median may be obtained by specifying a mask of type `MEDIAN_MASK_SQUARE_SEPARABLE`.

The `MedianFilter` operation takes one rendered source image and two parameters:

| Parameter | Type | Default | Description |
|---|---|---|---|
| `maskShape` | `Integer` | `MASK_ SQUARE` | The shape of the mask to be used for Median Filtering |
| `maskSize` | `Integer` | 3 | The size (width and height) of the mask to be used in Median Filtering. |

The `maskShape` parameter is one of the following:

| maskShape | Description |
|---|---|
| `MEDIAN_MASK_SQUARE` | A square-shaped mask. The default. |
| `MEDIAN_MASK_PLUS` | A plus-shaped mask. |
| `MEDIAN_MASK_X` | An X-shaped mask. |
| `MEDIAN_MASK_SQUARE_ SEPARABLE` | A separable square mask, used for the separable median operation. |

The `maskSize` parameter must be 1 (1 × 1) or greater. The default value, if one is not provided, is 3 (3 × 3). For large masks, the noise reduction effect of more pixels used in the computation of the median value reaches a point of diminishing returns. Typical mask sizes are 3 × 3 and 5 × 5.

## 7.9 Frequency Domain Processing

Images contain spatial details that are seen as brightness transitions, cycling from dark to light and back to dark. The rate at which the transitions occur in an image represent the image's *spatial frequency.*

An image's spatial frequency can be measured horizontally, vertically, or at any diagonal in between. An image contains many spatial frequencies that, when combined in the correct magnitude and phase, form the complex details of the image.

A *frequency transform* decomposes an image from its spatial domain form of brightness into a frequency domain form of fundamental frequency components. Each frequency component contains a magnitude and phase value. An *inverse frequency transform* converts an image from its frequency form back to its spatial form.

### 7.9.1 Fourier Transform

JAI supports the most common type of frequency transform, the *discrete Fourier transform* and its inverse, the inverse discrete Fourier transform. The discrete Fourier transform of an image is a two-dimensional process. The result of the transform is a two-dimensional array of values, each having two parts: real and imaginary. Each value represents a distinct spatial frequency component. The frequency-transform image has as many values as there are pixels in the source image.

The real portion of the values can be displayed as an image, visually showing the frequency components of the source image. The result is in "wrap around" order, with the zero-frequency point (also known as "DC" for direct current) at the upper left corner and the high frequencies at the center.

#### 7.9.1.1 Discrete Fourier Transform

The `DFT` (discrete Fourier transform) operation computes the discrete Fourier transform of an image. A negative exponential is used as the basis function for the transform. The operation supports real-to-complex, complex-to-complex, and complex-to-real transforms. A complex image must have an even number of bands, with the even bands (0, 2, etc.) representing the real parts and the odd bands (1, 3, etc.) the imaginary parts of each complex pixel.

If an underlying fast Fourier transform (FFT) implementation is used that requires that the image dimensions be powers of 2, the width and height may each be increased to the power of 2 greater than or equal to the original width and height, respectively.

The `dft` operation takes one rendered or renderable source image and two parameters.

| Parameter | Type | Description |
|---|---|---|
| `scalingType` | `Integer` | The type of scaling to perform. One of `DFTDescriptor.SCALING_NONE`, `DFTDescriptor.SCALING_UNITARY`, or `DFTDescriptor.SCALING_DIMENSIONS`. |
| `dataNature` | `Integer` | The nature of the data. One of `DFTDescriptor.REAL_TO_COMPLEX`, `DFTDescriptor.COMPLEX_TO_COMPLEX`, or `DFTDescriptor.COMPLEX_TO_REAL`. |

The default parameters for this operation are `SCALING_NONE` and `REAL_TO_COMPLEX`.

The `scalingType` parameter defines how the image dimensions may be scaled, as follows:

| scalingType | Description |
|---|---|
| `DFTDescriptor.SCALING_NONE` | The transform is not to be scaled (the default). |
| `DFTDescriptor.SCALING_UNITARY` | The transform is to be scaled by the square root of the product of its dimensions. |
| `DFTDescriptor.SCALING_DIMENSIONS` | The transform is to be scaled by the product of its dimensions. |

The `dataNature` parameter specifies the nature of the source and destination data, as follows.

| dataNature | Description |
|---|---|
| `DFTDescriptor.REAL_TO_COMPLEX` | The source data are real and the destination data complex. |
| `DFTDescriptor.COMPLEX_TO_COMPLEX` | The source and destination data are both complex. |
| `DFTDescriptor.COMPLEX_TO_REAL` | The source data are complex and the destination data real. |

If the source data are complex, the number of bands in the source image must be a multiple of 2. The number of bands in the destination must match that which would be expected given the number of bands in the source image and the specified nature of the source and destination data. If the source image is real, the number of bands in the destination will be twice that in the source. If the destination image is real, the number of bands in the destination will be half that in the source. Otherwise the number of bands in the source and destination must be equal.

The DFT operation defines a PropertyGenerator that sets the COMPLEX property of the image to FALSE if the dataNature parameter is COMPLEX_TO_REAL and to TRUE if the dataNature parameter is REAL_TO_COMPLEX or COMPLEX_TO_COMPLEX. The value of this property may be retrieved by calling the getProperty() method with COMPLEX as the property name.

Listing 7-7 shows a code sample for a DFT operation.

**Listing 7-7 Example DFT Operation**

```
// Create the ParameterBlock.
ParameterBlock pb = new ParameterBlock();
pb.addSource(src)
pb.add(DFTDescriptor.SCALING_NONE);
pb.add(DFTDescriptor.REAL_TO_COMPLEX);

// Create the DFT operation.
PlanarImage dft = (PlanarImage)JAI.create("dft", pb, null);

// Get the DFT image information.
int width = dft.getWidth();
int height = dft.getHeight();
int numBands = dft.getSampleModel().getNumBands();
int dataType = dft.getSampleModel().getDataType();

// Calculate the cutoff "frequencies" from the threshold.
threshold /= 200.0F;
int minX = (int)(width*threshold);
int maxX = width - 1 - minX;
int minY = (int)(height*threshold);
int maxY = height - 1 - minY;

// Retrieve the DFT data.
Raster dftData = dft.getData();
double[] real =
    dftData.getSamples(0, 0, width, height, 0, (double[])null);
double[] imag =
    dftData.getSamples(0, 0, width, height, 1, (double[])null);

double[] HR = new double[real.length];
double[] HI = new double[imag.length];
double[] LR = new double[real.length];
double[] LI = new double[imag.length];
```

### 7.9.1.2 Inverse Discrete Fourier Transform

The IDFT (inverse discrete Fourier transform) operation computes the inverse discrete Fourier transform of an image. A positive exponential is used as the basis function for the transform. The operation supports real-to-complex, complex-to-complex, and complex-to-real transforms. A complex image must have an even number of bands, with the even bands (0, 2, etc.) representing the real parts and the odd bands (1, 3, etc.) the imaginary parts of each complex pixel.

If an underlying fast Fourier transform (FFT) implementation is used that requires that the image dimensions be powers of 2, the width and height may each be increased to the power of 2 greater than or equal to the original width and height, respectively.

The IDFT operation takes one rendered or renderable source image and two parameters.

| Parameter | Type | Description |
|---|---|---|
| scalingType | Integer | The type of scaling to perform. One of DFTDescriptor.SCALING_NONE, DFTDescriptor.SCALING_UNITARY, or DFTDescriptor.SCALING_DIMENSIONS. |
| dataNature | Integer | The nature of the data. One of DFTDescriptor.REAL_TO_COMPLEX, DFTDescriptor.COMPLEX_TO_COMPLEX, or DFTDescriptor.COMPLEX_TO_REAL. |

The default parameters for this operation are SCALING_DIMENSIONS and COMPLEX_TO_REAL.

The scalingType parameter defines how the image dimensions may be scaled, as follows:

| scalingType | Description |
|---|---|
| DFTDescriptor.SCALING_NONE | The transform is not to be scaled. |
| DFTDescriptor.SCALING_UNITARY | The transform is to be scaled by the square root of the product of its dimensions. |
| DFTDescriptor.SCALING_DIMENSIONS | The transform is to be scaled by the product of its dimensions (the default). |

The `dataNature` parameter specifies the nature of the source and destination data, as follows.

| dataNature | Description |
|---|---|
| `DFTDescriptor.REAL_TO_COMPLEX` | The source data are real and the destination data complex. |
| `DFTDescriptor.COMPLEX_TO_COMPLEX` | The source and destination data are both complex. |
| `DFTDescriptor.COMPLEX_TO_REAL` | The source data are complex and the destination data real. |

If the source data are complex, the number of bands in the source image must be a multiple of 2. The number of bands in the destination must match that which would be expected given the number of bands in the source image and the specified nature of the source and destination data. If the source image is real, the number of bands in the destination will be twice that in the source. If the destination image is real, the number of bands in the destination will be half that in the source. Otherwise the number of bands in the source and destination must be equal.

The `IDFT` operation defines a `PropertyGenerator` that sets the `COMPLEX` property of the image to `FALSE` if the `dataNature` parameter is `COMPLEX_TO_REAL` and to `TRUE` if the `dataNature` parameter is `REAL_TO_COMPLEX` or `COMPLEX_TO_COMPLEX`. The value of this property may be retrieved by calling the getProperty() method with `COMPLEX` as the property name.

## 7.9.2 Cosine Transform

The discrete cosine transform (DCT) is similar to the discrete Fourier transform (see Section 7.9.1.1, “Discrete Fourier Transform”). However, the DCT is better at compactly representing very small images. Like the discrete Fourier transform (DFT), the DCT also has an inverse operation, the *inverse discrete cosine transform* (IDCT).

### 7.9.2.1 Discrete Cosine Transform (DCT)

The `DCT` operation computes the even discrete cosine transform of an image. Each band of the destination image is derived by performing a two-dimensional DCT on the corresponding band of the source image.

The `DCT` operation takes one rendered or renderable source image and no parameters.

Listing 7-8 shows a code sample for a DCT operation.

**Listing 7-8 Example DCT Operation**

```
// Load the source image.
RenderedImage src = (RenderedImage)JAI.create("fileload",
                    fileName);

// Calculate a DCT image from the source image.
ParameterBlock pb = (new ParameterBlock()).addSource(src);
PlanarImage dct = JAI.create("dct", pb, null);

// Get the DCT image data.
int width = dct.getWidth();
int height = dct.getHeight();
int numBands = dct.getSampleModel().getNumBands();
int dataType = dct.getSampleModel().getDataType();
double[] dctData =
    dct.getData().getPixels(0, 0, width, height,
                           (double[])null);
double[] pixels = new double[dctData.length];
```

### 7.9.2.2 Inverse Discrete Cosine Transform (IDCT)

The IDCT operation computes the inverse even discrete cosine transform of an image. Each band of the destination image is derived by performing a two-dimensional inverse DCT on the corresponding band of the source image.

The IDCT operation takes one rendered or renderable source image and no parameters.

Listing 7-9 shows a code sample for an operation that first takes the discrete cosine transform of an image, then computes the inverse discrete cosine transform.

**Listing 7-9 Example IDCT Operation**

```
// Calculate a DCT image from the source image.
System.out.println("Creating DCT of source image ...");
ParameterBlock pb = (new ParameterBlock()).addSource(src);
PlanarImage dct = JAI.create("dct", pb, null);

// Calculate an IDCT image from the DCT image.
System.out.println("Creating IDCT of DCT of source image ...");
pb = (new ParameterBlock()).addSource(dct);
PlanarImage idct = JAI.create("idct", pb, null);
```

**Listing 7-9 Example IDCT Operation**

```
    // Create display image for inverse DCT of DCT of source image.
    System.out.println("Creating display image for IDCT of DCT");
    pixels = idct.getData().getPixels(0, 0, width, height,
                                      (double[])pixels);
    BufferedImage bi = createBI(colorImage, width, height, pixels);
```

## 7.9.3 Magnitude Enhancement

The `magnitude` operation computes the magnitude of each pixel of a complex image. The source image must have an even number of bands, with the even bands (0, 2, etc.) representing the real parts and the odd bands (1, 3, etc.) the imaginary parts of each complex pixel. The destination image has at most half the number of bands of the source image with each sample in a pixel representing the magnitude of the corresponding complex source sample.

The magnitude values of the destination image are defined by the following pseudocode:

```
dstPixel[x][y][b] = sqrt(src[x][y][2b]$^2$ + src[x][y][2b + 1]$^2$)
```

where the number of bands *b* varies from zero to one less than the number of bands in the destination image.

For integral image data types, the result is rounded and clamped as needed.

The `magnitude` operation takes one rendered or renderable source image containing complex data and no parameters.

Listing 7-10 shows a code sample for a `magnitude` operation.

**Listing 7-10 Example Magnitude Operation**

```
    // Calculate a DFT image from the source image.
    pb = new ParameterBlock();
    pb.addSource(src).add(DFTDescriptor.SCALING_NONE);
    PlanarImage dft = JAI.create("dft", pb, null);

    // Create the ParameterBlock specifying the source image.
    pb = new ParameterBlock();
    pb.addSource(dft);

    // Calculate the magnitude.
    PlanarImage magnitude = JAI.create("magnitude", pb, null);
```

### 7.9.4 Magnitude-squared Enhancement

The MagnitudeSquared operation computes the squared magnitude of each pixel of a complex image. The source image must have an even number of bands, with the even bands (0, 2, etc.) representing the real parts and the odd bands (1, 3, etc.) the imaginary parts of each complex pixel. The destination image has at most half the number of bands of the source image with each sample in a pixel representing the magnitude of the corresponding complex source sample.

The squared magnitude values of the destination image are defined by the following pseudocode:

```
dstPixel[x][y][b] = $src[x][y][2b]^2$ + $src[x][y][2b + 1]^2$
```

where the number of bands *b* varies from zero to one less than the number of bands in the destination image.

For integral image data types, the result is rounded and clamped as needed.

The MagnitudeSquared operation takes one rendered or renderable source image containing complex data and no parameters.

### 7.9.5 Phase Enhancement

The Phase operation computes the phase angle of each pixel of a complex image. The source image must have an even number of bands, with the even bands (0, 2, etc.) representing the real parts and the odd bands (1, 3, etc.) the imaginary parts of each complex pixel. The destination image has at most half the number of bands of the source image with each sample in a pixel representing the phase angle of the corresponding complex source sample.

The angle values of the destination image are defined by the following pseudocode:

```
dst[x][y][b] = atan2(src[x][y][2b + 1], src[x][y][2b])
```

where the number of bands *b* varies from zero to one less than the number of bands in the destination image.

For integral image data types, the result is rounded and scaled so the “natural” arctangent range from [–π, π) is remapped into the range [0, MAXVALUE). The result for floating point image data types is the value returned by the atan2() method.

The phase operation takes one rendered or renderable source image containing complex data and no parameters.

### 7.9.6 Complex Conjugate

The `Conjugate` operation computes the complex conjugate of a complex image. The operation negates the imaginary components of a rendered or renderable source image containing complex data. The source image must contain an even number of bands with the even-indexed bands (0, 2, etc.) representing the real and the odd-indexed bands (1, 3, etc.) the imaginary parts of each pixel. The destination image similarly contains an even number of bands with the same interpretation and with contents defined by:

```
dst[x][y][2*k]   =  src[x][y][2*k];
dst[x][y][2*k+1] = -src[x][y][2*k+1];
```

where the index *k* varies from zero to one less than the number of complex components in the destination image.

The `Conjugate` operation takes one rendered or renderable source image containing complex data and no parameters.

### 7.9.7 Periodic Shift

The `PeriodicShift` operation computes the periodic translation of an image. The destination image of the `PeriodicShift` operation is the infinite periodic extension of the source image with horizontal and vertical periods equal to the image width and height, respectively, shifted by a specified amount along each axis and clipped to the bounds of the source image. Thus for each band *b* the destination image sample at location (*x*,*y*) is defined by:

```
if(x < width - shiftX) {
    if(y < height - shiftY) {
        dst[x][y][b] = src[x + shiftX][y + shiftY][b];
    } else {
        dst[x][y][b] = src[x + shiftX][y - height + shiftY][b];
    }
} else {
    if(y < height - shiftY) {
        dst[x][y][b] = src[x - width + shiftX][y + shiftY][b];
    } else {
        dst[x][y][b] = src[x - width + shiftX][y - height +
                                               shiftY][b];
    }
}
```

where `shiftX` and `shiftY` denote the translation factors along the *x* and *y* axes, respectively.

The PeriodicShift operation takes one rendered or renderable source image and two parameters.

| Parameter | Type | Description |
|---|---|---|
| shiftX | Integer | The displacement in the *x* direction. |
| shiftY | Integer | The displacement in the *y* direction. |

### 7.9.8 Polar to Complex

The PolarToComplex operation computes a complex image from a magnitude and a phase image. The operation creates an image with complex-valued pixels from two images, the respective pixel values of which represent the magnitude (modulus) and phase of the corresponding complex pixel in the destination image.

The source images should have the same number of bands. The first source image contains the magnitude values and the second source image the phase values. The destination will have twice as many bands with the even-indexed bands (0, 2, etc.) representing the real and the odd-indexed bands (1, 3, etc.) the imaginary parts of each pixel.

The pixel values of the destination image are defined for a given complex sample by the following pseudocode:

```
dst[x][y][2*b]   = src0[x][y][b]*Math.cos(src1[x][y][b])
dst[x][y][2*b+1] = src0[x][y][b]*Math.sin(src1[x][y][b])
```

where the index *b* varies from zero to one less than the number of bands in the source images.

For phase images with integral data type, it is assumed that the actual phase angle is scaled from the range [–PI, PI] to the range [0, MAX_VALUE] where MAX_VALUE is the maximum value of the data type in question.

The PolarToComplex operation takes two rendered or renderable source images and no parameters.

### 7.9.9 Images Based on a Functional Description

The ImageFunction operation generates an image from a functional description. This operation permits the creation of images on the basis of a functional specification, which is provided by an object that is an instance of a class that implements the javax.media.jai.ImageFunction interface. In other words, to use this operation, a class containing the functional information must be created and this class must implement the `ImageFunction` interface.

The `ImageFunction` interface merely defines the minimal set of methods required to represent such a function. The actual implementation of a class implementing this interface is left to the programmer.

For example, if the function you wanted to generate was the negative exponential

```
exp(-|x| - |y|)
```

The `javax.media.jai.ImageFunction` implementation would return the following values:

- `isComplex()` would return false
- `getNumElements()` would return 1
- `float[] real = new real[width*height];`
  `getElements(x, y, width, height, real, null);`

and the implementation would initialize the array `real` such that

```
real[j*width + i] = exp(-|x + i| - |y + j|)
```

or, equivalently

```
real[k] = exp(-|x + (k % width)]| - |y + (k / width)|)
```

where $0 \le k <$ width*height.

The (*x*,*y*) coordinates passed to the `ImageFunction.getElements()` methods are derived by applying an optional translation and scaling to the image *x* and *y* coordinates. The image *x* and *y* coordinates as usual depend on the values of the minimum *x* and *y* coordinates of the image, which need not be zero.

Specifically, the function coordinates passed to `getElements()` are calculated from the image coordinates as:

```
functionX = xScale*imageX + xTrans;
functionY = yScale*imageY + yTrans;
```

The number of bands in the destination image will be equal to the value returned by the `ImageFunction.getNumElements()` method unless the `ImageFunction.isComplex()` method returns `true`, in which case it will be twice that. The data type of the destination image is determined by the `SampleModel` specified by an `ImageLayout` object provided via a hint. If no layout hint is provided, the data type will default to single-precision floating point.

The double precision floating point form of the `getElements()` method will be invoked if and only if the data type is specified to be `double`. For all other data types the single precision form of `getElements()` will be invoked and the destination sample values will be clamped to the data type of the image.

The `ImageFunction` operation takes seven parameters.

| **Parameter** | **Type** | **Description** |
|---|---|---|
| `function` | `ImageFunction` | The functional description. |
| `width` | `Integer` | The image width. |
| `height` | `Integer` | The image height. |
| `xScale` | `Float` | The *x* scale factor. |
| `yScale` | `Float` | The *y* scale factor. |
| `xTrans` | `Float` | The *x* translation. |
| `yTrans` | `Float` | The *y* translation. |

The image width and height are provided explicitly as parameters. These values override the width and height specified by an `ImageLayout` if such is provided.

**API:** `javax.media.jai.ImageFunction`

- `boolean isComplex();`
  returns whether or not each value's elements are complex.
- `int getNumElements();`
  returns the number of elements per value at each position.

-
  ```
  void getElements(float startX, float startY, float deltaX,
      float deltaY, int countX, int countY, int element,
      float[] real, float[] imag);
  ```

  returns all values of a given element for a specified set of coordinates.

| *Parameters*: | | |
|---|---|---|
| | `startX` | The *x* coordinate of the upper left location to evaluate. |
| | `startY` | The *y* coordinate of the upper left location to evaluate. |
| | `deltaX` | The horizontal increment. |
| | `deltaY` | The vertical increment. |
| | `countX` | The number of points in the horizontal direction. |
| | `countY` | The number of points in the vertical direction. |
| | `element` | The element. |
| | `real` | A pre-allocated float array of length at least `countX*countY` in which the real parts of all elements will be returned. |
| | `imag` | A pre-allocated float array of length at least `countX*countY` in which the imaginary parts of all elements will be returned; may be null for real data, i.e., when `isComplex()` returns false. |

-
  ```
  void getElements(double startX, double startY, double deltaX,
      double deltaY, int countX, int countY, int element,
      double[] real, double[] imag);
  ```

  returns all values of a given element for a specified set of coordinates.

## 7.10 Single-image Pixel Point Processing

Pixel point operations are the most basic, yet necessary image processing operations. The pixel point operations are primarily contrast enhancement operations that alter the gray levels of an image's pixels. One-by-one, the gray level of each pixel in the source image is modified to a new value, usually by a mathematical relationship.

JAI supports the following single-image pixel point operations:

- Pixel inverting (`Invert`)
- Logarithmic enhancement (`Log`)

### 7.10.1 Pixel Inverting

The `Invert` operation inverts the pixel values of an image. For source images with signed data types, the pixel values of the destination image are defined by the following pseudocode:

```
dst[x][y][b] = -src[x][y][b]
```

For unsigned data types, the destination values are defined by the following pseudocode:

```
dst[x][y][b] = MAX_VALUE - src[x][y][b]
```

where `MAX_VALUE` is the maximum value supported by the system of the data type of the source pixel.

The `Invert` operation takes one rendered or renderable source image and no parameters.

Figure 7-12 shows a simple example of an `Invert` operation.

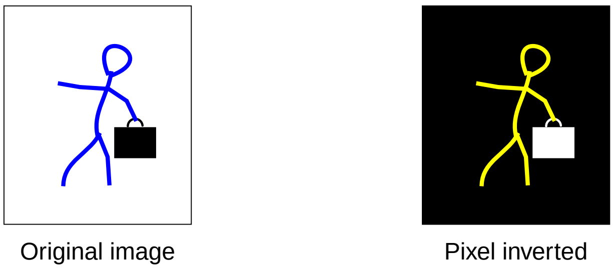


**Figure 7-12 Pixel Inverting**

### 7.10.2 Logarithmic Enhancement

Occasionally, it is desirable to quantize an image on a logarithmic scale rather than a linear scale. The human eye has a logarithmic intensity response but some images are digitized by equipment that quantizes the samples on a linear scale. To make the image better for use by a human observer, these images may be made to have a logarithmic response by the `Log` operation.

The `Log` operation takes the logarithm of the pixel values of the source image. The pixel values of the destination image are defined by the following pseudocode:

```
dst[x][y][b] = java.lang.Math.log(src[x][y][b])
```

For integral image data types, the result is rounded and clamped as needed. For all integral data types, the log of 0 is set to 0. For signed integral data types (`short` and `int`), the log of a negative pixel value is set to –1. For all floating point data types (`float` and `double`), the log of 0 is set to –`Infinity`, and the log of a negative pixel value is set to `NaN`.

The `Log` operation takes one rendered or renderable source image and no parameters.

Listing 7-11 shows a code sample for a `Log` operation.

**Listing 7-11 Example Log Operation**

```
// Create the ParameterBlock specifying the source image.
pb = new ParameterBlock();
pb.addSource(image);

// Create the Log operation.
RenderedImage dst = JAI.create("log", pb);
```

## 7.11 Dual Image Pixel Point Processing

The previous section described pixel point operations for single images. This section deals with pixel point processing on two images, also known as *dual-image point processing*. Dual-image point processing maps two pixel brightnesses, one from each image, to an output image.

JAI supports the following dual-image pixel point operations:

- Overlay images (`Overlay` operation)
- Image compositing (`Composite` operation)

### 7.11.1 Overlay Images

The `Overlay` operation takes two rendered or renderable source images, and overlays the second source image on top of the first source image. Usually, the images are identical scenes, but may have been acquired at different times through different spectral filters.

The two source images must have the same data type and number of bands. However, their `SampleModel` types may differ. The destination image will always have the same bounding rectangle as the first source image, that is, the image on the bottom, and the same data type and number of bands as the two source images. If the two source images don't intersect, the destination will be the same as the first source.

The `Overlay` operation is defined by the following pseudocode:

```
if (srcs[1] contains the point (x, y)) {
    dst[x][y][b] = srcs[1][x][y][b];
} else {
    dst[x][y][b] = srcs[0][x][y][b];
}
```

The `Overlay` operation takes two rendered or renderable source images and no parameters.

## 7.11.2 Image Compositing

The `Composite` operation merges unrelated objects from two images. The result is a new image that didn't exist before. The `Composite` operation combines two images based on their alpha values at each pixel. This is done on a per-band basis, and the source images are expected to have the same number of bands and the same data type. The destination image has the same data type as the two sources, but one extra band than the source images, which represents the result alpha channel.

The destination pixel values may be viewed as representing a fractional pixel coverage or transparency factor. Specifically, the `Composite` operation implements the Porter-Duff "over" rule[1], in which the output color of a pixel with source value and alpha tuples ($A$, $a$) and ($B$, $b$) is given by:

$$a*A + (1 - a)*(b*B)$$

The output alpha value is given by:

$$a + (1 - a)*b$$

For premultiplied sources tuples ($a*A$, $a$) and ($b*B$, $b$), the premultiplied output value is simply:

$$(a*A) + (1 - a)*(b*B)$$

1. See *Computer Graphics*, July 1984 pp. 253–259.

The color channels of the two source images are supplied via `source1` and `source2`. The two sources must either both be pre-multiplied by alpha or not. Alpha channel should not be included in `source1` and `source2`.

The `Composite` operation takes two rendered or renderable source images and four parameters:

| Parameter | Type | Description |
|---|---|---|
| `source1Alpha` | `PlanarImage` | An alpha image to override the alpha for the first source. |
| `source2Alpha` | `PlanarImage` | An alpha image to override the alpha for the second source. |
| `alphaPremultiplied` | `Boolean` | True if alpha has been premultiplied to both sources and the destination. |
| `destAlpha` | `Integer` | Indicates if the destination image should include an extra alpha channel, and if so, whether it should be the first or last band. One of: `CompositeDescriptor.DESTINATION_ALPHA_FIRST` `CompositeDescriptor.DESTINATION_ALPHA_LAST` `CompositeDescriptor.NO_DESTINATION_ALPHA` |

The alpha channel of the first source images must be supplied via the `source1Alpha` parameter. This parameter may not be null. The alpha channel of the second source image may be supplied via the `source2Alpha` parameter. This parameter may be null, in which case the second source is considered completely opaque. The alpha images should be single-banded, and have the same data type as the source image.

The `alphaPremultiplied` parameter indicates whether or not the supplied alpha image is premultiplied to both the source images.

The destination image is the combination of the two source images. It has the color channels and one additional alpha channel (the band index depends on the `alphaFirst` parameter). Whether the alpha value is pre-multiplied to the color channels also depends on the value of `alphaPremultiplied` (pre-multiplied if true).

Listing 7-12 shows a code sample for a composite operation.

**Listing 7-12 Example Composite Operation**

```
    // Get the first image.
    pb = new ParameterBlock();
    pb.add(s1);
    RenderedImage src1 = (RenderedImage)JAI.create("jpeg", pb);
```

**Listing 7-12 Example Composite Operation (Continued)**

```
// Get the second image
pb = new ParameterBlock();
pb.add(s2);
RenderedImage src2 = (RenderedImage)JAI.create("jpeg", pb);

// Create the ParameterBlock
pb = new ParameterBlock();
pb.addSource(src1);
pb.addSource(src2);
pb.add(new Boolean(false));
pb.add(new Boolean(false));

// Create the composite operation.
RenderedImage dst = (RenderedImage)JAI.create("composite", pb);
```

## 7.12 Thresholding

Thresholding, also known as *binary contrast enhancement*, provides a simple means of defining the boundaries of objects that appear on a contrasting background. The `Threshold` operation takes one rendered image, and maps all the pixels of this image whose values fall within a specified range to a specified constant. The range is specified by a low value and a high value.

The pixel values of the destination image are defined by the following pseudocode:

```
lowVal = (low.length < dstNumBands) ?
          low[0] : low[b];
highVal = (high.length < dstNumBands) ?
           high[0] : high[b];
const = (constants.length < dstNumBands) ?
         constants[0] : constants[b];

if (src[x][y][b] >= lowVal && src[x][y][b] <= highVal) {
    dst[x][y][b] = const;
} else {
    dst[x][y][b] = src[x][y][b];
}
```

The `Threshold` operation takes one rendered or renderable source image and three parameters:

| Parameters | Type | Description |
|---|---|---|
| `low` | `double[]` | The low value. |
| `high` | `double[]` | The high value |
| `constants` | `double[]` | The constant the pixels are mapped to. |

If the number of elements supplied via the `high`, `low`, and `constants` arrays are less than the number of bands of the source image, the element from entry 0 is applied to all the bands. Otherwise, the element from a different entry is applied to its corresponding band.

The `low` parameter defines the lower bound for the `threshold` operation for each band of the image. The operation will affect only values greater than or equal to `low[0]` in band 0, only values greater than or equal to `low[1]` in band 1, and so on. The `high` parameter defines the upper bound for the `threshold` operation for each band of the image.

A common way to arrive at the optimal values for the `low` and `high` parameters is to perform an `extrema` operation on the image (see Section 9.3, "Finding the Extrema of an Image").

Listing 7-13 shows a code sample for a `threshold` operation in which the three parameters are passed as arguments to the operation.

**Listing 7-13 Example Threshold Operation**

```
   // Set up the operation parameters.
   PlanarImage src, dst;
   Integer [] low, high, map;
   int bands;

   low  = new Integer[bands];
   high = new Integer[bands];
   map  = new Integer[bands];

   for (int i = 0; i < bands; i++) {
      low[i]  = new Integer(args[1]);
      high[i] = new Integer(args[2]);
      map[i]  = new Integer(args[3]);
   }
```

**Listing 7-13 Example Threshold Operation (Continued)**

```
   // Create the threshold operation.
   pb = new ParameterBlock();
   pb.addSource(src);
   pb.add(low);
   pb.add(high);
   pb.add(map);
   RenderedImage dst = JAI.create("threshold", pb);
```

CHAPTER 8

# Geometric Image Manipulation

THIS chapter describes the basics of JAI's geometric image manipulation functions. The geometric image manipulation operators are all part of the `javax.media.operator` package.

## 8.1 Introduction

The JAI geometric image manipulation functions are:

- Geometric transformation (`Translate`, `Scale`, `Rotate`, and `Affine`)
- Perspective transformation (`PerspectiveTransform`)
- Transposing (`Transpose`)
- Shearing (`Shear`)
- Warping (`Warp`, `WarpAffine`, `WarpPerspective`, `WarpPolynomial`, `WarpGeneralPolynomial`, `WarpQuadratic`, and `WarpOpImage`)

Most of these geometric functions require an interpolation argument, so this chapter begins with a discussion of interpolation.

## 8.2 Interpolation

Several geometric image operations, such as `Affine`, `Rotate`, `Scale`, `Shear`, `Translate`, and `Warp`, use a geometric transformation to compute the coordinate of a source image point for each destination image pixel. In most cases, the destination pixel does not lie at a source pixel location, but rather lands somewhere between neighboring pixels. The estimated value of each pixel is set in a process called interpolation or *image resampling*.

Resampling is the action of computing a pixel value at a possibly non-integral position of an image. The image defines pixel values at integer lattice points, and it is up to the resampler to produce a reasonable value for positions not falling on the lattice. A number of techniques are used in practice, the most common being the following:

- Nearest-neighbor, which simply takes the value of the closest lattice point
- Bilinear, which interpolates linearly between the four closest lattice points
- Bicubic, which applies a piecewise polynomial function to a $4 \times 4$ neighborhood of nearby points

The area over which a resampling function needs to be computed is referred to as its *support*; thus the standard resampling functions have supports of 1, 4, and 16 pixels respectively. Mathematically, the ideal resampling function for a band-limited image (one containing no energy above a given frequency) is the sinc function, equal to sin(x)/x. This has practical limitations, in particular its infinite support, which lead to the use of the standard approximations described above.

In interpolation, each pixel in a destination image is located with integer coordinates at a distinct point *D* in the image plane. The geometric transform *T* identifies each destination pixel with a corresponding point *S* in the source image. Thus, *D* is the point that *T* maps to *S*. In general, *S* doesn't correspond to a single source pixel; that is, it doesn't have integer coordinates. Therefore, the value assigned to the pixel *D* must be computed as an interpolated combination of the pixel values closest to *S* in the source image.

For most geometric transformations, you must specify the interpolation method to be used in calculating destination pixel values. Table 8-1 lists the names used to call the interpolation methods.

**Table 8-1 Interpolation Types**

| Name | Description |
|---|---|
| `INTERP_NEAREST` | Nearest-neighbor interpolation. Assigns to point D in the destination image the value of the pixel nearest S in the source image. See Section 8.2.1, "Nearest-neighbor Interpolation." |
| `INTERP_BILINEAR` | Bilinear interpolation. Assigns to Point D in the destination a value that is a bilinear function of the four pixels nearest S in the source image. See Section 8.2.2, "Bilinear Interpolation." |

**Table 8-1 Interpolation Types (Continued)**

| Name | Description |
|---|---|
| `INTERP_BICUBIC` | Bicubic interpolation. Assigns to point D in the destination image a value that is a bicubic function of the 16 pixels nearest S in the source image.Section 8.2.3, “Bicubic Interpolation.” |
| `INTERP_BICUBIC2` | Bicubic2 interpolation. Similar to Bicubic, but uses a different polynomial function. See Section 8.2.4, “Bicubic2 Interpolation.” |

Occasionally, these four options do not provide sufficient quality for a specific operation and a more general form of interpolation is called for. The more general form of interpolation, called *table interpolation* uses tables to store the interpolation kernels. See Section 8.2.5, “Table Interpolation.”

Other interpolation functions may be required to solve problems other than the resampling of band-limited image data. When shrinking an image, it is common to use a function that combines area averaging with resampling to remove undesirable high frequencies as part of the interpolation process. Other application areas may use interpolation functions that operate under other assumptions about image data, such as taking the maximum value of a $2 \times 2$ neighborhood. The `Interpolation` class provides a framework in which a variety of interpolation schemes may be expressed.

Many Interpolations are separable, that is, they may be equivalently rewritten as a horizontal interpolation followed by a vertical one (or vice versa). In practice, some precision may be lost by the rounding and truncation that takes place between the passes. The `Interpolation` class assumes separability and implements all vertical interpolation methods in terms of corresponding horizontal methods, and defines `isSeparable` to return true. A subclass may override these methods to provide distinct implementations of horizontal and vertical interpolation. Some subclasses may implement the two-dimensional interpolation methods directly, yielding more precise results, while others may implement these using a two-pass approach.

When interpolations that require padding the source such as Bilinear or Bicubic interpolation are specified, the boundary of the source image needs to be extended such that it has the extra pixels needed to compute all the destination pixels. This extension is performed via the `BorderExtender` class. The type of border extension can be specified as a `RenderingHint` to the `JAI.create` method. If no border extension type is provided, a default extension of `BorderExtender.BORDER_COPY` will be used to perform the extension. See Section 3.7.3, “Rendering Hints.”

Listing 8-1 shows a code sample for a `rotate` operation. First, the type of interpolation is specified (`INTERP_NEAREST` in this example) using the `Interpolation.create` method. Next, a parameter block is created and the interpolation method is added to the parameter block, as are all the other parameters required by the operation. Finally, a `rotate` operation is created with the specified parameter block.

**Listing 8-1 Example Using Nearest-neighbor Interpolation**

```
// Specify the interpolation method to be used
interp = Interpolation.create(Interpolation.INTERP_NEAREST);

// Create the parameter block and add the interpolation to it
ParameterBlock pb = new ParameterBlock();
pb.addSource(im);         // The source image
pb.add(0.0F);             // The x origin to rotate about
pb.add(0.0F);             // The y origin to rotate about
pb.add(theta);            // The rotation angle in radians
pb.add(interp);           // The interpolation method

// Create the rotation operation and include the parameter
// block
RenderedOp op JAI.create("rotate", pb, null);
```

The `Interpolation` class provides methods for the most common cases of $2 \times 1$, $1 \times 2$, $4 \times 1$, $1 \times 4$, $2 \times 2$, and $4 \times 4$ input grids, some of which are shown in Figure 8-1. These methods are defined in the superclass (`Interpolation`) to package their arguments into arrays and forward the call to the array versions, to simplify implementation. These methods should be called only on `Interpolation` objects with the correct width and height. In other words, an implementor of an `Interpolation` subclass may implement `interpolateH(int s0, int s1, int xfrac)`, assuming that the interpolation width is in fact equal to 2, and does not need to enforce this constraint.

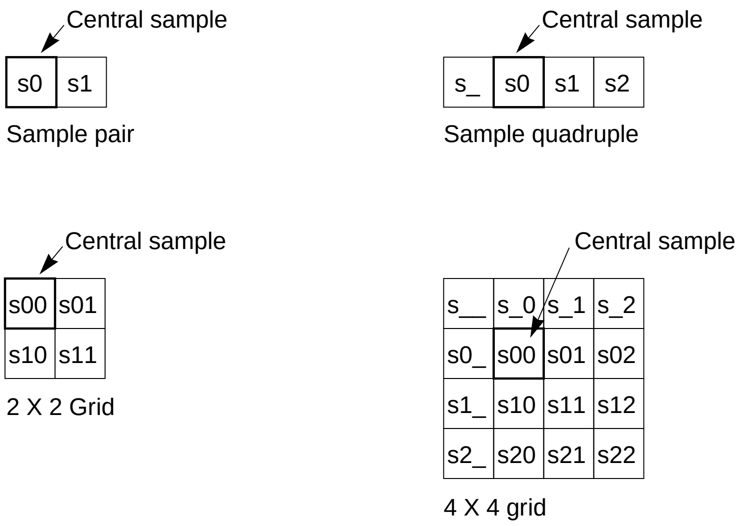


**Figure 8-1 Interpolation Samples**

Another possible source of inefficiency is the specification of the subsample position. When interpolating integral image data, JAI uses a fixed-point subsample position specification, that is, a number between 0 and ($2^n$ – 1) for some small value of *n*. The value of *n* in the horizontal and vertical directions may be obtained by calling the `getSubsampleBitsH` and `getSubsampleBitsV` methods. In general, code that makes use of an externally-provided `Interpolation` object must query that object to determine its desired positional precision.

For `float` and `double` images, JAI uses a `float` between 0.0F and 1.0F (not including 1.0F) as a positional specifier in the interest of greater accuracy.

**API:** `javax.media.jai.Interpolation`

- `static Interpolation getInstance(int type)`

  creates an interpolation of one of the standard types, where `type` is one of `INTERP_NEAREST`, `INTERP_BILINEAR`, `INTERP_BICUBIC`, or `INTERP_BICUBIC_2`.

- `int interpolate(int[][] samples, int xfrac, int yfrac)`

  performs interpolation on a two-dimensional array of integral samples. By default, this is implemented using a two-pass approach.

| | | |
|---|---|---|
| *Parameters*: | `samples` | A two-dimensional array of ints. |
| | `xfrac` | The *x* subsample position, multiplied by $2^{\text{subsampleBits}}$. |
| | `yfrac` | The *y* subsample position, multiplied by $2^{\text{subsampleBits}}$. |

- `float interpolate(float[][] samples, float xfrac, float yfrac)`

  performs interpolation on a two-dimensional array of floating-point samples. This is the same as the above method, only using float values instead of ints.

-
  ```
  double interpolate(double[][] samples, float xfrac,
      float yfrac)
  ```

  Performs interpolation on a 2-dimensional array of double samples.

-
  ```
  int interpolate(int s00, int s01, int s10, int s11, int xfrac,
      int yfrac)
  ```

  performs interpolation on a 2 × 2 grid of integral samples. It should only be called if width == height == 2 and leftPadding == topPadding == 0.

  The `s00`, `s01`, `s10`, and `s11` parameters are the sample values (see the 2 × 2 grid illustration in Figure 8-1).

-
  ```
  float interpolate(float s00, float s01, float s10, float s11,
      float xfrac, float yfrac)
  ```

  performs interpolation on a 2 × 2 grid of integral samples. This is the same as the above method, only using float values instead of ints.

-
  ```
  double interpolate(double s00, double s01, double s10, double
      s11, float xfrac, float yfrac)
  ```

  performs interpolation on a 2 × 2 grid of double samples.

-
  ```
  int interpolate(int s__, int s_0, int s_1, int s_2, int s0_,
      int s00, int s01, int s02, int s1_, int s10, int s11,
      int s12, int s2_, int s20, int s21, int s22, int xfrac,
      int yfrac)
  ```

  performs interpolation on a 4 × 4 grid of integral samples. It should only be called if width == height == 4 and leftPadding == topPadding == 1.

  The `s__, through` `s22` parameters are the sample values (see the 4 × 4 grid illustration in Figure 8-1).

-
  ```
  float interpolate(float s__, float s_0, float s_1, float s_2,
      float s0_, float s00, float s01, float s02, float s1_,
      float s10, float s11, float s12, float s2_, float s20,
      float s21, float s22, float xfrac, float yfrac)
  ```
  performs interpolation on a $4 \times 4$ grid of integral samples. This is the same as the above method, only using float values instead of ints.

- `abstract int getSubsampleBitsH()`

  returns the number of bits used to index subsample positions in the horizontal direction. All integral `xfrac` parameters should be in the range of 0 to $2^{(\text{getSubsampleBitsH})} - 1$.

- `int getSubsampleBitsV()`

  returns the number of bits used to index subsample positions in the vertical direction. All integral `yfrac` parameters should be in the range of 0 to $2^{(\text{getSubsampleBitsV})} - 1$.

### 8.2.1 Nearest-neighbor Interpolation

Nearest-neighbor interpolation, also known as zero-order interpolation, is the fastest interpolation method, though it can produce image artifacts called *jaggies* or *aliasing error*. Jaggies are image artifacts in which the straight edges of objects appear to be rough or jagged.

Nearest-neighbor interpolation simply assigns to point *D* in the destination image the value of the pixel nearest *S* in the source image.

Neighborhoods of sizes $2 \times 1$, $1 \times 2$, $2 \times 2$, $4 \times 1$, $1 \times 4$, $4 \times 4$, $N \times 1$, and $1 \times N$, that is, all the `interpolate()` methods defined in the `Interpolation` class, are supported in the interest of simplifying code that handles a number of types of interpolation. In each case, the central sample is returned and the rest are ignored.

**API:** `javax.media.jai.InterpolationNearest`

- `InterpolationNearest()`

  constructs an `InterpolationNearest`. The return value of `getSubsampleBitsH()` and `getSubsampleBitsV()` will be 0.

### 8.2.2 Bilinear Interpolation

Bilinear interpolation, also known as first-order interpolation, linearly interpolates pixels along each row of the source image, then interpolates along the columns. Bilinear interpolation assigns to Point *D* in the destination a value that is a bilinear function of the four pixels nearest *S* in the source image.

Bilinear interpolation results in an improvement in image quality over nearest-neighbor interpolation, but may still result in less-than-desirable smoothing effects.

Bilinear interpolation requires a neighborhood extending one pixel to the right and below the central sample. If the subsample position is given by (*u*, *v*), the resampled pixel value will be:

$$(1 - v) * [(1 - u) * p00 + u * p01] + v * [(1 - u) * p10 + u * p11]$$

**API:** `javax.media.jai.InterpolationBilinear`

- `InterpolationBilinear(int subsampleBits)`

  constructs an `InterpolationBilinear` object with a given subsample precision, in bits.

  *Parameters*: `subsampleBits` The subsample precision.

- `InterpolationBilinear()`

  constructs an `InterpolationBilinear` object with the default subsample precision.

### 8.2.3 Bicubic Interpolation

Bicubic interpolation reduces resampling artifacts even further by using the 16 nearest neighbors in the interpolation and by using bicubic waveforms rather than the linear waveforms used in bilinear interpolation. Bicubic interpolation preserves the fine detail present in the source image at the expense of the additional time it takes to perform the interpolation.

The bicubic interpolation routine assigns to point *D* in the destination image a value that is a bicubic function of the 16 pixels nearest *S* in the source image.

Bicubic interpolation performs interpolation using the following piecewise cubic polynomial:

$$
\begin{aligned}
r(x) &= (a+2)|x|^3 - (a+3)|x|^2 + 1, & 0 \le |x| < 1 \\
r(x) &= a|x|^3 - 5a|x|^2 + 8a|x| - 4a, & 1 \le |x| < 2 \\
r(x) &= 0, & \text{otherwise}
\end{aligned}
$$

with $a$ set to –0.5

Bicubic interpolation requires a neighborhood extending one sample to the left of and above the central sample, and two samples to the right of and below the central sample.

**API:** `javax.media.jai.InterpolationBicubic`

- `InterpolationBicubic(int subsampleBits)`

  constructs an `InterpolationBicubic` with a given subsample precision, in bits.

  *Parameters*: `subsampleBits` The subsample precision.

## 8.2.4 Bicubic2 Interpolation

Bicubic2 interpolation is basically the same as bicubic interpolation, but uses a different polynomial function. Bicubic2 interpolation uses the following piecewise cubic polynomial:

$$
\begin{aligned}
r(x) &= (a+2)|x|^3 - (a+3)|x|^2 + 1, & 0 \le |x| < 1 \\
r(x) &= a|x|^3 - 5a|x|^2 + 8a|x| - 4a, & 1 \le |x| < 2 \\
r(x) &= 0, & \text{otherwise}
\end{aligned}
$$

with $a$ set to –1.0

Bicubic interpolation requires a neighborhood extending one sample to the left of and above the central sample, and two samples to the right of and below the central sample.

**API:** `javax.media.jai.InterpolationBicubic2`

- `InterpolationBicubic2(int subsampleBits)`

  constructs an `InterpolationBicubic2` with a given subsample precision, in bits.

  *Parameters*: `subsampleBits` The subsample precision.

### 8.2.5 Table Interpolation

The previous-described types of interpolation, nearest-neighbor, bilinear, bicubic, and bicubic2, base the interpolation values on a relatively few pixels: one (nearest-neighbor), four (bilinear), or 16 (bicubic and bicubic2). Occasionally, these options don't provide sufficient quality for a specific operation and a general form of interpolation is called for. Table interpolation uses tables to store the interpolation kernels. The set of subpixel positions is broken up into a fixed number of "bins" and a distinct kernel is used for each bin. The number of bins must be a power of two.

An `InterpolationTable` defines a separable interpolation, with a separate set of kernels for the horizontal and vertical dimensions. The number of bins within each kernel may vary between the two dimensions. The horizontal and vertical kernels may be unique or the same. That is, you can either construct two separate kernels or use the same kernel for both the horizontal and vertical interpolation.

The kernels are stored in both floating- and fixed-point form. The fixed point representation has a user-specified fractional precision. You must specify an appropriate level of precision that will not cause overflow when accumulating the results of a convolution against a set of source pixels, using 32-bit integer arithmetic.

To use table interpolation, create an `InterpolationTable` with either identical horizontal and vertical resampling kernels or with different horizontal and vertical resampling kernels. The table forms the kernels used for the interpolation.

During a table interpolation operation, the key value of the resampling kernel, generally the center value, is laid over the source image pixel to be processed. The other kernel values lie over neighboring pixels much like a conventional $M \times N$ kernel operation. Each source image pixel that is covered by the kernel is then multiplied by the kernel value that lies over it. The multiplication products are then summed together and this sum becomes the pixel value in the destination.

To save memory space and computation time, the table interpolation operation does not use a conventional $M \times N$ kernel. Instead, the operation uses separate horizontal and vertical vector arrays (essentially, $M \times 1$ and $N \times 1$) to calculate the same values that a $M \times N$ kernel would calculate. The vector arrays allow you to provide fewer data elements for the kernel values. This is particularly significant for large tables with many subsamples.

The basic format for the `InterpolationTable` constructor is:

```
   InterpolationTable(int leftPadding, int topPadding, int width,
                      int height, int subsampleBitsH,
                      int subsampleBitsV, int precisionBits,
                      float[] dataH, float[] dataV)
```

The parameters to the constructor are described in the following paragraphs.

### 8.2.5.1 Padding

The `leftPadding` and `topPadding` parameters define the location of the central sample or key value, relative to the left and top of the horizontal and vertical kernels, respectively. These parameters actually define the number of samples to the left of or above the central sample, as shown in Figure 8-2.

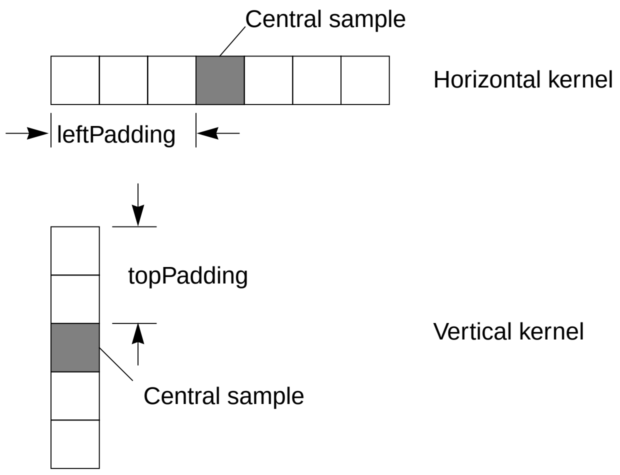


**Figure 8-2 Table Interpolation Padding**

### 8.2.5.2 Width and Height

The `width` and `height` parameters define the size of the horizontal and vertical kernels, respectively. These parameters specify the number of data elements in each subsample of the kernel. The horizontal and vertical tables can have different kernel sizes. For the two examples shown in Figure 8-2, the `width` parameter would be 7, the `height` parameter would be 5.

The `getWidth` and `getHeight` methods return the number of samples required for horizontal and vertical resampling, respectively.

### 8.2.5.3 Subsample Bits

The `subsampleBitsH` and `subsampleBitsV` parameters define the number of bins used to describe the horizontal and vertical subpixel positions, respectively. The number of bins must be a power of two, so the values are integers expressed as the $\log_2$ of the number of horizontal or vertical subsample positions, respectively. The value `subsampleBitsH` = 1 defines two subsamples per horizontal sample, `subsampleBitsH` = 2 defines four subsamples per sample, and so on.

For each subsample, you must define separate kernel data. Typically, the kernel values for each subsample are weighted according to the subsample location's proximity to the pixels used in the calculation. The closer a pixel is to the subsample location, the more weight it carries in the kernel.

Figure 8-3 shows how the interpolation tables are used to determine which kernel applies to a particular subsample location. The figure shows a subsample of 4 in both the horizontal and vertical directions.

Typically, the kernel values for each subsample are weighted according to the subsample location's proximity to the pixels used in the calculation. The closer a pixel is to the subsample location, the more weight it carries in the kernel.

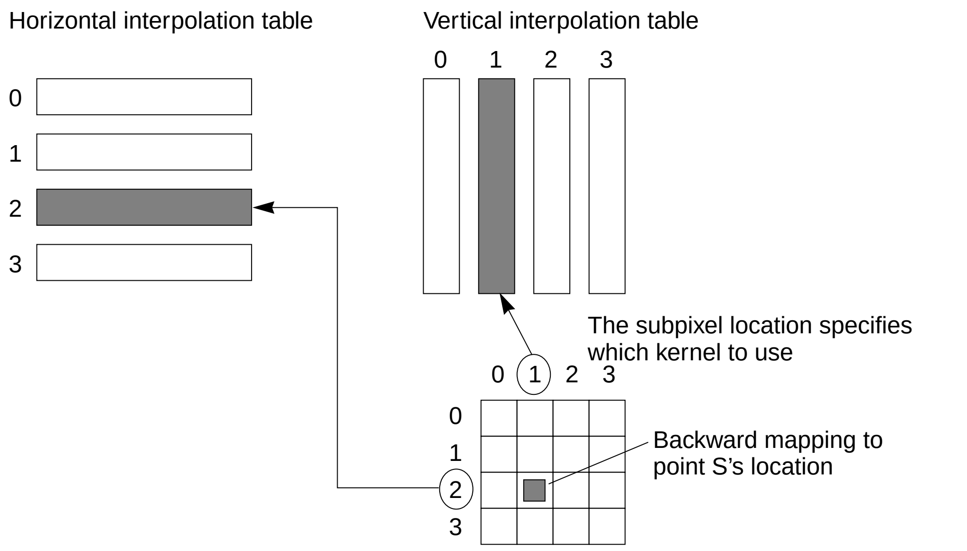

Horizontal interpolation table
0
1
2
3
Vertical interpolation table
0
1
2
3
The subpixel location specifies which kernel to use
0
1
2
3
0
1
2
3
Backward mapping to point S's location

**Figure 8-3 Table Interpolation Backwards Mapping**

### 8.2.5.4 Precision

The `precisionBits` parameter defines the number of fractional bits to be used when resampling integral sample values. The same precision value is used for both horizontal and vertical resampling. It is important to choose an appropriate level of precision that will not cause overflow when accumulating the results of a convolution against a set of source pixels, using 32-bit integer arithmetic.

### 8.2.5.5 Kernel Data

The kernel data for each table is an array of floating point numbers. The `dataH` and `dataV` parameters specify the floating-point data values for the horizontal and vertical kernels, respectively. The number of data elements in the kernel is:

`width` $\times 2^{\text{subsampleBitsH}}$ for `dataH`

`height` $\times 2^{\text{subsampleBitsV}}$ for `dataV`

For a two-element kernel size with eight subsample bins (`subsampleBits` = 4), you need to define an array of 16 floating point values. The first two values define the kernel for the first subsample, the second two values define the kernel for the second subsample, and so on. For example:

```
float[] kernelData = {1.0,    0.0,
                      0.875,  0.125,    // 7/8,  1/8
                      0.75,   0.25,     // 6/8,  2/8
                      0.625,  0.375,    // 5/8,  3/8
                      0.5,    0.5,      // 4/8,  4/8
                      0.375,  0.625,    // 3/8,  5/8
                      0.25,   0.75,     // 2/8,  6/8
                      0.125,  0.875 };  // 1/8,  7/8
```

The above example creates a bilinear interpolation table with eight subsamples. The kernel values indicate how much influence the source image pixels will have on the destination value. A kernel value of 1 indicates that a source pixel completely determines the value of the destination pixel. A kernel value of 0 indicates that the source pixel has no influence on the destination value.

To preserve the source image's intensity in the destination image, the sum of the data values in each interpolation kernel should equal one. If the kernel values sum to greater than one, the destination image's intensity will be increased. Conversely, if the kernel values sum to less than one, the destination image's intensity will be decreased.

If a value of `null` is given for `dataV`, the `dataH` table data is used for vertical interpolation as well, and the `topPadding`, `height`, and `subsampleBitsV` parameters are ignored.

**API:** `javax.media.jai.InterpolationTable`

-
  ```
  InterpolationTable(int padding, int width, int subsampleBits,
      int precisionBits, float[] data)
  ```

  constructs an `InterpolationTable` with identical horizontal and vertical resampling kernels.

| *Parameters*: | | |
|---|---|---|
| | `padding` | The number of samples to the left or above the central sample to be used during resampling. |
| | `width` | The width or height of a resampling kernel. |
| | `subsample-Bits` | The $\log_2$ of the number of subsample bins. |
| | `precision-Bits` | The number of bits of fractional precision to be used when resampling integral sample values. |
| | `data` | The kernel entries, as a float array of `width`$*2^{subsampleBitsH}$ entries. |

-
  ```
  InterpolationTable(int padding, int width, int subsampleBits,
      int precisionBits, double[] data)
  ```

  constructs an InterpolationTable with identical horizontal and vertical resampling kernels.

-
  ```
  InterpolationTable(int padding, int width, int subsampleBits,
      int precisionBits, int[] data)
  ```

  Constructs an InterpolationTable with identical horizontal and vertical resampling kernels.

-
  ```
  InterpolationTable(int leftPadding, int topPadding, int width,
      int height, int subsampleBitsH, int subsampleBitsV,
      int precisionBits, float[] dataH, float[] dataV)
  ```

  constructs an `InterpolationTable` with specified horizontal and vertical extents (support), number of horizontal and vertical bins, fixed-point fractional precision, and kernel entries. The kernel data values are organized as $2^{\text{subsampleBits}}$ entries each containing `width` floats.

| *Parameters*: | | |
|---|---|---|
| | `leftPadding` | The number of samples to the left of the central sample to be used during horizontal resampling. |
| | `topPadding` | The number of samples above the central sample to be used during vertical resampling. |
| | `width` | The width of a horizontal resampling kernel. |
| | `height` | The height of a vertical resampling kernel. Ignored if `dataV` is null. |
| | `subsample-BitsH` | The $\log_2$ of the number of horizontal subsample bins. |
| | `subsample-BitsV` | The $\log_2$ of the number of vertical subsample bins. Ignored if `dataV` is null. |
| | `precision-Bits` | The number of bits of fractional precision to be used when resampling integral sample values. The same value is used for both horizontal and vertical resampling. |
| | `dataH` | The horizontal table entries, as a float array of $2^{\text{subsampleBitsH}}$ entries each of length `width`. |
| | `dataV` | The vertical table entries, as a float array of $2^{\text{subsampleBitsV}}$ entries each of length `height`, or null. If null, the `dataH` table is used for vertical interpolation as well and the `topPadding`, `height`, and `subsampleBitsV` parameters are ignored. |

-
  ```
  InterpolationTable(int leftPadding, int topPadding, int width,
      int height, int subsampleBitsH, int subsampleBitsV,
      int precisionBits, double[] dataH, double[] dataV)
  ```

  constructs an `InterpolationTable` with specified horizontal and vertical extents (support), number of horizontal and vertical bins, fixed-point fractional precision, and kernel entries.

-
  ```
  InterpolationTable(int leftPadding, int topPadding, int width,
      int height, int subsampleBitsH, int subsampleBitsV,
      int precisionBits, int[] dataH, int[] dataV)
  ```
  constructs an InterpolationTable with specified horizontal and vertical extents (support), number of horizontal and vertical bins, fixed-point fractional precision, and int kernel entries.

### 8.2.5.6 Additional Interpolation Table-related Methods

The `InterpolationTable` class provides several methods for retrieving an interpolation table's kernel data values, subsample size, and precision.

**API:** `javax.media.jai.InterpolationTable`

- `int getSubsampleBitsH()`

  returns the number of bits used to index subsample positions in the horizontal direction.

- `int getSubsampleBitsV()`

  returns the number of bits used to index subsample positions in the vertical direction.

- `int getPrecisionBits()`

  returns the number of bits of fractional precision used to store the fixed-point table entries.

- `int getLeftPadding()`

  returns the number of bits of fractional precision used to store the fixed-point table entries.

- `int getTopPadding()`

  returns the number of samples required above the center.

- `int getWidth()`

  returns the number of samples required for horizontal resampling.

- `int getHeight()`

  returns the number of samples required for vertical resampling.

- `int[] getHorizontalTableData()`

  returns the integer (fixed-point) horizontal table data.

- `int[] getVerticalTableData()`
  returns the integer (fixed-point) vertical table data.

- `float[] getHorizontalTableDataFloat()`
  returns the floating-point horizontal table data.

- `float[] getVerticalTableDataFloat()`
  returns the floating-point vertical table data.

- `double[] getHorizontalTableDataDouble()`
  returns the double horizontal table data.

- `double[] getVerticalTableDataDouble()`
  returns the double vertical table data.

## 8.3 Geometric Transformation

Geometric transformations provide the ability to reposition pixels within an image. Pixels may be relocated from their (*x*,*y*) spatial coordinates in the source image to new coordinates in the destination. Geometric transformations are used, for example, to move (translate), rotate, and scale the geometry of an image. A general type of geometric transformation, warp, is discussed later in this chapter (see Section 8.7, “Warping”).

Geometric transformations are used to register multiple images, correct geometric distortions introduced in the image acquisition process, or to add visual effects. The geometric transformation operations discussed here include:

- Translation (`Translate`) – moves an image up, down, left, or right
- Scaling (`Scale`) – enlarges or shrinks an image
- Rotation (`Rotate`) – rotates an image about a given point
- Affine (`Affine`) – includes translation, scaling, and rotation in one operation

All transformation operations are performed by moving pixel values from their original spatial coordinates to new coordinates in the destination image. Every pixel in the source image is passed through this transformation, creating a geometrically-transformed output pixel location. Each pixel of the source image is transformed, pixel by pixel, to its new location in the destination image.

With a very few exceptions, all transformations result in some output pixel locations being missed because no input pixels were transformed there. The missed locations will be devoid of any pixel values and result in a black hole in the destination image. To overcome this problem, intermediate pixel values are estimated through interpolation (See “Interpolation” on page 249). One of four `interpolation` methods may be selected:

| **interpolation Methods** | **Description** |
|---|---|
| `INTERP_NEAREST` | Use nearest-neighbor interpolation |
| `INTERP_BILINEAR` | Use bilinear interpolation |
| `INTERP_BICUBIC` | Use bicubic interpolation |
| `INTERP_BICUBIC2` | Use bicubic2 interpolation (uses a different polynomial function) |

### 8.3.1 Translation Transformation

Image translation is the spatial shifting of an image up, down, left, or right. The relationships between the source and destination image coordinates are given by the following equation:

$$\begin{aligned} x_D &= x'_S + t_x \\ y_D &= y'_S + t_y \end{aligned} \tag{8.1}$$

where:

$x_D$ and $y_D$ are the integer pixel coordinates of the destination image

$t_x$ and $t_y$ are the translation values

$x'_S$ and $y'_S$ denote the source image point from which the pixel estimate is computed.

Translation is often used to register multiple images geometrically. The translation is often carried out to align the images before performing a combination operation, such as image addition, subtraction, division, or compositing.

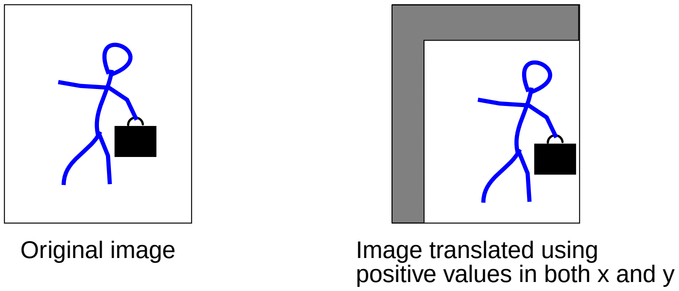


**Figure 8-4 Translate Operation**

The `translate` operation takes one rendered or renderable source image and three parameters:

| Parameters | Type | Description |
|---|---|---|
| `xTrans` | `Float` | The displacement in the *x* direction. The default value is 0.0F. |
| `yTrans` | `Float` | The displacement in the *y* direction. The default value is 0.0F. |
| `interpolation` | `Interpolation` | The interpolation method for resampling. One of `INTERP_NEAREST`, `INTERP_BILINEAR`, `INTERP_BICUBIC`, or `INTERP_BICUBIC2`. The default value is `null`. |

The `xTrans` parameter corresponds to $t_x$ and the `yTrans` parameter corresponds to $t_y$ in equation 8.1. If `xTrans` is positive, the translation is to the right; if negative, to the left. If `yTrans` is positive, the translation is down; if negative, upward. If both `xTrans` and `yTrans` are integral, the operation simply *wraps* its source image to change the image's position in the coordinate plane.

When interpolations that require padding the source such as bilinear or bicubic interpolation are specified, the boundary of the source image needs to be extended such that it has the extra pixels needed to compute all the destination pixels. This extension is performed via the `BorderExtender` class. The type of border extension can be specified as a `RenderingHint` to the `JAI.create` method. If no border extension type is provided, a default extension of `BorderExtender.BORDER_COPY` will be used to perform the extension. See Section 3.7.3, "Rendering Hints."

Listing 8-2 shows a code sample for a translate operation using nearest-neighbor interpolation.

**Listing 8-2 Example Translate Operation**

```
// Create a ParameterBlock and specify the source and
// parameters.
ParameterBlock pb = new ParameterBlock();
     pb.addSource(im);                    // The source image
     pb.add((float)Math.max(-mx, 0));    // The x translation
     pb.add((float)Math.max(-my, 0));    // The y translation
     pb.add(new InterpolationNearest()); // The interpolation

// Create the translate operation
im = JAI.create("translate", pb, null);
```

## 8.3.2 Scaling Transformation

Scaling, also known as *minification* and *magnification*, enlarges or shrinks an image. An *x*-value defines the amount of scaling in the *x* direction, and a *y*-value defines the amount of scaling in the *y* direction. The `Scale` operation both translates and resizes.

Scaling is often used to geometrically register multiple images prior to performing a combination operation, such as image addition, subtraction, division, or compositing. Scaling can also be used to correct geometric distortions introduced in the image acquisition process, although the `Affine` operation ("Affine Transformation" on page 272) would be more suitable for this.

For each pixel (*x*, *y*) of the destination, the source value at the fractional subpixel position is constructed by means of an `Interpolation` object and written to the destination.

$$x' = \frac{x - \mathrm{xTrans}}{\mathrm{xScale}}$$

$$y' = \frac{y - \mathrm{yTrans}}{\mathrm{yScale}}$$

The `scale` operation takes one rendered or renderable source image and five parameters:

| Parameters | Type | Description |
|---|---|---|
| `xScale` | `Float` | The *x* scale factor. |
| `yScale` | `Float` | The *y* scale factor. |

| Parameters | Type | Description |
|---|---|---|
| `xTrans` | `Float` | The *x* translation. |
| `xTrans` | `Float` | The *y* translation. |
| `interpolation` | `Interpolation` | The interpolation method for resampling. One of `INTERP_NEAREST`, `INTERP_BILINEAR`, `INTERP_BICUBIC`, or `INTERP_BICUBIC2`. |

When applying scale factors (`xScale` and `yScale`) to a source image with width of `src_width` and height of `src_height`, the resulting image is defined to have the following dimensions:

```
dst_width = src_width * xScale
dst_height = src_height * yScale
```

Scale factors greater than 1.0 magnify the image; less than 1.0 minify the image. The `xTrans` parameter corresponds to $t_x$ and the `yTrans` parameter corresponds to $t_y$ in equation 8.1. If `xTrans` is positive, the translation is to the right; if negative, to the left. If `yTrans` is positive, the translation is down; if negative, upward.

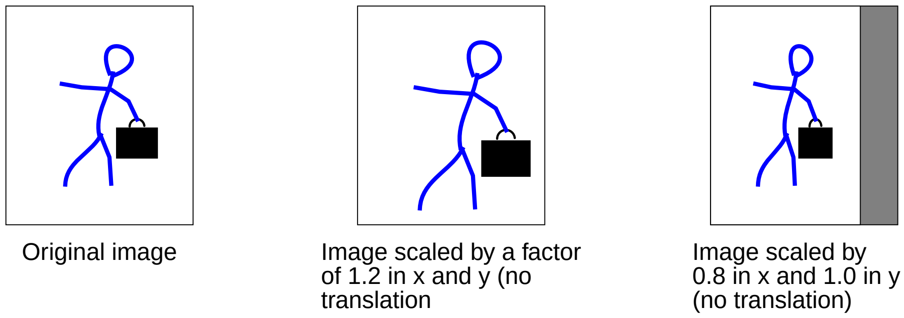

Original image
Image scaled by a factor of 1.2 in x and y (no translation
Image scaled by 0.8 in x and 1.0 in y (no translation)

**Figure 8-5 Scale Operation**

When interpolations that require padding the source such as Bilinear or Bicubic interpolation are specified, the boundary of the source image needs to be extended such that it has the extra pixels needed to compute all the destination pixels. This extension is performed via the `BorderExtender` class. The type of border extension can be specified as a `RenderingHint` to the `JAI.create` method. See Section 3.7.3, “Rendering Hints.”

If no Border Extension is specified, the source will not be extended. The scaled image size is still calculated according to the equation specified above. However since there isn’t enough source to compute all the destination pixels, only that subset of the destination image's pixels that can be computed will be written in the destination. The rest of the destination will not be written.

Listing 8-3 shows a code sample for a `Scale` operation using a scale factor of 1.2 and nearest-neighbor interpolation.

**Listing 8-3 Example Scale Operation**

```
// Create a ParameterBlock and specify the source and
// parameters
ParameterBlock pb = new ParameterBlock();
     pb.addSource(im);                    // The source image
     pb.add(1.2);                         // The xScale
     pb.add(1.2);                         // The yScale
     pb.add(0.0F);                        // The x translation
     pb.add(0.0F);                        // The y translation
     pb.add(new InterpolationNearest()); // The interpolation

// Create the scale operation
im = JAI.create("scale", pb, null);
```

## 8.3.3 Rotation Transformation

The `rotate` operation rotates an image about a given point by a given angle. Specified *x* and *y* values define the coordinate of the source image about which to rotate the image and a rotation angle in *radians* defines the angle of rotation about the rotation point. If no rotation point is specified, a default of (0,0) is assumed.

A negative rotation value rotates the image counter-clockwise, while a positive rotation value rotates the image clockwise.

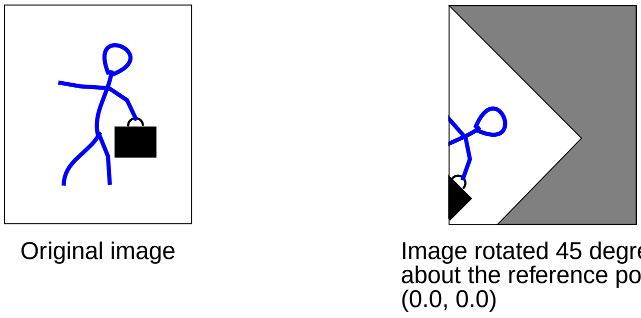

Original image
Image rotated 45 degrees about the reference point (0.0, 0.0)

**Figure 8-6 Rotate Operation**

The `rotate` operation takes one rendered or renderable source image and four parameters:

| Parameters | Type | Description |
|---|---|---|
| `xOrigin` | `Float` | The *x* origin to rotate about. |
| `yOrigin` | `Float` | The *y* origin to rotate about. |
| `angle` | `Float` | The rotation angle in radians. |
| `interpolation` | `Interpolation` | The interpolation method for resampling. One of `INTERP_NEAREST`, `INTERP_BILINEAR`, `INTERP_BICUBIC`, or `INTERP_BICUBIC2`. |

When interpolations that require padding the source such as Bilinear or Bicubic interpolation are specified, the boundary of the source image needs to be extended such that it has the extra pixels needed to compute all the destination pixels. This extension is performed via the `BorderExtender` class. The type of border extension can be specified as a `RenderingHint` to the `JAI.create` method. If no border extension type is provided, a default extension of `BorderExtender.BORDER_COPY` will be used to perform the extension. See Section 3.7.3, "Rendering Hints."

Listing 8-4 shows a code sample for a `Rotate` operation for a rotation angle of 45 degrees. Since the rotation angle must be specified in radians, the example first converts 45 degrees to radians.

**Listing 8-4 Example Rotate Operation**

```
   // Create the rotation angle (45 degrees) and convert to
   // radians.
   int value = 45;
   float angle = (float)(value * (Math.PI/180.0F));

   // Create a ParameterBlock and specify the source and
   // parameters
   ParameterBlock pb = new ParameterBlock();
        pb.addSource(im);                    // The source image
        pb.add(0.0F);                        // The x origin
        pb.add(0.0F);                        // The y origin
        pb.add(angle);                       // The rotation angle
        pb.add(new InterpolationNearest()); // The interpolation

   // Create the rotate operation
   im = JAI.create("Rotate", pb, null);
```

### 8.3.4 Affine Transformation

An *affine transform* is a transformation of an image in which straight lines remain straight and parallel lines remain parallel, but the distance between lines and the angles between lines may change. Affine transformations include translation, scaling, and rotation.

Although there are separate JAI operations to handle translation, scaling, and rotation, the `Affine` operation can perform any of these transformations or any combination, such as scale and rotate.

The `Affine` operation performs (possibly filtered) affine mapping between a source and a destination image. For each pixel ($x$, $y$) of the destination, the source value at the fractional subpixel position ($x'$, $y'$) is constructed by means of an `Interpolation` object and written to the destination.

The `affine` operation takes one rendered or renderable source image and two parameters:

| **Parameter** | **Type** | **Description** |
|---|---|---|
| `transform` | `AffineTransform` | The affine transform matrix. |
| `interpolation` | `Interpolation` | The interpolation method for resampling. One of `INTERP_NEAREST`, `INTERP_BILINEAR`, `INTERP_BICUBIC`, or `INTERP_BICUBIC2`. |

The mapping between the destination pixel ($x$, $y$) and the source position ($x'$, $y'$) is given by:

$$x' = m00 * x + m01 * y + m02$$
$$y' = m10 * x + m11 * y + m12 \qquad (8.2)$$

where $m$ is a $3 \times 2$ transform matrix that inverts the matrix supplied as the `transform` argument.

The six elements of the transform matrix are `m00`, `m01`, `m02`, `m10`, `m11`, and `m12`. The constructor looks like this:

```
AffineTransform tr = new AffineTransform(m00, m10,
                                         m01, m11,
                                         m02, m12);
```

These six elements affect the transformation as follows:

| **Element** | **Description** |
|---|---|
| `m00` | The *x* coordinate scale element |
| `m10` | The *y* coordinate shear element |

| Element | Description |
|---|---|
| `m01` | The *x* coordinate shear element |
| `m11` | The *y* coordinate scale element |
| `m02` | The *x* coordinate translate element |
| `m12` | The *y* coordinate translate element |

The following matrix will translate an image 100 pixels to the right and 200 pixels down:

```
    AffineTransform tr = new AffineTransform(1.0,
                                             0.0,
                                             0.0,
                                             1.0,
                                             100.0,
                                             200.0);
```

The following matrix will zoom an image by a factor of 2 in both the *x* and *y* directions:

```
    AffineTransform tr = new AffineTransform(2.0,
                                             0.0,
                                             0.0,
                                             2.0,
                                             0.0,
                                             0.0);
```

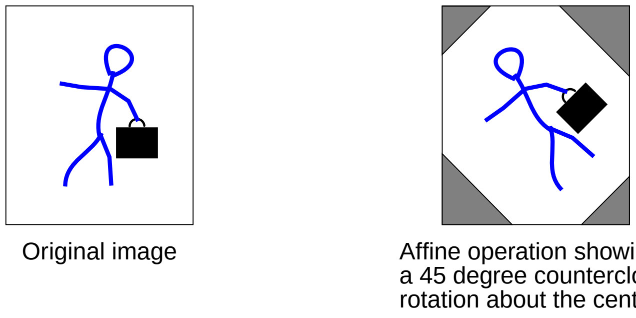

Original image
Affine operation showing a 45 degree counterclockwise rotation about the center

**Figure 8-7 Affine Operation**

When interpolations that require padding the source such as Bilinear or Bicubic interpolation are specified, the boundary of the source image needs to be extended such that it has the extra pixels needed to compute all the destination pixels. This extension is performed via the `BorderExtender` class. The type of border extension can be specified as a `RenderingHint` to the `JAI.create` method. If no border extension type is provided, a default extension of `BorderExtender.BORDER_COPY` will be used to perform the extension. See Section 3.7.3, "Rendering Hints."

Listing 8-5 shows a code sample for an `Affine` operation that performs a 45 degree counterclockwise rotation.

**Listing 8-5 Example Affine Transform Operation**

```
// Load the image.
String filename = "images/Trees.gif";
PlanarImage im = (PlanarImage)JAI.create("fileload",
                                         filename);

// Create the affine transform matrix.
AffineTransform tr = new AffineTransform(0.707107,
                                        -0.707106,
                                         0.707106,
                                         0.707106,
                                         0.0,
                                         0.0);

// Specify the type of interpolation.
Interpolation interp = new InterpolationNearest();

// Create the affine operation.
PlanarImage im2 = (PlanarImage)JAI.create("affine", im, tr,
                                          interp);
```

**API:** `java.awt.geom.AffineTransform`

- `static AffineTransform getTranslateInstance(double tx, double ty)`

  returns a transform representing a translation transformation.

| | | |
|---|---|---|
| *Parameters*: | `tx` | The distance by which coordinates are translated in the *x* axis direction. |
| | `ty` | The distance by which coordinates are translated in the *y* axis direction |

- `static AffineTransform getRotateInstance(double theta)`
  returns a transform representing a rotation transformation.

| | | |
|---|---|---|
| *Parameters*: | `theta` | The angle of rotation in radians. |

-
  ```
  static AffineTransform getRotateInstance(double theta,
      double x, double y)
  ```
  returns a transform that rotates coordinates around an anchor point.

| | | |
|---|---|---|
| *Parameters*: | `theta` | The angle of rotation in radians. |
| | `x` | The *x* coordinate of the anchor point of the rotation. |
| | `y` | The *y* coordinate of the anchor point of the rotation. |

- `static AffineTransform getScaleInstance(double sx, double sy)`
  returns a transform representing a scaling transformation.

| | | |
|---|---|---|
| *Parameters*: | `sx` | The factor by which coordinates are scaled along the *x* axis direction. |
| | `sy` | The factor by which coordinates are scaled along the *y* axis direction. |

- `static AffineTransform getShearInstance(double shx, double shy)`
  returns a transform representing a shearing transformation.

| | | |
|---|---|---|
| *Parameters*: | `shx` | The multiplier by which coordinates are shifted in the direction of the positive *x* axis as a factor of their *y* coordinate. |
| | `shy` | The multiplier by which coordinates are shifted in the direction of the positive *y* axis as a factor of their *x* coordinate. |

## 8.4 Perspective Transformation

Perspective distortions in images are sometimes introduced when the camera is at an angle to the subject. For an example, think of a camera in an aircraft above the earth. If the camera is aimed straight down, the resulting image will be a flat perspective image; that is, no distortion. All objects in the image appear in correct size relative to one another. However, if the camera is angled toward the earth horizon, perspective distortion is introduced. Objects closer to the camera appear larger than same-sized objects farther away from the camera. Perspective distortion has reduced the scale of the objects farthest away.

Perspective distortion can be corrected by applying a *perspective transform*. The perspective transform maps an arbitrary quadrilateral into another arbitrary quadrilateral, while preserving the straightness of lines. Unlike an affine transformation, the parallelism of lines in the source is not necessarily preserved in the output.

The perspective transform is represented by a 3 × 3 matrix that transforms homogenous source coordinates ($x$, $y$, 1) into destination coordinates ($x'$, $y'$, $w$). To convert back into non-homogenous coordinates, $x'$ and $y'$ are divided by $w$.

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m00x + m01y + m02 \\ m10x + m11y + m12 \\ m20x + m21y + m22 \end{bmatrix}$$

$$x' = \frac{m00x + m01y + m02}{m20x + m21y + m22}$$

$$y' = \frac{m10x + m11y + m12}{m20x + m21y + m22}$$

$$X = \frac{x'}{w}$$

$$Y = \frac{y'}{w}$$

The perspective transform is used with the perspective warp operation. See Section 8.7.7, “Perspective Warp.”

**API:** `javax.media.jai.PerspectiveTransform`

-
  ```
  PerspectiveTransform(float m00, float m01, float m02,
      float m10, float m11, float m12, float m20, float m21,
      float m22)
  ```

  constructs a new `PerspectiveTransform` from nine float values.

- `PerspectiveTransform(float[] flatmatrix)`

  constructs a new `PerspectiveTransform` from a one-dimensional array of nine float values, in row-major order.

- `PerspectiveTransform(float[][] matrix)`

  constructs a new `PerspectiveTransform` from a two-dimensional array of float values.

-
  ```
  PerspectiveTransform(double m00, double m01, double m02,
      double m10, double m11, double m12, double m20, double m21,
      double m22)
  ```

  constructs a new `PerspectiveTransform` from nine double values.

- `PerspectiveTransform(double[] flatmatrix)`

  constructs a new `PerspectiveTransform` from a one-dimensional array of nine double values, in row-major order.

- `PerspectiveTransform(double[][] matrix)`

  constructs a new `PerspectiveTransform` from a two-dimensional array of double values.

- `PerspectiveTransform(AffineTransform transform)`

  constructs a new `PerspectiveTransform` with the same effect as an existing `AffineTransform`.

### 8.4.1 Performing the Transform

The `PerspectiveTransform` class contains methods that perform the perspective transform on a specified point, an array of point objects, an array of floating point coordinates, or an array of double precision coordinates.

### 8.4.2 Mapping a Quadrilateral

The `PerspectiveTransform` class contains methods that may be used to create a perspective transform that can be used to map a unit square to or from an arbitrary quadrilateral and to map an arbitrary quadrilateral onto another arbitrary quadrilateral. The `getSquareToQuad` methods map the unit square onto an arbitrary quadrilateral:

$$(0, 0) \rightarrow (x0, y0)$$
$$(1, 0) \rightarrow (x1, y1)$$
$$(1, 1) \rightarrow (x2, y2)$$
$$(0, 1) \rightarrow (x3, y3)$$

The `getQuadToSquare` methods map an arbitrary quadrilateral onto the unit square:

$$(x0, y0) \rightarrow (0, 0)$$

$(x1, y1) \rightarrow (1, 0)$
$(x2, y2) \rightarrow (1, 1)$
$(x3, y3) \rightarrow (0, 1)$

The `getQuadToQuad` methods map an arbitrary quadrilateral onto another arbitrary quadrilateral:

$(x0, y0) \rightarrow (x0p, y0p)$
$(x1, y1) \rightarrow (x1p, y1p)$
$(x2, y2) \rightarrow (x2p, y2p)$
$(x3, y3) \rightarrow (x3p, y3p)$

**API:** `javax.media.jai.PerspectiveTransform`

- `static PerspectiveTransform getSquareToQuad(double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3)`

  creates a `PerspectiveTransform` that maps the unit square onto an arbitrary quadrilateral.

- `static PerspectiveTransform getSquareToQuad(float x0, float y0, float x1, float y1, float x2, float y2, float x3, float y3)`

  creates a `PerspectiveTransform` that maps the unit square onto an arbitrary quadrilateral.

- `static PerspectiveTransform getQuadToSquare(double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3)`

  creates a `PerspectiveTransform` that maps an arbitrary quadrilateral onto the unit square.

- `static PerspectiveTransform getQuadToSquare(float x0, float y0, float x1, float y1, float x2, float y2, float x3, float y3)`

  creates a `PerspectiveTransform` that maps an arbitrary quadrilateral onto the unit square.

- `static PerspectiveTransform getQuadToQuad(double x0, double y0, double x1, double y1, double x2, double y2, double x3, double y3, double x0p, double y0p, double x1p, double y1p, double x2p, double y2p, double x3p, double y3p)`

  creates a `PerspectiveTransform` that maps an arbitrary quadrilateral onto another arbitrary quadrilateral.

- `static PerspectiveTransform getQuadToQuad(float x0, float y0,`
  `float x1, float y1, float x2, float y2, float x3, float y3,`
  `float x0p, float y0p, float x1p, float y1p, float x2p,`
  `float y2p, float x3p, float y3p)`

  creates a `PerspectiveTransform` that maps an arbitrary quadrilateral onto another arbitrary quadrilateral.

### 8.4.3 Mapping Triangles

The `PerspectiveTransform` class contains methods that may be used to create a perspective transform that can be used to map one arbitrary triangle to another arbitrary triangle. This is done with one of the `getTriToTri` methods

**API:** `javax.media.jai.PerspectiveTransform`

- `static AffineTransform getTriToTri(double x0, double y0, double`
  `x1, double y1, double x2, double y2)`

  creates an `AffineTransform` that maps an arbitrary triangle onto another arbitrary triangle:

  $(x0, y0) \rightarrow (x0\text{p}, y0\text{p})$
  $(x1, y1) \rightarrow (x1\text{p}, y1\text{p})$
  $(x2, y2) \rightarrow (x2\text{p}, y2\text{p})$

- `static AffineTransform getTriToTri(float x0, float y0, float`
  `x1, float y1, float x2, float y2)`

  creates an `AffineTransform` that maps an arbitrary triangle onto another arbitrary triangle:

  $(x0, y0) \rightarrow (x0\text{p}, y0\text{p})$
  $(x1, y1) \rightarrow (x1\text{p}, y1\text{p})$
  $(x2, y2) \rightarrow (x2\text{p}, y2\text{p})$

### 8.4.4 Inverse Perspective Transform

The `PerspectiveTransform` class contains methods to perform an inverse perspective transform. One of the `inverseTransform` methods inverse transforms a specified Point2D to another Point2D. Another `inverseTransform` method inverse transforms an array of double-precision coordinates.

**API:** `javax.media.jai.PerspectiveTransform`

- `Point2D inverseTransform(Point2D ptSrc, Point2D ptDst)`

  inverse transforms the specified `ptSrc` and stores the result in `ptDst`. If `ptDst` is null, a new `Point2D` object will be allocated before storing. In either case, `ptDst` containing the transformed point is returned for convenience. Note that `ptSrc` and `ptDst` can the same. In this case, the input point will be overwritten with the transformed point.

| | | |
|---|---|---|
| *Parameters*: | `ptSrc` | The point to be inverse transformed. |
| | `ptDst` | The resulting transformed point. |

-
  ```
  inverseTransform(double[] srcPts, int srcOff, double[] dstPts,
      int dstOff, int numPts)
  ```

  inverse transforms an array of double precision coordinates by this transform.

| | | |
|---|---|---|
| *Parameters*: | `srcPts` | The array containing the source point coordinates. Each point is stored as a pair of *x*,*y* coordinates. |
| | `srcOff` | The offset to the first point to be transformed in the source array. |
| | `dstPts` | The array where the transformed point coordinates are returned. Each point is stored as a pair of *x*,*y* coordinates. |
| | `dstOff` | The offset to the location where the first transformed point is stored in the destination array. |
| | `numPts` | The number of point objects to be transformed. |

### 8.4.5 Creating the Adjoint of the Current Transform

The `PerspectiveTransform` class contains a method for creating a new PerspectiveTransform that is the adjoint of the current transform. The adjoint is defined as the matrix of cofactors, which in turn are the determinants of the submatrices defined by removing the row and column of each element from the original matrix in turn.

The adjoint is a scalar multiple of the inverse matrix. Because points to be transformed are converted into homogeneous coordinates, where scalar factors are irrelevant, the adjoint may be used in place of the true inverse. Since it is unnecessary to normalize the adjoint, it is both faster to compute and more numerically stable than the true inverse.

**API:** `javax.media.jai.PerspectiveTransform`

- `public PerspectiveTransform createAdjoint()`

  returns a new PerpectiveTransform that is the adjoint of the current transform.

## 8.5 Transposing

The `Transpose` operation is a combination of flipping and rotating. With a `Transpose` operation, you can (see Figure 8-8):

- Flip an image vertically across an imaginary horizontal axis that runs through the center of the image (`FLIP_VERTICAL`).
- Flip an image horizontally across an imaginary vertical axis that runs through the center of the image (`FLIP_HORIZONTAL`).
- Flip an image across its main diagonal axis, which runs from the upper left to the lower right corner (`FLIP_DIAGONAL`).
- Flip an image across its main anti-diagonal axis, which runs from the upper right to the lower left corner (`FLIP_ANTIDIAGONAL`).
- Rotate an image counterclockwise about its center by 90, 180, or 270 degrees (`ROTATE_90`, `ROTATE_180`, `ROTATE_270`).

The `transpose` operation takes one rendered or renderable source image and one parameter:

| Parameter | Type | Description |
|---|---|---|
| `type` | `Integer` | The type of flip operation to be performed. One of `FLIP_VERTICAL`, `FLIP_HORIZONTAL`, `FLIP_DIAGONAL`, `FLIP_ANTIDIAGONAL`, `ROTATE_90`, `ROTATE_180`, or `ROTATE_270` |

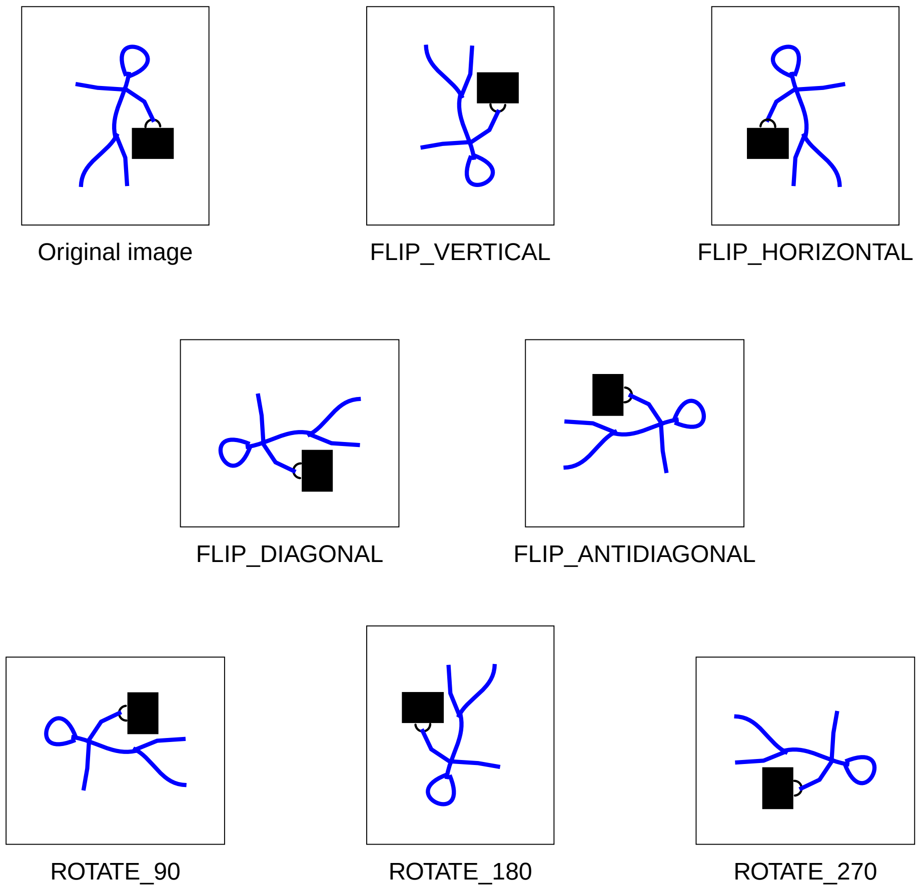


**Figure 8-8 Transpose Operations**

Listing 8-6 shows sample code for creating a `Transpose` operation. The example performs a horizontal flip on the source image and creates the destination image `im2`.

**Listing 8-6 Example Transpose Operation**

```
// Create a pattern image.
ParameterBlock pb = new ParameterBlock();
pb.add(image);
PlanarImage im0 = (PlanarImage)JAI.create("awtImage", pb);

// Transpose type : 0=FLIP_VERTICAL
//                : 1=FLIP_HORIZONTAL
//                : 2=FLIP_DIAGONAL
//                : 3=FLIP_ANTIDIAGONAL
//                : 4=ROTATE_90
//                : 5=ROTATE_180
//                : 6=ROTATE_270
int type = 1;

// Create the Transpose operation.
PlanarImage im2 = (PlanarImage)JAI.create("transpose", im0,
                                          type);
```

## 8.6 Shearing

Shearing can be visualized by thinking of an image superimposed onto a flexible rubber sheet. If you hold the sides of the sheet and move them up and down in opposite directions, the image will undergo a spatial stretching known as shearing. The `shear` operation shears an image either horizontally or vertically.

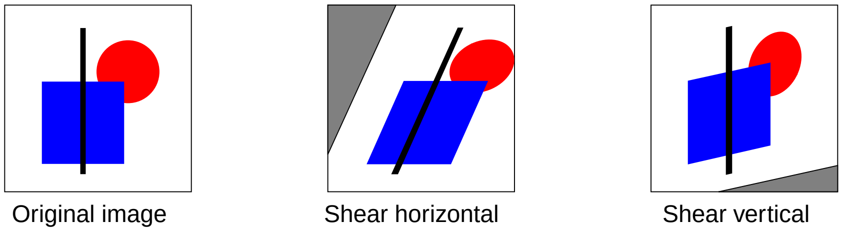


**Figure 8-9 Shearing Operations**

For each pixel (*x*, *y*) of the destination, the source value at the fractional subpixel position (*x*', *y*') is constructed by means of an `Interpolation` object and written to the destination (see “Interpolation” on page 249).

The shear operation takes one rendered source image and five parameters:

| Parameters | Type | Description |
|---|---|---|
| shear | Float | The shear value. |
| shearDir | Integer | The shear direction: SHEAR_HORIZONTAL or SHEAR_VERTICAL |
| xTrans | Float | The *x* translation. |
| yTrans | Float | The *y* translation. |
| interpolation | Interpolation | The interpolation method for resampling. One of INTERP_NEAREST, INTERP_BILINEAR, INTERP_BICUBIC, or INTERP_BICUBIC2. |

For a shearDir parameter of SHEAR_HORIZONTAL:

$$x' = x - \mathtt{xTrans} - y \cdot \mathtt{shear}$$

$$y' = y$$

For a shearDir parameter of SHEAR_VERTICAL:

$$x' = x$$

$$y' = y - \mathtt{yTrans} - x \cdot \mathtt{shear}$$

When interpolations that require padding the source such as Bilinear or Bicubic interpolation are specified, the boundary of the source image needs to be extended such that it has the extra pixels needed to compute all the destination pixels. This extension is performed via the BorderExtender class. The type of border extension can be specified as a RenderingHint to the JAI.create method. If no border extension type is provided, a default extension of BorderExtender.BORDER_COPY will be used to perform the extension. See Section 3.7.3, “Rendering Hints.”

Listing 8-7 shows a code sample for a Shear operation.

**Listing 8-7 Example Shear Operation**

```
// Load the image.
String filename = "images/Picketfence.gif";
PlanarImage im0 = (PlanarImage)JAI.create("fileload",
                                          filename);

imagePanel1 = new ScrollingImagePanel(im0, 512, 512);

// Specify the type of interpolation.
Interpolation interp = new InterpolationNearest();
```

**Listing 8-7 Example Shear Operation (Continued)**

```
    // Set the shear direction:
    //     0 = SHEAR_HORIZONTAL
    //     1 = SHEAR_VERTICAL
    int shear_dir = 1;

    // Set the shear value and the x and y translation values.
    float shear_amt = 0.7F;
    float x_trans = 50.0F;
    float y_trans = 100.0F;

    // Create the Shear operation.
    PlanarImage im2 = (PlanarImage)JAI.create("shear",
                                              im0,
                                              shear_amt,
                                              shear_dir,
                                              x_trans,
                                              y_trans,
                                              interp);

    // Display the image.
    imagePanel2 = new ScrollingImagePanel(im2, 512, 512);
    add(imagePanel2);
    pack();
    show();
```

## 8.7 Warping

The linear geometric transformations described in Section 8.3, “Geometric Transformation,” cannot introduce curvature in the mapping process. Image warping is a type of geometric transformation that introduces curvature into the mapping process. The introduction of curvature is important when an image has been distorted through lens aberrations and other non-linear processes.

Warping transformations, also known as *rubber sheet* transformations, can arbitrarily stretch the image about defined points. This type of operation provides a nonlinear transformation between source and destination coordinates.

JAI provides a transformation class, `Warp`, that is used for non-linear image coordinate transformation. As in the `Interpolation` class (see Section 8.2, “Interpolation”), pixel positions in the `Warp` class are represented using fixed-point coordinates, yielding subpixel accuracy but still allowing the use of integer arithmetic. The degree of precision is set by means of the `getSubSampleBitsH` (horizontal) and `getSubSampleBitsV` (vertical) parameters to the `warpRect` method.

The key method of this class is `warpRect`, which provides the locations of the pixels in source space that map to a given rectangular output region. The output region is specified using normal integer (full pixel) coordinates. The source positions returned by the method are specified in fixed-point, subpixel coordinates.

JAI supports seven warping functions:

- Polynomial warp – a polynomial-based description of an image warp (`WarpPolynomial`).
- General polynomial warp – a general polynomial-based description of an image warp (`WarpGeneralPolynomial`).
- Grid warp – a regular grid-based description of an image warp (`WarpGrid`).
- Quadratic warp – a quadratic-based description of an image warp (`WarpQuadratic`).
- Cubic warp – a cubic-based description of an image warp (`WarpCubic`).
- Perspective warp – a perspective (projective) warp (`WarpPerspective`).
- Affine warp – affine-based warp (`WarpAffine`).

**API:** `javax.media.jai.Warp`

-
  ```
  int[] warpRect(int x, int y, int width, int height,
      int subsampleBitsH, int subsampleBitsV, int[] destRect)
  ```
  computes the source subpixel positions for a given rectangular destination region. The destination region is specified using normal integral (full pixel) coordinates. The source positions returned by the method are specified in fixed point, subpixel coordinates using the current value of `getSubsampleBitsH()` and `getSubsampleBitsV()`.

| | | |
|---|---|---|
| *Parameters*: | `x` | The minimum *x* coordinate of the destination region. |
| | `y` | The minimum *y* coordinate of the destination region. |
| | `width` | The width of the destination region. |
| | `height` | The height of the destination region. |
| | `subsampleBitsH` | The number of fractional bits used to specify horizontal offsets in the `warpPositions` data. |

| | |
|---|---|
| `subsampleBitsV` | The number of fractional bits used to specify vertical offsets in the `warpPositions` data. |
| `destRect` | An int array containing at least 2*width*height elements, or null. If null, a new array will be constructed. |

As a convenience, an implementation is provided for this method that calls `warpSparseRect()`. Subclasses may wish to provide their own implementations for better performance.

-
  ```
  float[] warpRect(int x, int y, int width, int height,
      float[] destRect)
  ```

  computes the source subpixel positions for a given rectangular destination region. The destination region is specified using normal integral (full pixel) coordinates. The source positions returned by the method are specified in floating point.

  As a convenience, an implementation is provided for this method that calls `warpSparseRect()`. Subclasses may wish to provide their own implementations for better performance.

-
  ```
  int[] warpPoint(int x, int y, int subsampleBitsH,
      int subsampleBitsV, int[] destRect)
  ```

  computes the source subpixel position for a given destination pixel. The destination pixel is specified using normal integral (full pixel) coordinates. The source position returned by the method is specified in fixed point, subpixel coordinates using the `subsampleBitsH` and `subsampleBitsV` parameters.

  As a convenience, an implementation is provided for this method that calls `warpSparseRect()`. Subclasses may wish to provide their own implementations for better performance.

-
  ```
  float[] warpPoint(int x, int y, float[] destRect)
  ```

  computes the source subpixel position for a given destination pixel. The destination pixel is specified using normal integral (full pixel) coordinates. The source position returned by the method is specified in floating point.

  As a convenience, an implementation is provided for this method that calls `warpRect()`. Subclasses may wish to provide their own implementations for better performance.

-
  ```
  int[] warpSparseRect(int x, int y, int width, int height,
      int periodX, int periodY, int subsampleBitsH,
      int subsampleBitsV, int[] destRect)
  ```

  computes the source subpixel positions for a given rectangular destination region, subsampled with an integral period. The destination region is specified using normal integral (full pixel) coordinates. The source positions returned by the method are specified in fixed point, subpixel coordinates using the `subsampleBitsH` and `subsampleBitsV` parameters.

| *Parameters*: | | |
|---|---|---|
| | `x` | The minimum *X* coordinate of the destination region. |
| | `y` | The minimum *Y* coordinate of the destination region. |
| | `width` | The width of the destination region. |
| | `height` | The height of the destination region. |
| | `periodX` | The horizontal sampling period. |
| | `periodY` | The horizontal sampling period. |
| | `subsample-BitsH` | The number of fractional bits used to specify horizontal offsets in the `warpPositions` data. |
| | `subsample-BitsV` | The number of fractional bits used to specify vertical offsets in the `warpPositions` data. |
| | `destRect` | An int array containing at least $2\left(\frac{\text{width} + \text{periodX} - 1}{\text{periodX}} \times \frac{\text{height} + \text{periodY} - 1}{\text{periodY}}\right)$ elements, or null. If null, a new array will be constructed. |

  As a convenience, an implementation is provided for this method that calls `warpSparseRect()` with a float `destRect` parameter. Subclasses may wish to provide their own implementations for better performance.

-
  ```
  abstract float[] warpSparseRect(int x, int y, int width,
      int height, int periodX, int periodY, float[] destRect)
  ```

  computes the source subpixel positions for a given rectangular destination region, subsampled with an integral period. The destination region is specified using normal integral (full pixel) coordinates. The source positions returned by the method are specified in floating point.

This method is abstract in this class and must be provided in concrete subclasses.

- `Rectangle mapDestRect(Rectangle destRect)`

  computes a rectangle that is guaranteed to enclose the region of the source that is required in order to produce a given rectangular output region. The routine may return null if it is infeasible to compute such a bounding box.

  *Parameters*: `destRect` The `Rectangle` in destination coordinates.

  The default (superclass) implementation returns null.

### 8.7.1 Performing a Warp Operation

The `Warp` operation performs general warping on an image. The `warp` operation takes one rendered source image and two parameters:

| Parameters | Type | Description |
|---|---|---|
| `warp` | Warp | The warp object. One of `WarpAffine` `WarpGrid` `WarpPerspective` `WarpPolynomial` `WarpQuadratic` `WarpOpImage` |
| `interpolation` | `Interpolation` | The interpolation method for resampling. One of `INTERP_NEAREST`, `INTERP_BILINEAR`, `INTERP_BICUBIC`, or `INTERP_BICUBIC2`. |

To create a warp operation:

1. Create the warp object, which specifies the type of warp operation. The warp object will be one of the following:

| Object | Description |
|---|---|
| `WarpAffine` | An affine-based image warp. See Section 8.7.8, "Affine Warp." |
| `WarpCubic` | A cubic-based image warp. See Section 8.7.6, "Cubic Warp." |
| `WarpGeneralPolynomial` | A polynomial-based image warp for polynomials of a higher degree. See Section 8.7.3, "General Polynomial Warp." |
| `WarpGrid` | A grid-based image warp where the image may be warped in pieces. See Section 8.7.4, "Grid Warp." |
| `WarpPerspective` | A perspective or projective image warp. See Section 8.7.7, "Perspective Warp." |

| Object | Description |
|---|---|
| `WarpPolynomial` | A polynomial-based description of an image warp. See Section 8.7.2, “Polynomial Warp.” |
| `WarpQuadratic` | A quadratic-based description of an image warp. See Section 8.7.5, “Quadratic Warp.” |

2. Create the `ParameterBlock` object and add the source image and the necessary parameters to it. The `Warp` operation takes two parameters:

| Parameter | Description |
|---|---|
| `warp` | The `Warp` object. One of `WarpAffine`, `WarpCubic`, `WarpGeneralPolynomial`, `WarpGrid`, `WarpPerspective`, `WarpPolynomial`, or `WarpQuadratic`. |
| `interpolation` | The interpolation method for resampling. One of `INTERP_NEAREST`, `INTERP_BILINEAR`, `INTERP_BICUBIC`, or `INTERP_BICUBIC2`. |

   When interpolations that require padding the source such as Bilinear or Bicubic interpolation are specified, the boundary of the source image needs to be extended such that it has the extra pixels needed to compute all the destination pixels. This extension is performed via the `BorderExtender` class. The type of border extension can be specified as a `RenderingHint` to the `JAI.create` method. If no border extension type is provided, a default extension of `BorderExtender.BORDER_COPY` will be used to perform the extension. See Section 3.7.3, “Rendering Hints.”

3. Create the warp operation with the `JAI.create` method.

Listing 8-8 shows a sample code for a simple second-order warp operation.

**Listing 8-8 Example of a Second-order Warp**

```
// Create WarpPolynomial object for a polynomial warp
// operation.
WarpPolynomial warp;
     float[] coeffs = { 1.0F, 0.0F, 0.0F, 0.0F, 1.0F, 0.0F };

// Create the ParameterBlock and add the parameters to it.
ParameterBlock pb = new ParameterBlock();
     pb.addSource(srcImage);
     pb.add(warp);
     pb.add(new InterpolationNearest());

// Create the warp operation.
dstImage = JAI.create("warp", pb);
```

### 8.7.2 Polynomial Warp

The `WarpPolynomial` class provides a polynomial-based description of an image warp. The mapping is defined by two bivariate polynomial functions $X(x, y)$ and $Y(x, y)$ that define the source $x$ and $y$ positions that map to a given destination $(x, y)$ pixel coordinate.

The functions $X(x, y)$ and $Y(x, y)$ have the form:

$$\sum_{i=0}^{n} \sum_{j=0}^{i} a_{ij} \cdot x^{i-j} \cdot y^{j} \tag{8.3}$$

The `WarpPolynomial` constructor takes a `coeffs` parameter that must contain a number of coefficients of the form $(n + 1)(n + 2)$ for some $n$, where $n$ is the degree power of the polynomial. The coefficients appear in the following order:

$$1, x, y, x^2, xy, y^2, \ldots, x^n, x^{(n-1)y}, \ldots, xy^{(n-1)}, y^n$$

with the coefficients of the polynomial defining the source $x$ coordinates appearing before those defining the $y$ coordinates.

The source $(x, y)$ coordinate is pre-scaled by the factors `preScaleX` and `preScaleY` prior to the evaluation of the polynomial. The result of the polynomial evaluations are scaled by `postScaleX` and `postScaleY` to produce the destination pixel coordinates. This process allows for better precision of the results.

The number of points needed to control the alignment of the image relates directly to the order of warp. Three control points constitute a first-order warp. Six points constitute a second-order warp. The number of points required for each degree of warp are as follows:

| Degree of Warp | Number of Points |
|---|---|
| 1 | 3 |
| 2 | 6 |
| 3 | 10 |
| 4 | 15 |
| 5 | 21 |
| 6 | 28 |
| 7 | 36 |

**API:** `javax.media.jai.WarpPolynomial`

- `WarpPolynomial(float[] coeffs)`

  constructs a `WarpPolynomial` with pre- and post-scale factors of 1.

  | | | |
  |---|---|---|
  | *Parameters*: | `coeffs` | The destination to source transform coefficients. |

-
  ```
  WarpPolynomial(float[] coeffs, float preScaleX, float
      preScaleY, float postScaleX, float postScaleY)
  ```

  constructs a `WarpPolynomial` with a given transform mapping destination pixels into source space. Note that this is the inverse of the customary specification of the mapping of an image.

  | | | |
  |---|---|---|
  | *Parameters*: | `coeffs` | The destination-to-source transform coefficients. |
  | | `preScaleX` | The scale factor to apply to source *x* positions. |
  | | `preScaleY` | The scale factor to apply to source *y* positions. |
  | | `postScaleX` | The scale factor to apply to destination *x* positions. |
  | | `postScaleY` | The scale factor to apply to destination *y* positions. |

- `float[] getCoeffs()`

  returns the raw coefficients array.

- `int getDegree()`

  returns the degree of the warp polynomials.

-
  ```
  static WarpPolynomial createWarp(float[] sourceCoords,
      int sourceOffset, float[] destCoords, int destOffset,
      int numCoords, float preScaleX, float preScaleY,
      float postScaleX, float postScaleY, int degree)
  ```

  returns an instance of `WarpPolynomial` or its subclasses that approximately maps the given scaled destination image coordinates into the given scaled source image coordinates.

| | | |
|---|---|---|
| *Parameters*: | `sourceCoords` | An array of floats containing the source coordinates with *x* and *y* alternating. |
| | `sourceOffset` | The initial entry of `sourceCoords` to be used. |
| | `destCoords` | An array of floats containing the destination coordinates with *x* and *y* alternating. |
| | `destOffset` | The initial entry of `destCoords` to be used. |
| | `numCoords` | The number of coordinates from `sourceCoords` and `destCoords` to be used. |
| | `preScaleX` | The scale factor to apply to source *x* positions. |
| | `preScaleY` | The scale factor to apply to source *y* positions. |
| | `postScaleX` | The scale factor to apply to destination *x* positions. |
| | `postScaleY` | The scale factor to apply to destination *y* positions. |
| | `degree` | The desired degree of the warp polynomials. |

## 8.7.3 General Polynomial Warp

The `WarpGeneralPolynomial` class provides a concrete implementation of `WarpPolynomial` for polynomials of a higher degree.

The mapping is defined by two bivariate polynomial functions $X(x, y)$ and $Y(x, y)$ that define the source $X$ and $Y$ positions that map to a given destination $(x, y)$ pixel coordinate.

The functions $X(x, y)$ and $Y(x, y)$ have the form:

$$\sum_{i=0}^{n} \sum_{j=0}^{i} a_{ij} \cdot x^{i-j} \cdot y^{j} \tag{8.4}$$

The `xCoeffs` and `yCoeffs` parameters must contain the same number of coefficients of the form $(n + 1)(n + 2)/2$ for some $n$, where $n$ is the non-negative degree power of the polynomial. The coefficients, in order, are associated with the terms:

$1, x, y, x^2, x*y, y^2, ..., x^n, x^{(n-1)}*y, ..., x*y^{(n-1)}, y^n$

and coefficients of value 0 can not be omitted.

The destination pixel coordinates (the arguments to the X() and Y() functions) are given in normal integral pixel coordinates, while the output of the functions is given in fixed-point, subpixel coordinates with a number of fractional bits specified by the `subsampleBitsH` and `subsampleBitsV` parameters.

**API:** `javax.media.jai.WarpGeneralPolynomial`

- `WarpGeneralPolynomial(float[] xCoeffs, float[] yCoeffs)`
  constructs a `WarpGeneralPolynomial` with pre- and post-scale factors of 1.

| | | |
|---|---|---|
| *Parameters*: | `xCoeffs` | The destination to source transform coefficients for the *x* coordinate. |
| | `yCoeffs` | The destination to source transform coefficients for the *y* coordinate. |

-
  ```
  WarpGeneralPolynomial(float[] xCoeffs, float[] yCoeffs,
      float preScaleX,  float preScaleY, float postScaleX,
      float postScaleY)
  ```

  constructs a `WarpGeneralPolynomial` with a given transform mapping destination pixels into source space. Note that this is the inverse of the customary specification of the mapping of an image.

| | | |
|---|---|---|
| *Parameters*: | `xCoeffs` | The destination to source transform coefficients for the *x* coordinate. |
| | `yCoeffs` | The destination to source transform coefficients for the *y* coordinate. |
| | `preScaleX` | The scale factor to apply to source *x* positions. |
| | `preScaleY` | The scale factor to apply to source *y* positions. |
| | `postScaleX` | The scale factor to apply to destination *x* positions. |
| | `postScaleY` | The scale factor to apply to destination *y* positions. |

-
  ```
  float[] warpSparseRect(int x, int y, int width, int height,
      int periodX, int periodY, float[] destRect)
  ```

  computes the source subpixel positions for a given rectangular destination region, subsampled with an integral period.

| | | |
|---|---|---|
| *Parameters*: | `x` | The minimum *X* coordinate of the destination region. |
| | `y` | The minimum *Y* coordinate of the destination region. |
| | `width` | The width of the destination region. |
| | `height` | The height of the destination region. |
| | `periodX` | The horizontal sampling period. |
| | `periodY` | The horizontal sampling period. |
| | `destRect` | An int array containing at least $2\left(\frac{\text{width} + \text{periodX} - 1}{\text{periodX}} \times \frac{\text{height} + \text{periodY} - 1}{\text{periodY}}\right)$ elements, or null. If null, a new array will be constructed. |

### 8.7.4 Grid Warp

If polynomial warping is impractical, the image may be warped in pieces using grid warping, also known as *control grid interpolation*. In the most common implementation of grid warping, specified input control points form a grid of contiguous, horizontally-oriented rectangles in the output image. The mapping from destination pixels to source positions is described by bilinear interpolation between a rectilinear grid of points with known mappings.

Given a destination pixel coordinate ($x$, $y$) that lies within a cell having corners at ($x0$, $y0$), ($x1$, $y0$), ($x0$, $y1$), and ($x1$, $y1$), with source coordinates defined at each respective corner equal to ($sx0$, $sy0$), ($sx1$, $sy1$), ($sx2$, $sy2$), and ($sx3$, $sy3$), the source position ($sx$, $sy$) that maps onto ($x$, $y$) is given by the following equations:

$$\begin{aligned} \mathtt{xfrac} &= \frac{x - x0}{x1 - x0} \\ \mathtt{yfrac} &= \frac{y - y0}{y1 - y0} \end{aligned} \qquad (8.5)$$

$$\begin{aligned} s &= sx0 + (sx1 - sx0) \cdot \mathtt{xfrac} \\ t &= sy0 + (sy1 - sy0) \cdot \mathtt{xfrac} \end{aligned} \qquad (8.6)$$

$$\begin{aligned} u &= sx2 + (sx3 - sx2) \cdot \mathtt{xfrac} \\ v &= sy2 + (sy3 - sy2) \cdot \mathtt{xfrac} \end{aligned} \qquad (8.7)$$

$$\begin{aligned} sx &= s + (u - s) \cdot \mathtt{yfrac} \\ sy &= t + (v - t) \cdot \mathtt{yfrac} \end{aligned} \qquad (8.8)$$

The source $x$ and $y$ values are interpolated horizontally along the top and bottom edges of the grid cell, and the results are interpolated vertically, as shown in Figure 8-10.

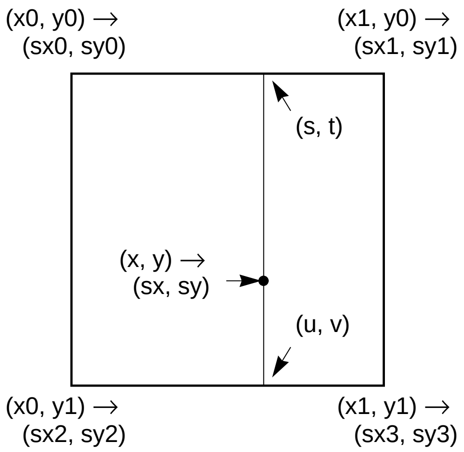


**Figure 8-10 Warp Grid**

The grid is defined by a set of equal-sized cells starting at (`xStart`, `yStart`). The width of each cell is defined by the `xStep` parameter and the height is defined by the `yStep` parameter. There are `xNumCells` cells horizontally and `yNumCells` cells vertically.

The degree of warping within each cell is defined by the values in the `warpPositions` parameter. This parameter must contain the following values:

`warpPositions` = 2(`xnumCells` + 1)(`yNumCells` + 1)

These values alternately contain the source *x* and *y* coordinates that map to the upper-left corner of each cell in the destination image. The cells are enumerated in row-major order, that is, all the grid points along a row are enumerated first, then the gird points for the next row are enumerated, and so on.

For example, if `xNumCells` is 2 and `yNumCells` is 1, the order of the data in the table would be as follows:

```
x00, y00, x10, y10, x20, y20, x01, y01, x11, y11, x21, y21
```

for a total of 2(2 + 1)(1 + 1) = 12 elements.

**API:** `javax.media.jai.WarpGrid`

-
  ```
  WarpGrid(int xStart, int xStep, int xNumCells, int yStart,
      int yStep, int yNumCells, float[] warpPositions)
  ```

  constructs a `WarpGrid` with a given grid-based transform mapping destination pixels into source space. Note that this is the inverse of the customary specification of the mapping of an image.

| | | |
|---|---|---|
| *Parameters*: | `xStart` | The minimum *x* coordinate of the grid. |
| | `xStep` | The horizontal spacing between grid cells. |
| | `xNumCells` | The number of grid cell columns. |
| | `yStart` | The minimum *y* coordinate of the grid. |
| | `yStep` | The vertical spacing between grid cells. |
| | `yNumCells` | The number of grid cell rows. |
| | `warp-Positions` | A float array of length $2(\text{xNumCells} + 1)(\text{yNumCells} + 1)$ containing the warp positions at the grid points in row-major order. |

-
  ```
  WarpGrid(Warp master, int xStart, int xStep, int xNumCells,
      int yStart, int yStep, int yNumCells)
  ```

  constructs a `WarpGrid` object by sampling the displacements given by another `Warp` object of any kind.

| | | |
|---|---|---|
| *Parameters*: | `master` | The `Warp` object used to initialized the grid displacements. |
| | `xStart` | The minimum *x* coordinate of the grid. |
| | `xStep` | The horizontal spacing between grid cells. |
| | `xNumCells` | The number of grid cell columns. |
| | `yStart` | The minimum *y* coordinate of the grid. |
| | `yStep` | The vertical spacing between grid cells. |
| | `yNumCells` | The number of grid cell rows. |

-
  ```
  float[] warpSparseRect(int x, int y, int width, int height,
      int periodX, int periodY, float[] destRect)
  ```

  computes the source subpixel positions for a given rectangular destination region. The destination region is specified using normal integer (full pixel) coordinates. The source positions returned by the method are specified in fixed point, subpixel coordinates using the of `subsampleBitsH` and `subsampleBitsV` parameters.

| *Parameters*: | | |
|---|---|---|
| | `x` | The minimum *x* coordinate of the destination region. |
| | `y` | The minimum *y* coordinate of the destination region. |
| | `width` | The width of the destination region. |
| | `height` | The height of the destination region. |
| | `periodX` | The horizontal sampling period. |
| | `periodY` | The vertical sampling period. |
| | `destRect` | An int array containing at least $2\left(\frac{\text{width} + \text{periodX} - 1}{\text{periodX}} \times \frac{\text{height} + \text{periodY} - 1}{\text{periodY}}\right)$ elements, or null. If null, a new array will be constructed. |

## 8.7.5 Quadratic Warp

The `WarpQuadratic` class provides a quadratic-based description of an image warp. The source position (*x*', *y*') of a point (*x*, *y*) is given by the following quadratic bivariate polynomial:

$$\begin{aligned} x' &= p(x, y) = c_1 + c_2x + c_3y + c_4x^2 + c_5xy + c_6y^2 \\ y' &= q(x, y) = c_7 + c_8x + c_9y + c_{10}x^2 + c_{11}xy + c_{12}y^2 \end{aligned} \tag{8.9}$$

**API:** `javax.media.jai.WarpQuadratic`

-
  ```
  WarpQuadratic(float[] xCoeffs, float[] yCoeffs,
      float preScaleX, float preScaleY, float postScaleX,
      float postScaleY)
  ```

  constructs a WarpQuadratic with a given transform mapping destination pixels into source space. Note that this is the inverse of the customary specification of the mapping of an image. The coeffs arrays must each contain six floats corresponding to the coefficients c1, c2, etc. as shown in the class comment..

| | | |
|---|---|---|
| *Parameters*: | `xCoeffs` | The six destination-to-source transform coefficients for the *x* coordinate. |
| | `yCoeffs` | The six destination-to-source transform coefficients for the *y* coordinate. |
| | `preScaleX` | The scale factor to apply to source *x* positions. |
| | `preScaleY` | The scale factor to apply to source *y* positions. |
| | `postScaleX` | The scale factor to apply to destination *x* positions. |
| | `postScaleY` | The scale factor to apply to destination *y* positions. |

- `WarpQuadratic(float[] xCoeffs, float[] yCoeffs)`

  constructs a WarpQuadratic with pre- and post-scale factors of 1.

-
  ```
  float[] warpSparseRect(int x, int y, int width, int height,
      int periodX, int periodY, float[] destRect)
  ```

  computes the source subpixel positions for a given rectangular destination region, subsampled with an integral period. The destination region is specified using normal integral (full pixel) coordinates. The source positions returned by the method are specified in floating point.

| | | |
|---|---|---|
| *Parameters*: | `x` | The minimum *x* coordinate of the destination region. |
| | `y` | The minimum *y* coordinate of the destination region. |
| | `width` | The width of the destination region. |
| | `height` | The height of the destination region. |
| | `periodX` | The horizontal sampling period. |

| | |
|---|---|
| `periodY` | The vertical sampling period. |
| `destRect` | A float array containing at least $2\left(\frac{\text{width} + \text{periodX} - 1}{\text{periodX}} \times \frac{\text{height} + \text{periodY} - 1}{\text{periodY}}\right)$ elements, or null. If null, a new array will be constructed. |

### 8.7.6 Cubic Warp

The `WarpCubic` class performs a cubic-based image warp. The source position ($x'$, $y'$) of a point ($x$, $y$) is given by the following cubic polynomial:

$$x' = p(x, y) = c_1 + c_2 x + c_3 y + c_4 x^2 + c_5 xy + c_6 y^2 + c_7 x^3 + c_8 x^2 y + c_9 xy^2 + c_{10} y^3 \tag{8.10}$$

$$y' = q(x, y) = c_{11} + c_{12} x + c_{13} y + c_{14} x^2 + c_{15} xy + c_{16} y^2 + c_{17} x^3 + c_{18} x^2 y + c_{19} xy^2 + c_{20} y^3 \tag{8.11}$$

**API:** `javax.media.jai.WarpCubic`

-
  ```
  WarpCubic(float[] xCoeffs, float[] yCoeffs, float preScaleX,
      float preScaleY, float postScaleX, float postScaleY)
  ```
  constructs a `WarpCubic` with a given transform mapping destination pixels into source space. Note that this is the inverse of the customary specification of the mapping of an image. The `coeffs` array must contain 12 floats corresponding to the coefficients a, b, etc. as shown in the class comment.

| | | |
|---|---|---|
| *Parameters*: | `xCoeffs` | The ten destination to source transform coefficients for the *x* coordinate. |
| | `yCoeffs` | The ten destination to source transform coefficients for the *y* coordinate. |
| | `preScaleX` | The scale factor to apply to source *x* positions. |
| | `preScaleY` | The scale factor to apply to source *y* positions. |

| | |
|---|---|
| `postScaleX` | The scale factor to apply to destination *x* positions. |
| `postScaleY` | The scale factor to apply to destination *y* positions. |

- `WarpCubic(float[] xCoeffs, float[] yCoeffs)`

  constructs a `WarpCubic` with pre- and post-scale factors of 1.

-
  ```
  float[] warpSparseRect(int x, int y, int width, int height,
      int periodX, int periodY, float[] destRect)
  ```

  computes the source subpixel positions for a given rectangular destination region, subsampled with an integral period.

| *Parameters*: | | |
|---|---|---|
| | `x` | The minimum *x* coordinate of the destination region. |
| | `y` | The minimum *y* coordinate of the destination region. |
| | `width` | The width of the destination region. |
| | `height` | The height of the destination region. |
| | `periodX` | The horizontal sampling period. |
| | `periodY` | The vertical sampling period. |
| | `destRect` | A float array containing at least $2\left(\frac{\text{width + periodX} - 1}{\text{periodX}} \times \frac{\text{height + periodY} - 1}{\text{periodY}}\right)$ elements, or null. If null, a new array will be constructed. |

### 8.7.7 Perspective Warp

Perspective distortions in images caused by camera-to-target viewing angle can be restored through perspective warping. Perspective distortion appears as the reduction in scale of an object that recedes from the foreground into the background of the image.

The `WarpPerspective` class provides a perspective (projective) warp. The transform is specified as a mapping from destination space to source space. In other words, it is the inverse of the normal specification of a perspective image transformation. See Section 8.4, “Perspective Transformation,” for a description of the `PerspectiveTransform` class.

**API:** `javax.media.jai.WarpPerspective`

- `WarpPerspective(PerspectiveTransform transform)`

  constructs a `WarpPerspective` with a given transform mapping destination pixels into source space. Note that this is the inverse of the customary specification of perspective mapping of an image.

| | | |
|---|---|---|
| *Parameters*: | `transform` | The destination-to-source transform. |

- `PerspectiveTransform getTransform()`

  returns a clone of the `PerspectiveTransform` associated with this `WarpPerspective` object.

-
  ```
  int[] warpSparseRect(int x, int y, int width, int height,
      int periodX, int periodY, float[] destRect)
  ```

  computes the source subpixel positions for a given rectangular destination regions subsampled with an integral period. The destination region is specified using normal integral (full pixel) coordinates. The source positions returned by the method are specified in floating-point.

| | | |
|---|---|---|
| *Parameters*: | `x` | The minimum *x* coordinate of the destination region. |
| | `y` | The minimum *y* coordinate of the destination region. |
| | `width` | The width of the destination region. |
| | `height` | The height of the destination region. |
| | `periodX` | The horizontal sampling period. |
| | `periodY` | The vertical sampling period. |
| | `destRect` | A float array containing at least $2\left(\frac{\text{width} + \text{periodX} - 1}{\text{periodX}} \times \frac{\text{height} + \text{periodY} - 1}{\text{periodY}}\right)$ elements, or null. If null, a new array will be constructed. |

## 8.7.8 Affine Warp

The `WarpAffine` class provides an affine-based warp. The transform is specified as a mapping from destination space to source space. In other words, it is the inverse of the normal specification of an affine image transformation.

The source position (*x*', *y*') of a point (*x*, *y*) is given by the quadratic bivariate polynomial:

$$\begin{aligned} x' &= p(x, y) = c_1 + c_2 x = c_3 y \\ y' &= q(x, y) = c_4 + c_5 x + c_6 y \end{aligned} \tag{8.12}$$

Listing 8-9 shows a code sample for an affine-based warp operation.

**Listing 8-9 Example Affine Warp**

```
// Create the transform parameter (WarpAffine).
double m00 = 0.8;
double m10 = 0.3;
double m01 = -0.7;
double m11 = 1.4;
double m02 = 230.3;
double m12 = -115.7;
AffineTransform transform = new AffineTransform(m00, m10,
                                                m01, m11,
                                                m02, m12);
Warp warp = new WarpAffine(transform);

// Create the interpolation parameter.
Interpolation interp = new InterpolationNearest(8);

// Create the ParameterBlock.
ParameterBlock pb = new ParameterBlock();
pb.addSource(src);
pb.add(warp);
pb.add(interp);

// Create the warp operation.
return (RenderedImage)JAI.create("warp", pb);
```

**API:** `javax.media.jai.WarpAffine`

-
  ```
  public WarpAffine(float[] xCoeffs, float[] yCoeffs,
      float preScaleX, float preScaleY, float postScaleX,
      float postScaleY)
  ```

  constructs a `WarpAffine` with a given transform mapping destination pixels into source space. The transform is given by:

  ```
  x’ = xCoeffs[0] + xCoeffs[1]*x + xCoeffs[2]*y;
  y’ = yCoeffs[0] + yCoeffs[1]*x + yCoeffs[2]*y;
  ```

where x' and y' are the source image coordinates and x and y are the destination image coordinates.

| | | |
|---|---|---|
| *Parameters*: | `xCoeffs` | The three destination-to-source transform coefficients for the *x* coordinate. |
| | `yCoeffs` | The three destination-to-source transform coefficients for the *y* coordinate. |
| | `preScaleX` | The scale factor to apply to source *x* positions. |
| | `preScaleY` | The scale factor to apply to source *y* positions. |
| | `postScaleX` | The scale factor to apply to destination *x* positions. |
| | `postScaleY` | The scale factor to apply to destination *y* positions. |

- `WarpAffine(float[] xCoeffs, float[] yCoeffs)`

  constructs a `WarpAffine` with pre- and post-scale factors of 1.

-
  ```
  public WarpAffine(AffineTransform transform, float preScaleX,
      float preScaleY, float postScaleX, float postScaleY)
  ```

  constructs a `WarpAffine` with a given transform mapping destination pixels into source space.

| | | |
|---|---|---|
| *Parameters*: | `transform` | The destination-to-source transform. |
| | `preScaleX` | The scale factor to apply to source *x* positions. |
| | `preScaleY` | The scale factor to apply to source *y* positions. |
| | `postScaleX` | The scale factor to apply to destination *x* positions. |
| | `postScaleY` | The scale factor to apply to destination *y* positions. |

- `WarpAffine(AffineTransform transform)`

  constructs a `WarpAffine` with pre- and post-scale factors of 1.

| | | |
|---|---|---|
| *Parameters*: | `transform` | An `AffineTransform`. |

- `AffineTransform getTransform()`

  returns a clone of the `AffineTransform` associated with this `WarpAffine` object.

-
  ```
  float[] warpSparseRect(int x, int y, int width, int height,
      int periodX, int periodY, float[] destRect)
  ```

  computes the source subpixel positions for a given rectangular destination region, subsampled with an integral period. The destination region is specified using normal integral (full pixel) coordinates. The source positions returned by the method are specified in floating point.

| | | |
|---|---|---|
| *Parameters*: | `x` | The minimum *x* coordinate of the destination region. |
| | `y` | The minimum *y* coordinate of the destination region. |
| | `width` | The width of the destination region. |
| | `height` | The height of the destination region. |
| | `periodX` | The horizontal sampling period. |
| | `periodY` | The vertical sampling period. |
| | `destRect` | A float array containing at least $2\left(\frac{\text{width} + \text{periodX} - 1}{\text{periodX}} \times \frac{\text{height} + \text{periodY} - 1}{\text{periodY}}\right)$ elements, or null. If null, a new array will be constructed. |

- `Rectangle mapDestRect(Rectangle destRect)`

  computes a `Rectangle` that is guaranteed to enclose the region of the source that is required in order to produce a given rectangular output region.

| | | |
|---|---|---|
| *Parameter*: | `destRect` | The `Rectangle` in destination coordinates. |

CHAPTER 9

# Image Analysis

THIS chapter describes the JAI API image analysis operators.

## 9.1 Introduction

The JAI API image analysis operators are used to directly or indirectly extract information from an image. The JAI API supports the following image analysis functions:

- Finding the mean value of an image region
- Finding the minimum and maximum values in an image (extrema)
- Producing a histogram of an image
- Detecting edges in an image
- Performing statistical operations

## 9.2 Finding the Mean Value of an Image Region

The `Mean` operation scans a specified region of an image and computes the image-wise mean pixel value for each band within the region. The region of interest does not have to be a rectangle. If no region is specified (null), the entire image is scanned to generate the histogram. The image data pass through the operation unchanged.

The mean operation takes one rendered source image and three parameters:

| Parameter | Type | Description |
|---|---|---|
| roi | ROI | The region of the image to scan. A null value means the whole image. |
| xPeriod | Integer | The horizontal sampling rate. May not be less than 1. |
| yPeriod | Integer | The vertical sampling rate. May not be less than 1. |

The region of interest (ROI) does not have to be a rectangle. It may be null, in which case the entire image is scanned to find the image-wise mean pixel value for each band.

The set of pixels scanned may be reduced by specifying the xPeriod and yPeriod parameters, which define the sampling rate along each axis. These variables may not be less than 1. However, they may be null, in which case the sampling rate is set to 1; that is, every pixel in the ROI is processed.

The image-wise mean pixel value for each band may be retrieved by calling the getProperty method with "mean" as the property name. The return value has type java.lang.Number[#bands].

Listing 9-1 shows a partial code sample of finding the image-wise mean pixel value of an image in the rendered mode.

**Listing 9-1 Finding the Mean Value of an Image Region**

```
     // Set up the parameter block for the source image and
     // the three parameters.
     ParameterBlock pb = new ParameterBlock();
     pb.addSource(im);   // The source image
     pb.add(null);       // null ROI means whole image
     pb.add(1);          // check every pixel horizontally
     pb.add(1);          // check every pixel vertically

     // Perform the mean operation on the source image.
     RenderedImage meanImage = JAI.create("mean", pb, null);

     // Retrieve and report the mean pixel value.
     double[] mean = (double[])meanImage.getProperty("mean");
     System.out.println("Band 0 mean = " + mean[0]);
```

## 9.3 Finding the Extrema of an Image

The Extrema operation scans a specific region of a rendered image and finds the image-wise minimum and maximum pixel values for each band within that region of the image. The image pixel data values pass through the operation unchanged. The `extrema` operation can be used to obtain information to compute the scale and offset factors for the amplitude rescaling operation (see Section 7.4, “Amplitude Rescaling”).

The region-wise maximum and minimum pixel values may be obtained as properties. Calling the `getProperty` method on this operation with “`extrema`” as the property name retrieves both the region-wise maximum and minimum pixel values. Calling it with “`maximum`” as the property name retrieves the region-wise maximum pixel value, and with “`minimum`” as the property name retrieves the region-wise minimum pixel value.

The return value for `extrema` has type `double[2][#bands]`, and those for `maximum` and `minimum` have type `double[#bands]`.

The region of interest (ROI) does not have to be a rectangle. It may be `null`, in which case the entire image is scanned to find the image-wise maximum and minimum pixel values for each band.

The `extrema` operation takes one rendered source image and three parameters:

| Parameter | Type | Description |
|---|---|---|
| `roi` | `ROI` | The region of the image to scan. |
| `xPeriod` | `Integer` | The horizontal sampling rate (may not be less than 1). |
| `yPeriod` | `Integer` | The vertical sampling rate (may not be less than 1). |

The set of pixels scanned may be further reduced by specifying the `xPeriod` and `yPeriod` parameters that represent the sampling rate along each axis. These variables may not be less than 1. However, they may be `null`, in which case the sampling rate is set to 1; that is, every pixel in the ROI is processed.

Listing 9-2 shows a partial code sample of using the `extrema` operation to obtain both the image-wise maximum and minimum pixel values of the source image.

**Listing 9-2 Finding the Extrema of an Image**

```
   // Set up the parameter block for the source image and
   // the constants
   ParameterBlock pb = new ParameterBlock();
   pb.addSource(im);   // The source image
   pb.add(roi);        // The region of the image to scan
   pb.add(50);         // The horizontal sampling rate
   pb.add(50);         // The vertical sampling rate
```

**Listing 9-2 Finding the Extrema of an Image (Continued)**

```
   // Perform the extrema operation on the source image
   RenderedOp op = JAI.create("extrema", pb);

   // Retrieve both the maximum and minimum pixel value
   double[][] extrema = (double[][]) op.getProperty("extrema");
```

## 9.4 Histogram Generation

An image histogram is an analytic tool used to measure the amplitude distribution of pixels within an image. For example, a histogram can be used to provide a count of the number of pixels at amplitude 0, the number at amplitude 1, and so on. By analyzing the distribution of pixel amplitudes, you can gain some information about the visual appearance of an image. A high-contrast image contains a wide distribution of pixel counts covering the entire amplitude range. A low contrast image has most of the pixel amplitudes congregated in a relatively narrow range.

Usually, the wider histogram represents a more visually-appealing image.

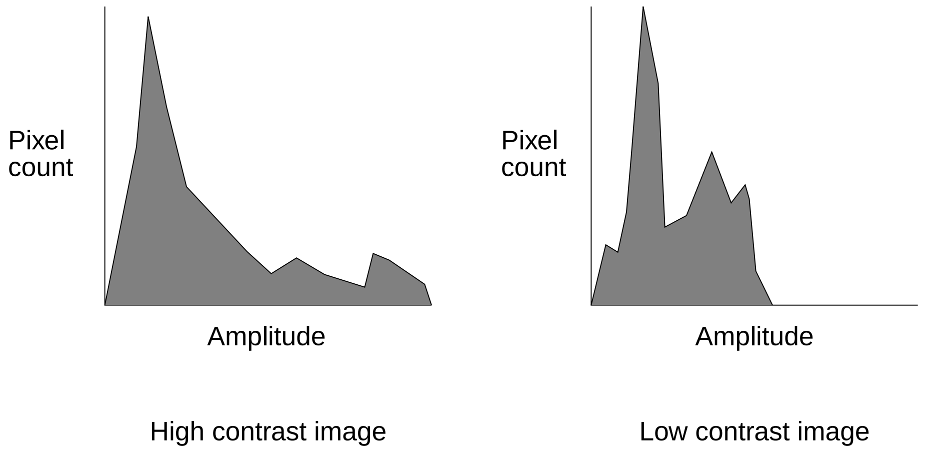


**Figure 9-1 Example Histograms**

The primary tasks needed to perform a histogram operation are as follows:

1. Create a `Histogram` object, which specifies the type of histogram to be generated.

2. Create a `Histogram` operation with the required parameters or create a `ParameterBlock` with the parameters and pass it to the `Histogram` operation.
3. Read the histogram data stored in the object. The data consists of:
   - Number of bands in the histogram
   - Number of bins for each band of the image
   - Lowest value checked for each band
   - Highest value checked for each band

## 9.4.1 Specifying the Histogram

The `Histogram` object accumulates the histogram information. A histogram counts the number of image samples whose values lie within a given range of values, or "bins." The source image may be of any data type.

The `Histogram` contains a set of bins for each band of the image. These bins hold the information about gray or color levels. For example, to take the histogram of an eight-bit grayscale image, the `Histogram` might contain 256 bins. When reading the `Histogram`, bin 0 will contain the number of 0's in the image, bin 1 will contain the number of 1's, and so on.

The `Histogram` need not contain a bin for every possible value in the image. It is possible to specify the lowest and highest values that will result in a bin count being incremented. It is also possible to specify fewer bins than the number of levels being checked. In this case, each bin will hold the count for a range of values. For example, for a `Histogram` with only four bins used with an 8-bit grayscale image, the number of occurrences of values 0 through 63 will be stored in bin 0, occurrences of values 64 through 127 will be stored in bin 1, and so on.

The `Histogram` object takes three parameters:

| **Parameter** | **Description** |
|---|---|
| `numBins` | An array of `ints`, each element of which specifies the number of bins to be used for one band of the image. The number of elements in the array must match the number of bands in the image. |
| `lowValue` | An array of `floats`, each element of which specifies the lowest gray or color level that will be checked for in one band of the image. The number of elements in the array must match the number of bands in the image. |
| `highValue` | An array of `floats`, each element of which specifies the highest gray or color level that will be checked for in one band of the image. The number of elements in the array must match the number of bands in the image. |

For an example histogram, see Listing 9-3 on page 315.

**API:** `javax.media.jai.Histogram`

- `Histogram(int[] numBins, float[] lowValue, float[] highValue)`
  constructs a `Histogram` that may be used to accumulate data within a given range for each band of an image. The legal pixel range and the number of bins may be controlled separately.

| *Parameters*: | | |
|---|---|---|
| | `numBins` | The number of bins for each band of the image; `numBins.length` must be equal to the number of bands of the image which the histogram is taken. |
| | `lowValue` | The lowest pixel value checked for each band. |
| | `highValue` | The highest pixel value checked for each band. Note when counting the pixel values, this `highValue` is not included based on the formula below. |

If `binWidth` is defined as (`highValue` – `lowValue`)/`numBins`, bin i will count pixel values in the range from

$$\text{lowValue} + i \cdot \text{binWidth} \le x < \text{lowValue} + (i + 1) \cdot \text{binWidth}$$

## 9.4.2 Performing the Histogram Operation

Once you have created the `Histogram` object to accumulate the histogram information, you generate the histogram for an image with the `histogram` operation. The `histogram` operation scans a specified region of an image and generates a histogram based on the pixel values within that region of the image. The region of interest does not have to be a rectangle. If no region is specified (null), the entire image is scanned to generate the histogram. The image data passes through the operation unchanged.

The `histogram` operation takes one rendered source image and four parameters:

| Parameter | Type | Description |
|---|---|---|
| `specification` | `Histogram` | The specification for the type of histogram to be generated. See Section 9.4.1, “Specifying the Histogram.” |
| `roi` | `ROI` | The region of the image to scan. See Section 6.2, “Region of Interest Control.” |

| Parameter | Type | Description |
|---|---|---|
| `xPeriod` | `Integer` | The horizontal sampling rate. May not be less than 1. |
| `yPeriod` | `Integer` | The vertical sampling rate. May not be less than 1. |

The set of pixels scanned may be further reduced by specifying the `xPeriod` and `yPeriod` parameters that represent the sampling rate along each axis. These variables may not be less than 1. However, they may be null, in which case the sampling rate is set to 1; that is, every pixel in the ROI is processed.

### 9.4.3 Reading the Histogram Data

The histogram data is stored in the user supplied `Histogram` object, and may be retrieved by calling the `getProperty` method on this operation with "`histogram`" as the property name. The return value will be of type `Histogram`.

Several get methods allow you to check on the four histogram parameters:

- The bin data for all bands (`getBins`)
- The bin data for a specified band (`getBins`)
- The number of pixel values found in a given bin for a given band (`getBinSize`)
- The lowest pixel value found in a given bin for a given band (`getBinLowValue`)

The set of pixels counted in the histogram may be limited by the use of a region of interest (ROI), and by horizontal and vertical subsampling factors. These factors allow the accuracy of the histogram to be traded for speed of computation.

**API:** `javax.media.jai.Histogram`

- `int[][] getBins()`

  returns the bins of the histogram for all bands.

- `int[] getBins(int band)`

  returns the bins of the histogram for a specified band.

  *Parameters*: `band` The band to be checked

- `int getBinSize(int band, int bin)`

  returns the number of pixel values found in a given bin for a given band.

| | | |
|---|---|---|
| *Parameters*: | `band` | The band to be checked |
| | `bin` | The bin to be checked |

- `float getBinLowValue(int band, int bin)`

  returns the lowest pixel value found in a given bin for a given band.

| | | |
|---|---|---|
| *Parameters*: | `band` | The band to be checked |
| | `bin` | The bin to be checked |

- `void clearHistogram()`

  resets the counts of all bins to zero.

-
  ```
  void countPixels(java.awt.image.Raster pixels, ROI roi,
      int xStart, int yStart, int xPeriod, int yPeriod)
  ```

  adds the pixels of a `Raster` that lie within a given region of interest (ROI) to the histogram. The set of pixels is further reduced by subsampling factors in the horizontal and vertical directions. The set of pixels to be accumulated may be obtained by intersecting the grid

  $$(\text{xStart} + i \cdot \text{xPeriod}, \text{yStart} + j \cdot \text{yPeriod});\ \text{i, j} \geq 0$$

  with the region of interest and the bounding rectangle of the `Raster`.

| | | |
|---|---|---|
| *Parameters*: | `pixels` | A Raster containing pixels to be histogrammed. |
| | `roi` | The region of interest, as a ROI. |
| | `xStart` | The initial *x* sample coordinate. |
| | `yStart` | The initial *y* sample coordinate. |
| | `xPeriod` | The *x* sampling rate. |
| | `yPeriod` | The *y* sampling rate. |

### 9.4.4 Histogram Operation Example

Listing 9-3 shows a sample listing for a histogram operation on a three-banded source image.

**Listing 9-3 Example Histogram Operation**

```
// Set up the parameters for the Histogram object.
int[] bins = {256, 256, 256};            // The number of bins.
double[] low = {0.0D, 0.0D, 0.0D};        // The low value.
double[] high = {256.0D, 256.0D, 256.0D}; // The high value.

// Construct the Histogram object.
Histogram hist = new Histogram(bins, low, high);

// Create the parameter block.
ParameterBlock pb = new ParameterBlock();
pb.addSource(image);               // Specify the source image
pb.add(hist);                      // Specify the histogram
pb.add(null);                      // No ROI
pb.add(1);                         // Sampling
pb.add(1);                         // periods

// Perform the histogram operation.
dst = (PlanarImage)JAI.create("histogram", pb, null);

// Retrieve the histogram data.
hist = (Histogram) dst.getProperty("histogram");

// Print 3-band histogram.
for (int i=0; i< histogram.getNumBins(); i++) {
   System.out.println(hist.getBinSize(0, i) + " " +
                      hist.getBinSize(1, i) + " " +
                      hist.getBinSize(2, i) + " " +
}
```

## 9.5 Edge Detection

Edge detection is useful for locating the boundaries of objects within an image. Any abrupt change in image frequency over a relatively small area within an image is defined as an edge. Image edges usually occur at the boundaries of objects within an image, where the amplitude of the object abruptly changes to the amplitude of the background or another object.

The `GradientMagnitude` operation is an edge detector that computes the magnitude of the image gradient vector in two orthogonal directions. This operation is used to improve an image by showing the directional information only for those pixels that have a strong magnitude for the brightness gradient.

- It performs two convolution operations on the source image. One convolution detects edges in one direction, the other convolution detects edges the orthogonal direction. These two convolutions yield two intermediate images.
- It squares all the pixel values in the two intermediate images, yielding two more intermediate images.
- It takes the square root of the last two images forming the final image.

The result of the `GradientMagnitude` operation may be defined as:

$$\text{dst}[x][y][b] = \sqrt{(\text{SH}(x,y,b))^2 + (\text{SV}(x,y,b))^2}$$

where `SH(x,y,b)` and `SV(x,y,b)` are the horizontal and vertical gradient images generated from band *b* of the source image by correlating it with the supplied orthogonal (horizontal and vertical) gradient masks.

The `GradientMagnitude` operation uses two gradient masks; one for passing over the image in each direction. The `GradientMagnitude` operation takes one rendered source image and two parameters.

| Parameter | Type | Description |
|---|---|---|
| `mask1` | `KernelJAI` | A gradient mask. |
| `mask2` | `KernelJAI` | A gradient mask orthogonal to the first one. |

The default masks for the `GradientMagnitude` operation are:

- `KernelJAI.GRADIENT_MASK_SOBEL_HORIZONTAL`
- `KernelJAI.GRADIENT_MASK_SOBEL_VERTICAL`

These masks, shown in Figure 9-2 perform the Sobel edge enhancement operation. The Sobel operation extracts all of the edges in an image, regardless of the direction. The resulting image appears as an omnidirectional outline of the objects in the original image. Constant brightness regions are highlighted.

| | | |
|---|---|---|
| –1.0 | –2.0 | –1.0 |
| 0.0 | 0.0 | 0.0 |
| 1.0 | 2.0 | 1.0 |

Vertical mask

| | | |
|---|---|---|
| 1.0 | 0.0 | –1.0 |
| 2.0 | 0.0 | –2.0 |
| 1.0 | 0.0 | –1.0 |

Horizontal mask

**Figure 9-2 Sobel Edge Enhancement Masks**

The Roberts' cross edge enhancement operation uses the two masks shown in Figure 9-3. This operation extracts edges in an image by taking the combined differences of directions at right angles to each other to determine the gradient. The resulting image appears as a fairly-coarse directional outline of the objects within the image. Constant brightness regions become black and changing brightness regions become highlighted. The following is a listing of how the two masks are constructed.

```
float[] roberts_h_data        = { 0.0F,  0.0F, -1.0F,
                                  0.0F,  1.0F,  0.0F,
                                  0.0F,  0.0F,  0.0F
};
float[] roberts_v_data        = {-1.0F,  0.0F,  0.0F,
                                  0.0F,  1.0F,  0.0F,
                                  0.0F,  0.0F,  0.0F
};

KernelJAI kern_h = new KernelJAI(3,3,roberts_h_data);
KernelJAI kern_v = new KernelJAI(3,3,roberts_v_data);
```

| | | |
|---|---|---|
| –1.0 | 0.0 | 0.0 |
| 0.0 | 1.0 | 0.0 |
| 0.0 | 0.0 | 0.0 |

Vertical mask

| | | |
|---|---|---|
| 0.0 | 0.0 | –1.0 |
| 0.0 | 1.0 | 0.0 |
| 0.0 | 0.0 | 0.0 |

Horizontal mask

**Figure 9-3 Roberts' Cross Edge Enhancement Masks**

The Prewitt gradient edge enhancement operation uses the two masks shown in Figure 9-4. This operation extracts the north, northeast, east, southeast, south, southwest, west, or northwest edges in an image. The resulting image appears as a directional outline of the objects within the image. Constant brightness regions become black and changing brightness regions become highlighted. The following is a listing of how the two masks are constructed.

```
float[] prewitt_h_data        = { 1.0F,  0.0F, -1.0F,
                                  1.0F,  0.0F, -1.0F,
                                  1.0F,  0.0F, -1.0F
};
float[] prewitt_v_data        = {-1.0F, -1.0F, -1.0F,
                                  0.0F,  0.0F,  0.0F,
                                  1.0F,  1.0F,  1.0F
};

KernelJAI kern_h = new KernelJAI(3,3,prewitt_h_data);
KernelJAI kern_v = new KernelJAI(3,3,prewitt_v_data);
```

| | | |
|---|---|---|
| –1.0 | –1.0 | –1.0 |
| 0.0 | 0.0 | 0.0 |
| 1.0 | 1.0 | 1.0 |

Vertical mask

| | | |
|---|---|---|
| 1.0 | 0.0 | –1.0 |
| 1.0 | 0.0 | –1.0 |
| 1.0 | 0.0 | –1.0 |

Horizontal mask

**Figure 9-4 Prewitt Edge Enhancement Masks**

The Frei and Chen edge enhancement operation uses the two masks shown in Figure 9-5. This operation, when compared to the other edge enhancement, operations, is more sensitive to a configuration of relative pixel values independent of the brightness magnitude. The following is a listing of how the two masks are constructed.

```
   float[] freichen_h_data       = { 1.0F,   0.0F,   -1.0F,
                                     1.414F, 0.0F,   -1.414F,
                                     1.0F,   0.0F,   -1.0F
   };
   float[] freichen_v_data       = {-1.0F,  -1.414F, -1.0F,
                                     0.0F,   0.0F,    0.0F,
                                     1.0F,   1.414F,  1.0F
   };

   KernelJAI kern_h = new KernelJAI(3,3,freichen_h_data);
   KernelJAI kern_v = new KernelJAI(3,3,freichen_v_data);
```

| | | |
|---|---|---|
| –1.0 | –1.414 | –1.0 |
| 0.0 | 0.0 | 0.0 |
| 1.0 | 1.414 | 1.0 |

Vertical mask

| | | |
|---|---|---|
| 1.0 | 0.0 | –1.0 |
| 1.414 | 0.0 | –1.414 |
| 1.0 | 0.0 | –1.0 |

Horizontal mask

**Figure 9-5 Frei and Chen Edge Enhancement Masks**

To use a different mask, see Section 6.9, "Constructing a Kernel."

Listing 9-4 shows a sample listing for a `GradientMagnitude` operation, using the Frei and Chen edge detection kernel.

**Listing 9-4 Example GradientMagnitude Operation**

```
     // Load the image.
     PlanarImage im0 = (PlanarImage)JAI.create("fileload",
                                          filename);

     // Create the two kernels.
     float data_h[] = new float[] { 1.0F,   0.0F,   -1.0F,
                                    1.414F, 0.0F,   -1.414F,
                                    1.0F,   0.0F,   -1.0F};
     float data_v[] = new float[] {-1.0F,  -1.414F, -1.0F,
                                    0.0F,   0.0F,    0.0F,
                                    1.0F,   1.414F,  1.0F};

     KernelJAI kern_h = new KernelJAI(3,3,data_h);
     KernelJAI kern_v = new KernelJAI(3,3,data_v);

     // Create the Gradient operation.
     PlanarImage im1 =
             (PlanarImage)JAI.create("gradientmagnitude", im0,
                                   kern_h, kern_v);

     // Display the image.
     imagePanel = new ScrollingImagePanel(im1, 512, 512);
             add(imagePanel);
             pack();
             show();
```

## 9.6 Statistical Operations

The `StatisticsOpImage` class is an abstract class for image operators that compute statistics on a given region of an image and with a given sampling rate. A subclass of `StatisticsOpImage` simply passes pixels through unchanged from its parent image. However, the desired statistics are available as a property or set of properties on the image (see Chapter 11, "Image Properties").

All instances of `StatisticsOpImage` make use of a region of interest, specified as an `ROI` object. Additionally, they may perform spatial subsampling of the region of interest according to `xPeriod` and `yPeriod` parameters that may vary from 1 (sample every pixel of the `ROI`) upwards. This allows the speed and quality of statistics gathering to be traded off against one another.

The `accumulateStatistics` method is used to accumulate statistics on a specified region into the previously-created statistics object.

**API:** `javax.media.jai.StatisticsOpImage`

- `StatisticsOpImage()`

  constructs a default `StatisticsOpImage`.

-
  ```
  StatisticsOpImage(RenderedImage source, ROI roi, int xStart,
      int yStart, int xPeriod, int yPeriod, int maxWidth,
      int maxHeight)
  ```

  constructs a `StatisticsOpImage`. The image layout is copied from the source image.

| | | |
|---|---|---|
| *Parameters*: | `source` | A `RenderedImage`. |
| | `roi` | The region of interest, as an `ROI`. |
| | `xStart` | The initial *x* sample coordinate. |
| | `ystart` | The initial *y* sample coordinate. |
| | `xPeriod` | The *x* sampling rate. |
| | `yPeriod` | The *y* sampling rate. |
| | `maxWidth` | The largest allowed width for processing. |
| | `maxHeight` | The largest allowed height for processing. |

CHAPTER 10

# Graphics Rendering

THIS chapter describes the JAI presentation of rendering shapes, text, and images.

## 10.1 Introduction

JAI provides classes that support drawing operations beyond the `Graphics2D` class. Three different types of graphics rendering are offered: simple 2D graphics, renderable graphics, and tiled image graphics. These are described in more detail in the sections that follow.

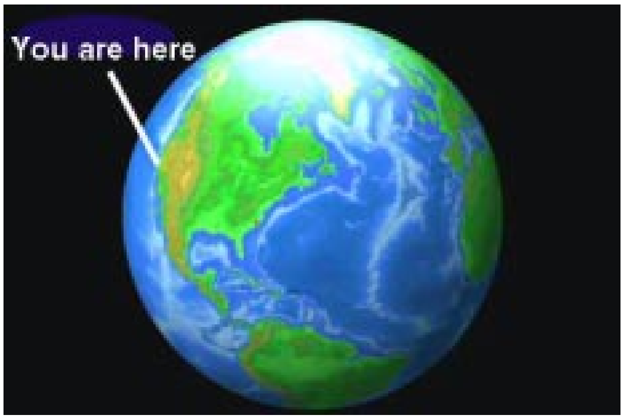


**Figure 10-1 Simple Text and Line Added to an Image**

### 10.1.1 Simple 2D Graphics

The `Graphics2D` class extends the even simpler `Graphics` class to provide more control over geometry, coordinate transformations, color management, and text layout. Graphics2D is the fundamental class for rendering two-dimensional shapes, text and images. Graphics2D supports geometric rendering by providing a mechanism for rendering virtually any geometric shape, draw styled lines of any width, and fill geometric shapes with virtually any texture.

The BufferedImage.createGraphics method creates a Graphics2D object, which can then be used to draw into this BufferedImage.

Geometric shapes are provided through implementations of the Shape interface, such as Polygon, Rectangle, CubicCurve2D, and QuadCurve2D. Fill and pen styles are provided through implementations of the Paint and Stroke interfaces. For example, the Paint interface supports Color, GradientPaint, and TexturePaint. The Stroke interface supports BasicStroke, which defines a set of attributes for the outlines of graphics primitives.

Text is added to graphics using the Font class, which represents character fonts. A Font is defined by a collections of Glyphs, which in turn are defined by individual Shapes. Since text is represented by glyphs, text strings can also be stroked and filled like other geometric objects.

### 10.1.2 Renderable Graphics

The RenderableGraphics class is an implementation of Graphics2D with RenderableImage semantics. This means that content may be drawn into the image using the Graphics2D interface and later be turned into RenderedImages with different resolutions and characteristics.

The RenderableGraphics class allows you to store a sequence of drawing commands and "replay" them at an arbitrary output resolution. By serializing an instance of RenderableGraphics, you create a kind of metafile for storing the graphical content.

The methods in the RenderableGraphics class override the methods in the java.awt.Graphics and java.awt.Graphics2D classes. This means that you can use the methods in RenderableGraphics to set your fonts and colors, to create the graphics shapes and text, define a clipping path, and so on.

The only method unique to RenderableGraphics is the createRendering method, which creates a RenderedImage that represents a rendering of the image using a given RenderContext. This is the most general way to obtain a rendering of a RenderableImage.

## 10.2 A Review of Graphics Rendering

To render a graphic object, you set up the `Graphics2D` context and pass the graphic object to one of the `Graphics2D` rendering methods. Before rendering the graphic object, you first need to set certain state attributes that define how the `Graphics2D` context displays the graphics. For example, you specify:

- The stroke width
- How strokes are joined
- A clipping path to limit the area that is rendered
- Define colors and patterns to fill shapes with

`Graphics2D` defines several methods that add or change attributes in the graphics context. Most of these methods take an object that represents a particular attribute, such as a `Paint` or `Stroke` object.

### 10.2.1 Overview of the Rendering Process

When a graphic object is rendered, the geometry, image, and attribute information are combined to calculate which pixel values must be changed on the display.

The rendering process for a `Shape` is described into the following four steps:

1. If the `Shape` is to be stroked, the `Stroke` attribute in the `Graphics2D` context is used to generate a new `Shape` that encompasses the stroked path.
2. The coordinates of the `Shape`'s path are transformed from user space into device coordinate space according to the transform attribute in the `Graphics2D` context.
3. The `Shape`'s path is clipped using the clip attribute in the `Graphics2D` context.
4. The remaining `Shape` is filled using the `Paint` and `Composite` attributes in the `Graphics2D` context.

Rendering text is similar to rendering a `Shape`, since the text is rendered as glyphs and each glyph is a `Shape`. However, you still must specify what `Font` to use for the text and get the appropriate glyphs from the `Font` before rendering. The attributes are described in more detail in the following sections.

### 10.2.2 Stroke Attributes

The Graphics2D Stroke attribute defines the characteristics of strokes. The BasicStroke object is used to define the stroke attributes for a Graphics2D context. BasicStroke defines characteristics such as line width, endcap style, segment join style, and pattern (solid or dashing). To change the Stroke attribute in the Graphics2D context, you call the setStroke method.

#### 10.2.2.1 Line Width

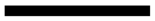
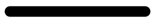
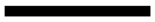
The line width is specified in *points* (there are 72 points to the inch). To set the stroke width, create a BasicStroke object with the desired width and call setStroke. The following example sets the stroke width to 12 points.

```
   wideStroke = new BasicStroke(12.0);
   g2.setStroke(wideStroke);
```

#### 10.2.2.2 Endcap Style

Table 10-1 lists the endcap style attributes.

**Table 10-1 Endcap Styles**

| Appearance | Attribute | Description |
|---|---|---|
|  | CAP_BUTT | Ends unclosed subpaths with no added decoration. |
|  | CAP_ROUND | Ends unclosed subpaths with a round end cap that has a radius equal to half the pen width. |
|  | CAP_SQUARED | Ends unclosed subpaths with a square projection that extends beyond the end of the segment to a distance equal to half the line width. |

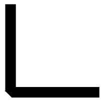
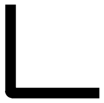
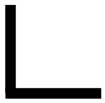
To set the endcap style, create a BasicStroke object with the desired attribute. The following example sets the stroke width to 12 points and endcap style is set to CAP_ROUND.

```
   wideStroke = new BasicStroke(12.0, BasicStroke.CAP_ROUND);
   g2.setStroke(roundStroke);
```

### 10.2.2.3 Join Style

Table 10-2 lists the join style attributes. These attributes affect the appearance of line junctions.

**Table 10-2 Join Styles**

| Appearance | Attribute | Description |
|---|---|---|
|  | `JOIN_BEVEL` | Joins path segments by connecting the outer corners of their wide outlines with a straight segment. |
|  | `JOIN_ROUND` | Joins path segments by rounding off the corner at a radius of half the line width. |
|  | `JOIN_MITER` | Joins path segments by extending their outside edges until they meet. |

To set the join style, create a `BasicStroke` object with the desired attribute. The following example sets the stroke width to 12 points, an endcap style of `CAP_ROUND`, and a join style of `JOIN_ROUND`.

```
   wideStroke = new BasicStroke(12.0, BasicStroke.CAP_ROUND,
                                 BasicStroke.JOIN_ROUND);
   g2.setStroke(roundStroke);
```

### 10.2.2.4 Stroke Style

The stroke style is defined by two parameters:

- `dash` – an array that represents the dashing pattern. Alternating elements in the array represent the dash size and the size of the space between dashes. Element 0 represents the first dash, element 1 represents the first space.
- `dash_phase` – an offset that defines where the dashing pattern starts.

Listing 10-1 shows a code sample in which two different dashing patterns are created. In the first pattern, the size of the dashes and the space between them is constant. The second pattern uses a six-element array to define the dashing pattern. The two dash patterns are shown in Figure 10-2.

**Listing 10-1 Example Stroke Styles**

```
   // Define the first dashed line.
   float dash1[] = {10.0f};
   BasicStroke bs = new BasicStroke(5.0f, BasicStroke.CAP_BUTT,
                       BasicStroke.JOIN_MITER, 10.0f, dash1, 0.0f);

   g2.setStroke(bs);
   Line2D line = new Line2D.Float(20.0f, 10.0f, 100.0f, 10.0f);
   g2.draw(line);

   // Define the second dashed line.
   float[] dash2 = {6.0f, 4.0f, 2.0f, 4.0f, 2.0f, 4.0f};
   bs = new BasicStroke(5.0f, BasicStroke.CAP_BUTT,
                       BasicStroke.JOIN_MITER, 10.0f, dash2, 0.0f);
   g2.setStroke(bs);
   g2.draw(line);
```

dash1
dash2

**Figure 10-2 Example Stroke Styles**

### 10.2.2.5 Fill Styles

The `Paint` attribute in the `Graphics2D` context defines the fill color or pattern used when text and `Shapes` are rendered.

#### *Filling a Shape with a Gradient*

The `GradientPaint` class allows a shape to be filled with a gradient of one color to another. When creating a `GradientPaint` object, you specify a beginning position and color, and an ending position and color. The fill color changes proportionally from one color to the other along the line connecting the two positions, as shown in Figure 10-3.

In all three stars, the gradient line extends from point P1 to point P2. In the middle star, all of the points along the gradient line extending to the left of P1 take the beginning color and the points to the right of P2 take the ending color.

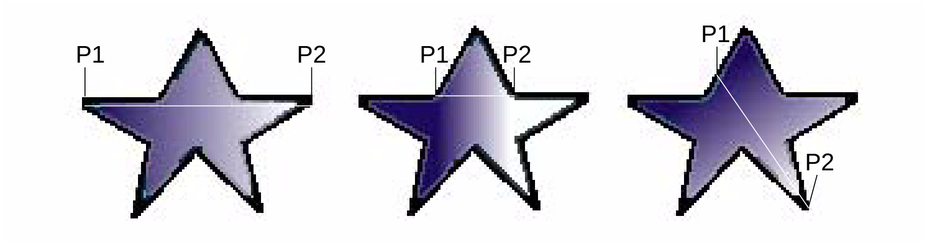


**Figure 10-3 Filling a Shape with a Gradient**

To fill a shape with a gradient of one color to another:

1. Create a `GradientPaint` object
2. Call `Graphics2D.setPaint`
3. Create the `Shape` object
4. Call `Graphics2D.fill(shape)`

Listing 10-2 shows sample code in which a rectangle is filled with a blue-green gradient.

**Listing 10-2 Example Filling a Rectangle with a Gradient**

```
GradientPaint gp = new GradientPaint(50.0f, 50.0f, Color.blue,
                                     50.0f, 250.0f, Color.green);
g2.setPaint(gp);
g2.fillRect(50, 50, 200, 200);
```

### *Filling a Shape with a Texture*

The `TexturePaint` class allows you to fill a shape with a repeating pattern. When you create a `TexturePaint`, you specify a `BufferedImage` to use as the pattern. You also pass the constructor a rectangle to define the repetition frequency of the pattern.

To fill a shape with a texture:

1. Create a `TexturePaint` object
2. Call `Graphics2D.setPaint`
3. Create the `Shape`
4. Call `Graphics2D.fill(shape)`

Listing 10-3 shows sample code in which a shape is filled with texture.

**Listing 10-3 Example Filling a Shape with Texture**

```
// Create a buffered image texture patch of size 5 X 5.
BufferedImage bi = new BufferedImage(5, 5,
                          BufferedImage.TYPE_INT_RGB);
Graphics2D big bi.createGraphics();

// Render into the BufferedImage graphics to create the texture.
big.setColor(Color.green);
big.fillRect(0, 0, 5, 5);
big.setColor(Color.lightGray);
big.fillOval(0, 0, 5, 5);

// Create a texture paint from the buffered image.
Rectangle r = new Rectangle(0, 0, 5, 5);
TexturePaint tp = new
    TexturePaint(bi, r, TexturePaint.NEAREST_NEIGHBOR);

// Add the texture paint to the graphics context.
g2.setPaint(tp);

// Create and render a rectangle filled with the texture.
g2.fillRect(0, 0, 200, 200);
}
```

## 10.2.3 Rendering Graphics Primitives

The `Graphics2D` class provides methods for creating `Shapes` and `Text`, and for rendering `Images`. Table 10-3 lists these methods.

**Table 10-3 Graphics Primitives Methods**

| Method | Description |
|---|---|
| `draw` | Strokes the outline of a `Shape` using the `Stroke` and `Paint` settings of the current `Graphics2D` context. |
| `fill` | Fills the interior of a `Shape` using the `Paint` settings of the `Graphics2D` context. |
| `drawString` | Renders the specified text string using the `Paint` setting of the `Graphics2D` context. |
| `drawImage` | Renders the specified `Image`. |
| `drawRenderableImage` | Renders the specified `RenderableImage`. |
| `drawRenderedImage` | Renders the specified `RenderedImage`. |

### 10.2.3.1 Drawing a Shape

The `Graphics2D.draw` method is used to render the outline of any `Shape`. The `Graphics2D` class also inherits draw methods from the `Graphics` class, such as `drawLine`, `drawRect`, `drawRoundRect`, `drawOval`, `drawArc`, `drawPolyline`, `drawPolygon`, and `draw3DRect`.

When a `Shape` is drawn, its path is stroked with the `Stroke` object in the `Graphics2D` context. (See Section 10.2.2, “Stroke Attributes,” for more information.) By setting an appropriate `BasicStroke` object in the `Graphics2D` context, you can draw lines of any width or pattern. The `BasicStroke` object also defines the line’s endcap and join attributes.

To render a `Shape`’s outline:

1. Create the `BasicStroke` object
2. Call `Graphics2D.setStroke`
3. Create the `Shape`
4. Call `Graphics2D.draw(shape)`

Listing 10-4 shows a code example in which a `GeneralPath` object is used to define a star and a `BasicStroke` object is added to the `Graphics2D` context to define the star’s line width and join attributes.

**Listing 10-4 Example Drawing a Shape**

```
public void paint(Graphics g) {
   Graphics2D g2 = (Graphics2D) g;

   // Create and set the stroke.
   g2.setStroke(new BasicStroke(4.0f));

   // Create a star using a general path object.
   GeneralPath p new GeneralPath(GeneralPath.NON_ZERO);
   p.moveTo(- 100.0f, - 25.0f);
   p.lineTo(+ 100.0f, - 25.0f);
   p.lineTo(- 50.0f, + 100.0f);
   p.lineTo(+ 0.0f, - 100.0f);
   p.lineTo(+ 50.0f, + 100.0f);
   p.closePath();

   // Translate the origin towards the center of the canvas.
   g2.translate(100.0f, 100.0f);
```

**Listing 10-4 Example Drawing a Shape (Continued)**

```
      // Render the star’s path.
      g2.draw(p);
   }
```

### 10.2.3.2 Filling a Shape

The `Graphics2D.fill` method is used to fill any `Shape`. When a `Shape` is filled, the area within its path is rendered with the `Paint` object in the Graphics2D context: a `Color`, `TexturePaint`, or `GradientPaint`.

The `Graphics2D` class also inherits fill methods from the `Graphics` class, such as `fillRect`, `fill3DRect`, `fillRoundRect`, `FillOval`, `fillArc`, `fillPolygon`, and `clearRect`.

To fill a `Shape`:

1. Set the fill color or pattern on the `Graphics2D` context using `Graphics2D.setColor`, or `Graphics2DsetPaint`.
2. Create the `Shape`
3. Call `Graphics2D.fill` to render the `Shape`

Listing 10-5 shows a code example in which the `setColor` method is called to define a green fill for a `Rectangle2D`.

**Listing 10-5 Example Filling a Shape**

```
   Public void paint(Graphics g) {
      Graphics2D g2 = (Graphics2D) g;

      g2.setpaint(Color.green);
      Rectangle2D r2 = new Rectangle2D.float(25, 25, 150, 150);

      g2.fill(r2);
   }
```

### 10.2.3.3 Rendering Text

The entire subject of fonts and text layout is too extensive to try to describe here. In this section, we’ll give a brief overview of the `Graphics2D.drawString` method, which is used to render a text string.

There are two basic variations on the `drawString` method. Two methods takes a `String` for an argument and two methods take an `AttributedCharacterIterator`. If the argument is a `String`, the current `Font` in the `Graphics2D` context is used to convert the characters in the `String` into a set of glyphs with whatever basic layout and shaping algorithms the font implements. If the argument is an `AttributedCharacterIterator`, the iterator is asked to convert itself to a `TextLayout` using its embedded font attributes. The `TextLayout` implements more sophisticated glyph layout algorithms that perform Unicode I-directional layout adjustments automatically for multiple fonts of differing writing directions.

A third method used to render text is the `Graphics2D.drawGlyphVector` method, which takes a `GlyphVector` as an argument. The `GlyphVector` object contains the appropriate font-specific glyph codes with explicit coordinates for the position of each glyph.

The character outlines are filled with the `Paint` object in the Graphics2D context.

## 10.3 Graphics2D Example

Listing 10-6 shows a code sample for a Graphics2D example.

**Listing 10-6 Graphics2D Example**

```
   // Read a RenderedImage and convert it to a BufferedImage.
   imagePath = new String("./images/sample.jpg");
   Image ai = loadAWTImage(imagePath, this);
   RenderedImage ri = JAI.create("awtimage", ai);
   BufferedImage bi = getBufferedImage(ri);
   RenderedImage targetImage = null;
   targetImage = new BufferedImage(bi.getWidth(),
                                   bi.getHeight(),
                                   bi.getType());

   // Create a Graphics2D object to draw into the BufferedImage.
   Graphics2D g2d = targetImage.createGraphics();
```

## 10.4 Adding Graphics and Text to an Image

The `java.awt.Graphics2D` class enables you to draw lines, geometric shapes, images, and text. These objects can then be “painted” over a `TiledImage`.

CHAPTER 11

# Image Properties

THIS chapter describes image properties.

## 11.1 Introduction

In addition to the pixel data, images occasionally have many other kinds of data associated with them. These data, known as *properties*, is a simple database of arbitrary data attached to the images. Each property is simply an Object with a unique, case-insensitive name.

The properties are arbitrary and depend on the intended application. JAI provides methods that enable effective use of properties in the context of an image processing application but, in most cases, leaves the specification of the property objects themselves to the developer.

Some examples of properties are:

- Descriptions of exotic shapes, such as hexagonal grids
- Mapping from digital pixel values to photometric values
- A defined region of interest (ROI) in the source image

Every node in an image chain may be queried for its properties. The value of a property at a particular node may be derived by one of the following mechanisms:

- It may be *copied* from the node's sources. This is the default behavior if no other behavior is specified.
- It may be *produced* by the node from non-property information available in the node.
- It may be *synthesized* by the node from a rendering.

- It may be *inherited* or produced computationally from the properties of the node's sources.
- It may be *set explicitly* by the `setProperty` method in one of the appropriate classes: `Planarimage`, `RenderedOp`, or `RenderableOp`. Properties of a node may not be set once the node has been rendered.

When the value of a property is requested from a node in a rendered chain, i.e., a `RenderedOp` node, it will be derived from the first of the following for which it is defined:

1. Synthetic properties (see below).
2. Local properties, i.e., those set by an invocation of setProperty() on the node.
3. The source image of the operation specified by invoking the method `OperationRegsitry.copyPropertyFromSource()`.
4. The rendering of the node. Note however that properties set by invoking `setProperty()` on the rendering of the node rather than on the node itself will not be propagated back to the node itself.
5. Any PropertyGenerators either defined by the associated operation or added by an invocation of `RenderedOp.addPropertyGenerator()`. PropertyGenerators added by the latter method supersede those associated with the operation, e.g., via its OperationDescriptor.
6. The sources of the operation. The first source has higher precedence than the second source and so on.

The same order of precedence applies in the case of renderable chains, i.e., RenderableOp nodes, with the exception of item 4, viz., properties created within the contextual rendering of the RenderableOp are not propagated back to the RenderableOp node itself.

There are a couple of important items to note at this point. First, when a node is created with another node or nodes as its source(s), it might invoke methods on the source node that force the source node to be rendered. Consequently properties should be set on a node before it is used as the source of other operations. Second, the rendering of a node does *not* inherit the properties of the node itself nor are properties set on the rendering of the node propagated back to the node. Image properties are controlled and generated by the `PropertySource` and `PropertyGenerator` interfaces.

### 11.1.1 The PropertySource Interface

The `PropertySource` interface contains methods from the `RenderedImage` and `RenderableImage` interfaces that identify and read properties. `PlanarImage`, `RenderableOp`, and `RenderedOp` all implement `PropertySource`.

The interface consists of the `getProperty` and `getPropertyNames` methods familiar from the `RenderedImage` and `RenderableImage` interfaces.

`PropertySource` is implemented by `ImageJAI`. Since all RenderedImages used within JAI are descendents of `PlanarImage` which implements `ImageJAI`, all images may be assumed to implement `PropertySource`.

**API:** `javax.media.jai.PropertySource`

- `String[] getPropertyNames()`

  returns an array of `Strings` recognized as names by this property source.

- `String[] getPropertyNames(String prefix)`

  returns an array of `Strings` recognized as names by this property source that begin with the supplied `prefix`. If the method cannot find any property names that match, null is returned.

- `Object getProperty(String name)`

  returns the value of a property.

| | | |
|---|---|---|
| *Parameters*: | `name` | The name of the property, as a `String`. |

### 11.1.2 The PropertyGenerator Interface

The `PropertyGenerator` interface allows you to affect the property inheritance computation of an operation. A `PropertyGenerator` simply implements two methods:

- The `getPropertyNames` method returns a list of the names of all available properties.
- The `getProperty` method returns the value of the property, given the property name and a `RenderedOp`.

New `PropertyGenerators` may be added to the `OperationRegistry` to be applied at a particular operation node. The `OperationRegistry` also allows an existing property on a node to be suppressed if it is no longer useful. See

Chapter 14, “Extending the API,” for more information on the `OperationRegistry`.

**API:** `javax.media.jai.PropertyGenerator`

- `String[] getPropertyNames()`

  returns an array of `Strings` naming properties emitted by this property generator.

- `Object getProperty(String name, RenderedOp op)`

  computes the value of a property relative to an environment of pre-existing properties emitted by the sources of a `RenderedOp`, and the parameters of that operation.

| | | |
|---|---|---|
| *Parameters*: | `name` | The name of the property, as a `String`. |
| | `op` | The `RenderedOp` representing the operation. |

  The operation name, sources, and `ParameterBlock` of the `RenderedOp` being processed may be obtained by means of the `op.getOperationName`, `op.getSources()`, and `op.getParameterBlock()` methods. It is legal to call `getProperty()` on the operation’s sources.

## 11.2 Synthetic Properties

Certain properties are *synthesized* when a node is rendered. These synthetic properties are image width (`image_width`), image height (`image_height`), minimum *x* coordinate (`image_min_x_coord`), and minimum *y* coordinate (`image_min_y_coord`). All of these properties have a value of class `java.lang.Integer`. These properties are fixed and any attempt to set them will result in an error.

## 11.3 Regions of Interest

The specification of a region of interest (ROI) is a common property that is supported by all of the standard operators. The ROI is simply a description of some portion of an image. This description is propogated, along with the image, through the rendering chain. The ROI is transformed appropriately (inherited) for all geometric and area operators. For all other types of operations it is simply copied. The ROI has no bearing on the processing of image pixels, although in its rendered form it can be used as input to histogram operations. For more information, see Section 6.2, "Region of Interest Control."

The ROI may be used as an argument to the `TiledImage.set` and `TiledImage.setData` methods so as to copy a selected area of a source or `Raster` into an existing `TiledImage` (see Section 4.2.2, "Tiled Image"). The ROI may also be used as an argument to many compositing (see Section 7.11.2, "Image Compositing") and statistical operators (see Chapter 9, "Image Analysis").

## 11.4 Complex Data

The COMPLEX property has value of class `java.lang.Boolean` and indicates whether the pixel values of an image represent complex-value data. (A complex-valued image wherein each pixel has N complex elements contains 2N bands with the real and imaginary components of the *i*th complex element being stored in bands 2i and 2i + 1, respectively.) This property may be *produced* by a given node either with a fixed value or with a value dependent on the parameters of the node. See Section 7.9, "Frequency Domain Processing."

CHAPTER 12

# Client-Server Imaging

THIS chapter describes JAI's client-server imaging system.

## 12.1 Introduction

Client-server imaging provides the ability to distribute computation between a set of processing nodes. For example, it is possible to set up a large, powerful server that provides image processing services to several thin clients. With JAI, it is possible for a client to set up a complex imaging chain on a server, including references to source images on other network hosts, and to request rendered output from the server.

JAI uses Java Remote Method Invocation (RMI) to implement client-server imaging. To communicate using Remote Method Invocation, both the client and server must be running Java. A *stub* object is instantiated on the client. The stub object forwards its method calls to a corresponding server object. Method call arguments and return values are transmitted between the client and server by means of the Java Development Environment's *serialization* capability.

The hostname and port depend on the local setup. The host must be running an RMI registry process and have a `RemoteImageServer` listening at the desired port.

This call will result in the creation of a server-side `RMIImageImpl` object and a client-side stub object. The client stub serializes its method arguments and transfers them to the server over a socket; the server serializes its return values and returns them in the same manner.

## 12.2 Server Name and Port Number

The `RemoteImage` constructor requires a `serverName` parameter that consists of a host name and port number, in the following format:

```
host:port
```

For example:

```
camus.eng.sun.com:1099
```

The port number is optional and need be supplied only if the host name is supplied. If the `serverName` parameter is null, the default is to search for the RMIImage service on the local host at the default *rmiregistry* port (1099.

**API:** `javax.media.jai.RemoteImage`

- `RemoteImage(String serverName, RenderedImage source)`

  constructs a `RemoteImage` from a `RenderedImage`.

  | *Parameters*: | | |
  |---|---|---|
  | | `serverName` | The name of the server in the appropriate format. |
  | | `source` | A `RenderedImage` source. |

- `RemoteImage(String serverName, RenderedOp source)`

  constructs a `RemoteImage` from a `RenderedOp`, i.e., an imaging DAG (directed acyclic graph). Note that the properties of the `RemoteImage` will be those of the `RenderedOp` node and not of its rendering.

-
  ```
  RemoteImage(String serverName, RenderableOp source,
      RenderContext renderContext)
  ```

  constructs a `RemoteImage` from a `RenderableOp` and `RenderContext`. The entire `RenderableOp` DAG will be copied over to the server. Note that the properties of the `RemoteImage` will be those of the `RenderableOp` node and not of its rendering.

## 12.3 Setting the Timeout Period and Number of Retries

A network error or a delay caused by the server failing to respond to the request for an image is dealt with through retries. If, on the first attempt, the server fails to respond, the program will wait a specified amount of time and then make another request for the image. When the limit of retries is exceeded, a null Raster may be returned.

The amount of time to wait between retries defaults to 1 second (1000 milliseconds). The `getTimeout` method is used to get the amount of time between retries, in milliseconds. The `setTimeout` method is used to set the amount of time between retries.

The number of times the program will attempt to read the remote image may be read with the `getNumRetries` method. The `setNumRetries` method is used to set the maximum number of retries.

**API:** `javax.media.jai.RemoteImage`

- `void setTimeout(int timeout)`

  sets the amount of time between retries.

| | | |
|---|---|---|
| *Parameter*: | `timeout` | The time interval between retries in milliseconds. |

- `int getTimeout()`

  returns the amount of time between retries.

- `void setNumRetries(int numRetries)`

  sets the number of retries.

| | | |
|---|---|---|
| *Parameter*: | `numRetries` | The maximum number of retries. If this is a negative value, the number of retries is unchanged. |

## 12.4 Remote Imaging Test Example

This section contains two examples of remote imaging programs.

### 12.4.1 Simple Remote Imaging Example

Listing 12-1 shows a complete code example of a `RemoteImaging` test. This example displays a 2 × 2 grid of `ScrollingImagePanels`, with each window displaying the sum of two byte images that were rescaled to the range [0,127] prior to addition. The panels display the following specific results:

- upper left: local rendering
- upper right: result of remote rendering of a RenderedOp graph
- lower left: result of remote loading of a RenderedImage

- lower right: result of remote rendering of a RenderableOp graph

The lower right image is a dithered version of the sum image passed through a color cube lookup table and may appear slightly different from the other three images, which should be identical.

**Listing 12-1 Remote Imaging Example Program (Sheet 1 of 4)**

```
import java.awt.*;
import java.awt.event.WindowEvent;
import java.awt.geom.*;
import java.awt.image.*;
import java.awt.image.renderable.*;
import java.util.*;
import javax.media.jai.*;
import javax.media.jai.operator.*;
import javax.media.jai.widget.*;
public class RemoteImagingTest extends WindowContainer {

/** Default remote server. */
private static final String DEFAULT_SERVER =
                         "camus.eng.sun.com:1099";

/** Tile dimensions. */
private static final int TILE_WIDTH = 256;
private static final int TILE_HEIGHT = 256;

public static void main(String args[]) {
String fileName1 = null;
String fileName2 = null;

// Check args.
    if(!(args.length >= 0 && args.length <= 3)) {
       System.out.println("\nUsage: java RemoteImagingTest "+
                 "[[[serverName] | [fileName1 fileName2]] | "+
                   "[serverName fileName1 fileName2]]"+"\n");
       System.exit(1);
    }

// Set the server name.
String serverName = null;
   if(args.length == 0 || args.length == 2) {
       serverName = DEFAULT_SERVER;
       System.out.println("\nUsing default server '"+
                          DEFAULT_SERVER+"'\n");
   } else {
       serverName = args[0];
   }
```

**Listing 12-1 Remote Imaging Example Program (Sheet 2 of 4)**

```
// Set the file names.
   if(args.length == 2) {
       fileName1 = args[0];
       fileName2 = args[1];
   } else if(args.length == 3) {
       fileName1 = args[1];
       fileName2 = args[2];
   } else {
 fileName1 = "/import/jai/JAI_RP/test/images/Boat_At_Dock.tif";
 fileName2 = "/import/jai/JAI_RP/test/images/FarmHouse.tif";
       System.out.println("\nUsing default images '"+
                      fileName1 + "' and '" + fileName2 + "'\n");
     }

RemoteImagingTest riTest =
       new RemoteImagingTest(serverName, fileName1, fileName2);
    }

/**
* Run a remote imaging test.
*
* @param serverName The name of the remote server to use.
* @param fileName1 The first addend image file to use.
* @param fileName2 The second addend image file to use.
*/
RemoteImagingTest(String serverName, String fileName1, String
                  fileName2) {
// Create the operations to load the images from files.
RenderedOp src1 = JAI.create("fileload", fileName1);
RenderedOp src2 = JAI.create("fileload", fileName2);

// Render the sources without freezing the nodes.
PlanarImage ren1 = src1.createInstance();
PlanarImage ren2 = src2.createInstance();
```

**Listing 12-1 Remote Imaging Example Program (Sheet 3 of 4)**

```
// Create TiledImages with the file images as their sources
// thereby ensuring that the serialized images are truly tiled.
   SampleModel sampleModel1 =
ren1.getSampleModel().createCompatibleSampleModel(TILE_WIDTH,
                                                  TILE_HEIGHT);
TiledImage ti1 = new TiledImage(ren1.getMinX(), ren1.getMinY(),
                             ren1.getWidth(), ren1.getHeight(),
                                ren1.getTileGridXOffset(),
                                ren1.getTileGridYOffset(),
                           sampleModel1, ren1.getColorModel());
ti1.set(src1);
SampleModel sampleModel2 =
  ren2.getSampleModel().createCompatibleSampleModel(TILE_WIDTH,
                                                  TILE_HEIGHT);
 TiledImage ti2 = new TiledImage(ren2.getMinX(), ren2.getMinY(),
                             ren2.getWidth(), ren2.getHeight(),
                                 ren2.getTileGridXOffset(),
                                 ren2.getTileGridYOffset(),
                           sampleModel2, ren2.getColorModel());
  ti2.set(src2);

// Create a hint to specify the tile dimensions.
ImageLayout layout = new ImageLayout();
layout.setTileWidth(TILE_WIDTH).setTileHeight(TILE_HEIGHT);
     RenderingHints rh = new
         RenderingHints(JAI.KEY_IMAGE_LAYOUT, layout);

// Rescale the images to the range [0, 127].
ParameterBlock pb = (new ParameterBlock());
pb.addSource(ti1);
pb.add(new double[] {0.5}).add(new double[] {0.0});
RenderedOp addend1 = JAI.create("rescale", pb, rh);
pb = (new ParameterBlock());
pb.addSource(ti2);
pb.add(new double[] {0.5}).add(new double[] {0.0});
RenderedOp addend2 = JAI.create("rescale", pb, rh);

// Add the rescaled images.
pb = (new
   ParameterBlock()).addSource(addend1).addSource(addend2);
        RenderedOp sum = JAI.create("add", pb, rh);

        // Dither the sum of the rescaled images.
        pb = (new ParameterBlock()).addSource(sum);

pb.add(ColorCube.BYTE_496).add(KernelJAI.DITHER_MASK_443);
      RenderedOp dithered = JAI.create("ordereddither", pb, rh);
```

**Listing 12-1 Remote Imaging Example Program (Sheet 4 of 4)**

```
    // Construct a RemoteImage from the RenderedOp chain.
    RemoteImage remoteImage = new RemoteImage(serverName, sum);

    // Set the display title and window layout.
    setTitle(getClass().getName());
    setLayout(new GridLayout(2, 2));

    // Local rendering.
    add(new ScrollingImagePanel(sum,
                                sum.getWidth(),
                                sum.getHeight()));

    // RenderedOp remote rendering.
    add(new ScrollingImagePanel(remoteImage,
                                remoteImage.getWidth(),
                                remoteImage.getHeight()));

    // RenderedImage remote rendering
    PlanarImage sumImage = sum.getRendering();
    remoteImage = new RemoteImage(serverName, sumImage);
    add(new ScrollingImagePanel(remoteImage,
                                remoteImage.getWidth(),
                                remoteImage.getHeight()));

    // RenderableOp remote rendering.
    pb = new ParameterBlock();
    pb.addSource(dithered);
    RenderableOp absImage = JAI.createRenderable("absolute", pb);
    pb = new ParameterBlock();
    pb.addSource(absImage).add(ColorCube.BYTE_496);
    RenderableOp lutImage = JAI.createRenderable("lookup", pb);
    AffineTransform tf =
          AffineTransform.getScaleInstance(384/dithered.getWidth(),
                                           256/dithered.getHeight());
    Rectangle aoi = new Rectangle(128, 128, 384, 256);
    RenderContext rc = new RenderContext(tf, aoi, rh);
    remoteImage = new RemoteImage(serverName, lutImage, rc);
    add(new ScrollingImagePanel(remoteImage,
                                remoteImage.getWidth(),
                                remoteImage.getHeight()));

    // Finally display everything
            pack();
            show();
        }
    }
```

### 12.4.2 RemoteImaging Example Across Two Nodes

Listing 12-2 shows an example of a RemoteImaging chain spread across two remote nodes, and displays the results locally.

**Listing 12-2 RemoteImaging Example Program Using Two Nodes (Sheet 1 of 2)**

```
import java.awt.image.*;
import java.awt.image.renderable.ParameterBlock;
import javax.media.jai.*;
import javax.media.jai.widget.*;

/**
 * This test creates an imaging chain spread across two remote
 * nodes and displays the result locally.
 */

public class MultiNodeTest extends WindowContainer {
    public static void main(String[] args) {
        if(args.length != 3) {
        throw new RuntimeException("Usage: java MultiNodeTest "+
                                   "file node1 node2");
        }

        new MultiNodeTest(args[0], args[1], args[2]);
    }
public MultiNodeTest(String fileName, String node1, String
                     node2) {

// Create a chain on node 1.
System.out.println("Creating dst1 = log(invert(fileload("+
                         fileName+"))) on "+node1);
        RenderedOp src = JAI.create("fileload", fileName);
        RenderedOp op1 = JAI.create("invert", src);
        RenderedOp op2 = JAI.create("log", op1);
        RemoteImage rmt1 = new RemoteImage(node1, op2);

// Create a chain on node 2.
System.out.println("Creating dst2 = not(exp(dst1)) on "+node2);
        RenderedOp op3 = JAI.create("exp", rmt1);
        RenderedOp op4 = JAI.create("not", op3);
        RemoteImage rmt2 = new RemoteImage(node2, op4);
```

**Listing 12-2 RemoteImaging Example Program Using Two Nodes (Sheet 2 of 2)**

```
// Display the result of node 2.
System.out.println("Displaying results");
setTitle(getClass().getName()+" "+fileName);
add(new ScrollingImagePanel(rmt2, rmt2.getWidth(),
                            rmt2.getHeight()));
        pack();
        show();
    }
}
```

**API:** `javax.media.jai.RemoteImage`

- `int getWidth()`

  returns the width of the `RemoteImage`.

- `int getHeight()`

  returns the height of the `RemoteImage`.

- `Raster getData()`

  returns the entire image as one large tile.

- `Raster getData(Rectangle rect)`

  returns an arbitrary rectangular region of the `RemoteImage`.

| | | |
|---|---|---|
| *Parameters*: | `rect` | The region of the `RemoteImage` to be returned. |

- `WritableRaster copyData(WritableRaster raster)`

  returns an arbitrary rectangular region of the `RemoteImage` in a user-supplied `WritableRaster`. The rectangular region is the entire image if the argument is null or the intersection of the argument bounds with the image bounds if the region is non-null. If the argument is non-null but has bounds that have an empty intersection with the image bounds, the return value will be null. The return value may also be null if the argument is non-null but is incompatible with the `Raster` returned from the remote image.

| | | |
|---|---|---|
| *Parameters*: | `raster` | A `WritableRaster` to hold the returned portion of the image. |

  If the `raster` argument is null, the entire image will be copied into a newly-created WritableRaster with a SampleModel that is compatible with that of the image.

- `Raster getTile(int x, int y)`

  returns a tile (*x*, *y*). Note that *x* and *y* are indices into the tile array, not pixel locations. Unlike in the true `RenderedImage` interface, the `Raster` that is returned should be considered a copy.

| | | |
|---|---|---|
| *Parameters*: | `x` | The *x* index of the requested tile in the tile array |
| | y | The *y* index of the requested tile in the tile array |

## 12.5 Running Remote Imaging

To run remote imaging in JAI, you have to do the following:

1. Create a security policy file
2. Start the RMI registry
3. Start the remote image server
4. Run the local application

These four steps are explained in more detail in the following sections.

### 12.5.1 Step 1: Create a Security Policy File

The default RMI security policy implementation is specified within one or more policy configuration files. These configuration files specify what permissions are allowed for code from various sources. There is a default system-wide policy file and a single user policy file. For more information on policy files and permissions, see:

```
http://java.sun.com/products/jdk/1.2/docs/guide/security/
PolicyFiles.html
http://java.sun.com/products/jdk/1.2/docs/guide/security/
permissions.html
```

The policy file is located in the base directory where Java Advanced Imaging is installed. If `$JAI` is the base directory where Java Advanced Imaging is installed, use any simple text editor to create a text file named `$JAI/policy` containing the following:

```
grant {
// Allow everything for now
   permission java.security.AllPermission;
};
```

Note that this policy file is for testing purposes only.

### 12.5.2 Step 2: Start the RMI Registry

The RMI registry is a simple server-side name server that allows remote clients to get a reference to a remote object. Typically, the registry is used only to locate the first remote object an application needs to talk to. Then that object in turn provides application-specific support for finding other objects.

---

**Note:** Before starting the rmiregistry, make sure that the shell or window in which you will run the registry either has no `CLASSPATH` set or has a `CLASSPATH` that does not include the path to any classes you want downloaded to your client, including the stubs for your remote object implementation classes.

---

To start the registry on the server, log in to the remote system where the image server will be running and execute the `rmiregistry` command.

For example, in the **Solaris** operating environment using a Bourne-compatible shell (e.g., /bin/sh):

```
$ unset CLASSPATH
$ rmiregistry &
```

Note that the `CLASSPATH` environment variable is deliberately not set.

For example, on **Windows 95** or **Windows NT**:

```
start rmiregistry
```

If the `start` command is not available, use `javaw`.

### 12.5.3 Step 3: Start the Remote Image Server

While still logged in to the remote server system, set the `CLASSPATH` and `LD_LIBRARY_PATH` environment variables as required for JAI (see the `INSTALL` file) and start the remote imaging server. For example:

```
$ CLASSPATH=$JAI/lib/jai.jar:\
            $JAI/lib/mlibwrapper_jai.jar
$ export CLASSPATH
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JAI/lib
$ export LD_LIBRARY_PATH
$ java \
-Djava.rmi.server.codebase=\
     file:$JAI/lib/jai.jar \
-Djava.rmi.server.useCodebaseOnly=false \
-Djava.security.policy=file:$JAI/policy \
      com.sun.media.jai.rmi.RMIImageImpl
```

For example, when the above steps are executed on a machine with IP address 123.456.78.90 the following is printed:

```
Server: using host 123.456.78.90 port 1099
Registering image server as
  "rmi://123.456.78.90:1099/RemoteImageServer".
Server: Bound RemoteImageServer into
  the registry.
```

### 12.5.4 Step 4: Run the Local Application

After completing steps 1 through 3, you are ready to run the local application. When running the local application, make sure that the `serverName` parameter of any RemoteImage constructors corresponds to the machine on which the remote image server is running. For example, if the machine with IP address 123.456.78.90 above is named `myserver`, the `serverName` parameter of any `RemoteImage()` constructors should be `"myserver"`.

## 12.6 Internet Imaging Protocol (IIP)

There are two JAI operations that support Internet Imaging Protocol (IIP) operations. Two separate operations provide client-side support of the Internet Imaging Protocol. These operations, `IIP` and `IIPResolution`, request an image from an IIP server then create either a RenderedImage or a RenderableImage.

### 12.6.1 IIP Operation

The `IIP` operation provides client-side support of the Internet Imaging Protocol (IIP) in both the rendered and renderable modes. It creates a `RenderedImage` or a `RenderableImage` based on the data received from the IIP server, and optionally applies a sequence of operations to the created image.

The operations that may be applied and the order in which they are applied are defined in section 2.2.1.1 of the *Internet Imaging Protocol Specification* version 1.0.5. Some or all of the requested operations may be executed on the IIP server if it is determined that the server supports such operations. Any of the requested operations not supported by the server will be executed on the host on which the operation chain is rendered.

The processing sequence for the supplied operations is as follows:

- Filtering (blur or sharpen)
- Tone and color correction ("color twist")
- Contrast adjustment
- Selection of source rectangle of interest
- Spatial orientation (rendering-independent affine transformation)
- Selection of destination rectangle of interest
- Rendering transformation (renderable mode only)
- Transposition (rotation and/or mirroring).

As indicated, the rendering transformation is performed only in renderable mode processing. This transformation is derived from the AffineTransform supplied in the RenderContext when rendering actually occurs. Rendered mode processing creates a RenderedImage which is the default rendering of the RenderableImage created in renderable mode processing.

The `IIP` operation takes 14 parameters.

| **Parameter** | **Type** | **Description** |
|---|---|---|
| `URL` | `String` | The URL of the IIP image |
| `subImages` | `int[]` | The sub-images to be used by the server for images at each resolution level |
| `filter` | `Float` | The filtering value |
| `colorTwist` | `float[]` | The color twist matrix |
| `contrast` | `Float` | The contrast value |
| `sourceROI` | `Rectangle2D.Float` | The source rectangle of interest in rendering-independent coordinates |
| `transform` | `AffineTransform` | The rendering-independent spatial orientation transform |
| `aspectRatio` | `Float` | The aspect ratio of the destination image |
| `destROI` | `Rectangle2D.Float` | The destination rectangle of interest in rendering-independent coordinates |
| `rotation` | `Integer` | The counterclockwise rotation angle to be applied to the destination |
| `mirrorAxis` | `String` | The mirror axis |
| `ICCProfile` | `color.ICC_Profile` | The ICC profile used to represent the color space of the source image |

| Parameter | Type | Description |
|---|---|---|
| `JPEGQuality` | `Integer` | The JPEG quality factor |
| `JPEGTable` | `Integer` | The JPEG compression group index number |

The `URL` parameter specifies the URL of the IIP image as a `java.lang.String`. It must represent a valid URL and include any required FIF or SDS commands. It cannot be null.

The `subImages` parameter optionally indicates the sub-images to be used by the server to get the images at each resolution level. The values in this `int` array cannot be negative. If this parameter is not specified, or if the array is too short (length is 0), or if a negative value is specified, this operation will use the zeroth sub-image of the resolution level actually processed.

The `filter` parameter specifies a blur or sharpen operation; a positive value indicates sharpen and a negative value blur. A unit step should produce a perceptible change in the image. The default value is 0 which signifies that no filtering will occur.

The `colorTwist` parameter represents a $4 \times 4$ matrix stored in row-major order and should have an array length of at least 16. If an array of length greater than 16 is specified, all elements from index 16 and beyond are ignored. Elements 12, 13, and 14 must be 0. This matrix will be applied to the (possibly padded) data in an intermediate normalized PhotoYCC color space with a premultiplied alpha channel. This operation will force an alpha channel to be added to the image if the last column of the last row of the color twist matrix is not 1.0F. Also, if the image originally has a grayscale color space it will be cast up to RGB if casting the data back to grayscale after applying the color twist matrix would result in any loss of data. The default value is null.

The `contrast` parameter specifies a contrast enhancement operation with increasing contrast for larger value. It must be greater than or equal to 1.0F. A value of 1.0F indicates no contrast adjustment. The default value is 1.0F.

The `sourceROI` parameter specifies the rectangle of interest in the source image in rendering-independent coordinates. The intersection of this rectangle with the rendering-independent bounds of the source image must equal itself. The rendering-independent bounds of the source image are defined to be (0.0F, 0.0F, r, 1.0F) where r is the aspect ratio (width/height) of the source image. Note that the source image will not in fact be cropped to these limits but values outside of this rectangle will be suppressed.

The `transform` parameter represents an affine backward mapping to be applied in rendering-independent coordinates. Note that the direction of transformation is opposite to that of the AffineTransform supplied in the RenderContext which is a forward mapping. The default value of this transform is the identity mapping. The supplied AffineTransform must be invertible.

The `aspectRatio` parameter specifies the rendering-independent width of the destination image and must be positive. The rendering-independent bounds of the destination image are (0.0F, 0.0F, aspectRatio, 1.0F). If this parameter is not provided, the destination aspect ratio defaults to that of the source.

The `destROI` parameter specifies the rectangle of interest in the destination image in rendering-independent coordinates. This rectangle must have a non-empty intersection with the rendering-independent bounds of the destination image but is not constrained to the destination image bounds.

The `rotation` parameter specifies a counter-clockwise rotation angle of the destination image. The rotation angle is limited to 0, 90, 180, or 270 degrees. By default, the destination image is not rotated.

The `mirrorAxis` parameter may be null, in which case no flipping is applied, or a String of `x`, `X`, `y`, or `Y`.

The `ICCProfile` parameter may only be used with client-side processing or with server-side processing if the connection protocol supports the ability to transfer a profile.

The `JPEGQuality` and `JPEGTable` parameters are only used with server-side processing. If provided, `JPEGQuality` must be in the range [0,100] and `JPEGTable` in [1,255].

There is no source image associated with this operation.

Listing 12-3 shows a code sample for an `IIP` operation.

**Listing 12-3 IIP Operation Example**

```
public static final String SERVER = "http://istserver:8087/";
public static final String DEFAULT_IMAGE = "cat.fpx";
public static final int DEFAULT_HEIGHT = 512;

public static void main(String[] args) {
    String imagePath = DEFAULT_IMAGE;
```

**Listing 12-3 IIP Operation Example (Continued)**

```
        for(int i = 0; i < args.length; i++) {
            if(args[i].equalsIgnoreCase("-image")) {
                imagePath = args[++i];
                if(!(imagePath.toLowerCase().endsWith(".fpx"))) {
                     imagePath += ".fpx";
                }
            }
        }

        String url = SERVER + "FIF=" + imagePath;

        new IIPTest(url);
    }

    // Define the parameter block.
    ParameterBlock pb = (new ParameterBlock()).add(url);

    // Default sub-image array
    pb.set(-10.0F, 2); // filter
    float[] colorTwist = new float[]
        {1.0F, 0.0F, 0.0F, 0.0F,
         0.0F, 0.0F, 1.0F, 0.0F,
         0.0F, 1.0F, 0.0F, 0.0F,
         0.0F, 0.0F, 0.0F, 1.0F};
    pb.set(colorTwist, 3); //color-twist
    pb.set(2.0F, 4); // contrast
    pb.set(new Rectangle2D.Float(0.10F, 0.10F,
                             0.80F*aspectRatioSource, 0.80F),
        5); // srcROI

    AffineTransform afn = AffineTransform.getShearInstance(0.2,
                        0.1);
    pb.set(afn, 6); // transform
    Rectangle2D destBounds = null;

    try {
         Rectangle2D sourceRect =
               new Rectangle2D.Float(0.0F, 0.0F, aspectRatioSource,
                                        1.0F);
         Shape shape =
            afn.createInverse().createTransformedShape(sourceRect);
         destBounds = shape.getBounds2D();
    } catch(Exception e) {
    }
```

**Listing 12-3 IIP Operation Example (Continued)**

```
    float aspectRatio = (float)destBounds.getHeight();
    pb.set(aspectRatio, 7); // destination aspect ratio
    pb.set(new Rectangle2D.Float(0.0F, 0.0F,
                           0.75F*aspectRatio, 0.75F), 8); // dstROI
    pb.set(90, 9); // rotation angle
    pb.set("x", 10); // mirror axis

    // Default ICC profile
    // Default JPEG quality
    // Default JPEG table index

    int height = DEFAULT_HEIGHT;
    AffineTransform at =
        AffineTransform.getScaleInstance(height*aspectRatioSource,
                           height);
    RenderContext rc = new RenderContext(at);

    // Create a RenderableImage.
    RenderableImage renderable = JAI.createRenderable("iip", pb);
```

## 12.6.2 IIPResolution Operation

The `IIPResolution` operation provides client-side support of the Internet Imaging Protocol (IIP) in the rendered mode. It is resolution-specific. It requests from the IIP server an image at a particular resolution level, and creates a `RenderedImage` based on the data received from the server. Once the `RenderedImage` is created, the resolution level cannot be changed.

The layout of the created RenderedImage is set as follows:

- minX, minY, tileGridXOffset, and tileGridYOffset are set to 0
- width and height are determined based on the specified resolution level
- tileWidth and tileHeight are set to 64
- sampleModel is of the type `PixelInterleavedSampleModel` with byte data type and the appropriate number of bands
- colorModel is of the type `java.awt.image.ComponentColorModel`, with the ColorSpace set to sRGB, PhotoYCC, or Grayscale, depending on the color space of the remote image; if an alpha channel is present, it will be premultiplied

The `IIPResolution` operation takes three parameters.

| Parameter | Type | Description |
|---|---|---|
| `URL` | `String` | The URL of the IIP image |
| `resolution` | `Integer` | The resolution level to request |
| `subImage` | `Integer` | The sub-image to be used by the server |

The `URL` parameter specifies the URL of the IIP image as a `java.lang.String`. It must represent a valid URL, and include any required FIF or SDS commands. It cannot be null.

The `resolution` parameter specifies the resolution level of the requested IIP image from the server. The lowest resolution level is 0, with larger integers representing higher resolution levels. If the requested resolution level does not exist, the nearest resolution level is used. If this parameter is not specified, it is set to the default value `IIPResolutionDescriptor.MAX_RESOLUTION`, which indicates the highest resolution level.

The `subImage` parameter indicates the sub-image to be used by the server to get the image at the specified resolution level. This parameter cannot be negative. If this parameter is not specified, it is set to the default value 0.

There is no source image associated with this operation.

If available from the IIP server certain properties may be set on the RenderedImage. The names of properties and the class types of their associated values are listed in the following table. See the IIP specification for information on each of these properties.

| Property | Type |
|---|---|
| affine-transform | `java.awt.geom.AffineTransform` |
| app-name | `java.lang.String` |
| aspect-ratio | `java.lang.Float` |
| author | `java.lang.String` |
| colorspace | `int[]` |
| color-twist | `float[16]` |
| comment | `java.lang.String` |
| contrast-adjust | `java.lang.Float` |
| copyright | `java.lang.String` |
| create-dtm | `java.lang.String` |

| Property | Type |
|---|---|
| edit-time | `java.lang.String` |
| filtering-value | `java.lang.Float` |
| iip | `java.lang.String` |
| iip-server | `java.lang.String` |
| keywords | `java.lang.String` |
| last-author | `java.lang.String` |
| last-printed | `java.lang.String` |
| last-save-dtm | `java.lang.String` |
| max-size | `int[2]` |
| resolution-number | `java.lang.Integer` |
| rev-number | `java.lang.String` |
| roi-iip | `java.awt.geom.Rectangle2D.Float` |
| subject | `java.lang.String` |
| title | `java.lang.String` |

Listing 12-4 shows a code sample for an `IIPResolution` operation.

**Listing 12-4 IIPResolution Operation Example**

```
public static final String SERVER = "http://istserver:8087/";
public static final String DEFAULT_IMAGE = "cat.fpx";
public static final int DEFAULT_RESOLUTION = 3;

public static void main(String[] args) {
    String imagePath = DEFAULT_IMAGE;
    int resolution = DEFAULT_RESOLUTION;

    for(int i = 0; i < args.length; i++) {
        if(args[i].equalsIgnoreCase("-image")) {
            imagePath = args[++i];
            if(!(imagePath.toLowerCase().endsWith(".fpx"))) {
                imagePath += ".fpx";
            }
        } else if(args[i].equalsIgnoreCase("-res")) {
            resolution = Integer.valueOf(args[++i]).intValue();
        }
    }

    String url = SERVER + "FIF=" + imagePath;

    new IIPResolutionTest(url, resolution);
}
```

**Listing 12-4 IIPResolution Operation Example (Continued)**

```
   ParameterBlock pb = new ParameterBlock();
   pb.add(url).add(resolution);
   PlanarImage pi = JAI.create("iipresolution", pb);
```

CHAPTER 13

# Writing Image Files

THIS chapter describes JAI's codec system for writing image data files.

## 13.1 Introduction

The JAI codec system supports a variety of image formats for writing an image to a file or to an `OutputStream` for further manipulation. For writing an image to a file, the `FileStore` operation (see Section 13.2, "Writing to a File") writes an image to a specified file in the specified format. For encoding an image to an `OutputStream`, the `Encode` operation (see Section 13.3, "Writing to an Output Stream") writes an image to a given `OutputStream` in a specified format using the encoding parameters supplied via the `ImageEncodeParam` operation parameter.

## 13.2 Writing to a File

The `FileStore` operation writes an image to a given file in a specified format using the specified encoding parameters. This operation is much simpler than the encoders described in the remainder of this chapter.

The `FileStore` operation takes one rendered source image and three parameters:

| Parameter | Type | Description |
|---|---|---|
| `filename` | `String` | The path of the file to write to. |
| `format` | `String` | The format of the file. |
| `param` | `ImageEncodeParam` | The encoding parameters. |

The `filename` parameter must be supplied or the operation will not be performed. Also, the specified file path must be writable.

The `format` parameter defaults to `tiff` if no value is provided. Table 13-1 lists the recognized JAI file formats.

**Table 13-1 JAI Writable File Formats**

| File Format | Description |
|---|---|
| BMP | Microsoft Windows bitmap image file |
| JPEG | A file format developed by the Joint Photographic Experts Group |
| PNG | Portable Network Graphics |
| PNM | Portable aNy Map file format. Includes PBM, PGM, and PPM |
| TIFF | Tag Image File Format |

The `param` parameter must either be null or an instance of an `ImageEncodeParam` subclass appropriate to the format.

Listing 13-1 shows a code sample demonstrating the use of both the `Encode` and `FileStore` operations.

## 13.3 Writing to an Output Stream

The `Encode` operation writes an image to a given `OutputStream` in a specified format using the encoding parameters supplied via the `ImageEncodeParam` operation parameter.

The `Encode` operation takes one rendered source image and three parameters:

| Parameter | Type | Description |
|---|---|---|
| `stream` | `OutputStream` | The `OutputStream` to write to. |
| `format` | `String` | The format of the created file. |
| `param` | `ImageEncodeParam` | The encoding parameters. |

The `param` parameter must either be null or an instance of an `ImageEncodeParam` subclass appropriate to the specified image format. The image encode parameter depends on the type of image file to be encoded. This parameter contains all of the information about the file type that the encoder needs to create the file. For example, the BMP format requires two parameter values, as described in the `BMPEncodeParam` class:

- Version number – One of three values: `VERSION_2`, `VERSION_3`, or `VERSION_4`.
- Data layout – One of two values: `TOP_DOWN` or `BOTTOM_UP`.

These parameters are described in detail in Section 13.4, "Writing BMP Image Files."

Listing 13-1 shows a code sample demonstrating the use of both the `Encode` and `FileStore` operations.

**Listing 13-1 Writing an OutputStream and a File**

```
// Define the source and destination file names.
String inputFile = /images/FarmHouse.tif
String outputFile = /images/FarmHouse.bmp

// Load the input image.
RenderedOp src = JAI.create("fileload", inputFile);

// Encode the file as a BMP image.
FileOutputStream stream =
    new FileOutputStream(outputFile);
JAI.create("encode", src, stream, BMP, null);

// Store the image in the BMP format.
JAI.create("filestore", src, outputFile, BMP, null);
```

## 13.4 Writing BMP Image Files

As described above, the encoding of BMP images requires the specification of two parameters: version and data layout. By default, these values are:

- Version – VERSION_3
- Data layout – pixels are stored in bottom-up order

The JAI BMP encoder does not support compression of BMP image files.

### 13.4.1 BMP Version

JAI currently reads and writes Version2, Version3, and some of the Version 4 images. The BMP version number is read and specified with `getVersion` and `setVersion` methods in the `BMPEncodeParam` class. The BMP version parameters are as follows:

| Parameter | Description |
|---|---|
| `VERSION_2` | Specifies BMP Version 2 |
| `VERSION_3` | Specifies BMP Version 3 |
| `VERSION_4` | Specifies BMP Version 4 |

If not specifically set, `VERSION_3` is the default version.

**API:** `com.sun.media.jai.codec.BMPEncodeParam`

- `void setVersion(int versionNumber)`
  sets the BMP version to be used.

- `int getVersion()`
  returns the BMP version to be used.

### 13.4.2 BMP Data Layout

The scan lines in the BMP bitmap are stored from the bottom up. This means that the first byte in the array represents the pixels in the lower-left corner of the bitmap, and the last byte represents the pixels in the upper-right corner.

The in-memory layout of the image data to be encoded is specified with `getDataLayout` and `setDataLayout` methods in the `BMPEncodeParam` class.

**API:** `com.sun.media.jai.codec.BMPEncodeParam`

- `void setTopDown(boolean topDown)`
  sets the data layout to be top down.

### 13.4.3 Example Code

Listing 13-2 shows a code sample for encoding a BMP image.

**Listing 13-2 Encoding a BMP Image**

```
OutputStream os = new FileOutputStream(fileToWriteTo);
BMPEncodeParam param = new BMPEncodeParam();
ImageEncoder enc = ImageCodec.createImageEncoder("BMP", os,
                                                 param);
enc.encode(op);
os.close();
```

## 13.5 Writing JPEG Image Files

The JPEG standard was developed by a working group, known as the Joint Photographic Experts Group (JPEG). The JPEG image data compression standard handles grayscale and color images of varying resolution and size.

JPEG compression identifies and discards "extra" data that is beyond what the human eye can see. Since it discards data, the JPEG compression algorithm is considered "lossy." This means that once an image has been compressed and then decompressed, it will not be identical to the original image. In most cases, the difference between the original and compressed version of the image is indistinguishable.

An advantage of JPEG compression is the ability to select the quality when compressing the image. The lower the quality, the smaller the image file size, but the more different it will appear than the original.

Table 13-2 lists the JPEG encode parameters that may be set and the default values. The remaining sections describe these settings and how to change them.

**Table 13-2 JPEG Encode Parameters**

| **Parameter** | **Description** | **Default Value** |
|---|---|---|
| `writeJFIFHeader` | Controls whether the encoder writes a JFIF header using the APP0 marker. See Section 13.5.1, "JFIF Header." | True |
| `qTabSlot[0],[1],[2]` | Quantization tables. See Section 13.5.3, "Quantization Table." | 0 for Y channel, 1 for Cb and Cr channels |
| `qTab[0],[1],[2]` | Quantization table contents. See Section 13.5.3, "Quantization Table." | Null for all three channels |
| `qTabSet[0],[1],[2]` | Quantization table usage. See Section 13.5.3, "Quantization Table." | False for all three channels |
| `hSamp[0],[1],[2]` | Horizontal subsampling. See Section 13.5.4, "Horizontal and Vertical Subsampling." | 1 for Y channel, 2 for Cb and Cr channels |
| `vSamp[0],[1],[2]` | Vertical subsampling. See Section 13.5.4, "Horizontal and Vertical Subsampling." | 1 for Y channel, 2 for Cb and Cr channels |
| `qual` | Quality setting. See Section 13.5.5, "Compression Quality." | 0.75F |
| `rstInterval` | Restart interval. Section 13.5.6, "Restart Interval." | 0 |
| `writeImageOnly` | Controls whether encoder writes only the compressed image data. See Section 13.5.7, "Writing an Abbreviated JPEG Stream." | False |

## 13.5.1 JFIF Header

The JPEG File Interchange Format (JFIF) is a minimal file format that enables JPEG bitstreams to be exchanged between a wide variety of platforms and applications. This minimal format does not include any of the advanced features found in the TIFF JPEG specification or any application-specific file format. The sole purpose of this simplified format is to allow the exchange of JPEG compressed images.

The JFIF features are:

- Uses the JPEG baseline image compression algorithm
- Uses JPEG interchange format compressed image representation
- Compatible with most platforms (PC, Mac, or Unix)
- Standard color space: one or three components. For three components, $YC_bC_r$ (CCIR 601-256 levels)

An APP0 marker is used to identify a JFIF file. The marker provides information that is missing from the JPEG stream, such as version number, *x* and *y* pixel density (dots per inch or dots per cm.), pixel aspect ratio (derived from *x* and *y* pixel density), and thumbnail. The `setWriteJFIFHeader` method controls whether the encoder writes a JFIF header using the APP0 marker.

**API:** `com.sun.media.jai.codec.JPEGEncodeParam`

- `void setWriteJFIFHeader(boolean writeJFIF)`

  controls whether the encoder writes a JFIF header using the APP0 marker. By default an APP0 marker is written to create a JFIF file.

  *Parameter*: `writeJFIF` If true, writes a JFIF header.

## 13.5.2 JPEG DCT Compression Parameters

JAI uses the JPEG baseline DCT coding process, shown in Figure 13-1.

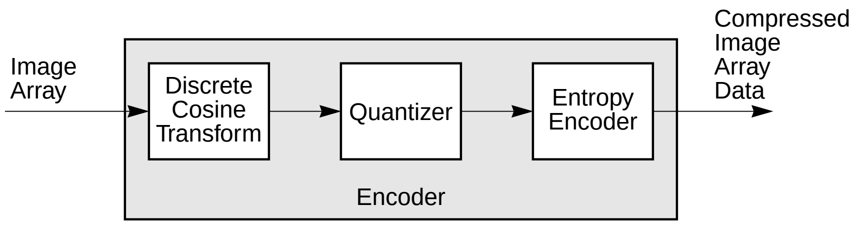


**Figure 13-1 JPEG Baseline DCT Coding**

For encoding, the image array is divided into 8 × 8 pixel blocks and a discrete cosine transform (DCT) is taken of each block, resulting in an 8 × 8array of transform coefficients. The DCT is a mathematical operation that takes the block of image samples as its input and converts the information from the spatial domain to the frequency domain. The $8 \times 8$ matrix input to the DCT represents brightness levels at specific $x$, $y$ coordinates. The resulting $8 \times 8$ matrix values represent relative amounts of 64 spatial frequencies that make up the spectrum of the input data.

The next stage in the encoder quantizes the transform coefficients by dividing each DCT coefficient by a value from a quantization table. The quantization operation discards the smaller-valued frequency components, leaving only the larger-valued components.

After an image block has been quantized, it enters the entropy encoder, which creates the actual JPEG bitstream. The entropy encoder assigns a binary Huffman code to coefficient values. The length of each code is chosen to be inversely proportional to the expected probability of occurrence of a coefficient amplitude – frequently-occurring coefficient values get short code words, seldom-occurring coefficient values get long code words. The entropy encoder uses two tables, one for the AC frequency components and one for the DC frequency components.

The JPEG decoding process is essentially the inverse of the encoding process. The compressed image array data stream passes through the entropy encoder, which recreates the quantized coefficient values. Then, the quantized coefficients are reconstructed by multiplication with the quantizer table values. Finally, an inverse DCT is performed and the reconstructed image array is produced.

The following are the parameters that may be specified for JPEG DCT compression.

### 13.5.3 Quantization Table

The `setQTable` and `getQTable` methods are used to specify and retrieve the quantization table that will be used in encoding a particular band of the image. There are, by default, two quantizer tables:

| Table | Band |
|---|---|
| 0 | Band 0 |
| 1 | All other bands |

The parameter `tableNum` is usually a value between 0 and 3. This value indicates which of four quantization tables you are specifying. Table 0 is designed to be used with the luminance band of eight-bit YCC images. Table 1 is designed to be used with the chrominance bands of eight-bit YCC images. The two tables can also be set individually using the `setLumaQTable` (table 0) and `setChromaQTable` (table 1) methods. Tables 2 and 3 are not normally used.

**API:** `com.sun.media.jai.codec.JPEGEncodeParam`

- `void setQTable(int component, int tableNum, int[] qTable)`

  sets a quantization table to be used for a component. This method allows up to four independent tables to be specified. This disables any quality setting.

| | | |
|---|---|---|
| *Parameters*: | `component` | The band to which this table applies. |
| | `tableNum` | The table number that this table is assigned to (0 to 3). |
| | `qTable` | Quantization table values in “zig-zag” order. |

- `int[] getQTable(int component)`

  returns the contents of the quantization table used for a component. If this method is called before the quantization table is set, an error is thrown.

- `void setLumaQTable(int[] qTable)`

  sets the quantization table to be used for luminance data. This is a convenience method that explicitly sets the contents of quantization table 0. The length of the table must be 64. This disables any quality setting.

- `void setChromaQTable(int[] qTable)`

  sets the quantization table to be used for luminance data. This is a convenience method that explicitly sets the contents of quantization table 0. The length of the table must be 64. This method assumes that all chroma components will use the same table. This disables any quality setting.

- `int getQTableSlot(int component)`

  returns the quantization table slot used for a component. If this method is called before the quantization table data is set, an error is thrown.

### 13.5.4 Horizontal and Vertical Subsampling

JPEG allows the image components to be subsampled to reduce their resolution prior to encoding. This is typically done with YCC images, where the two chroma components can be subsampled, usually by a factor of two in both axes.

This is possible due to the human visual system's low sensitivity to color images relative to luminance (Y) errors By default, the sampling factors for YCC input images are set to {1, 2, 2} for both horizontal and vertical axes.

**API:** `com.sun.media.jai.codec.JPEGEncodeParam`

- `void setHorizontalSubsampling(int component, int subsample)`

  sets the horizontal subsampling to be applied to an image band. Defaults to 1 for grayscale and (1,2,2) for RGB.

  | | | |
  |---|---|---|
  | *Parameter*: | `component` | The band for which to set horizontal subsampling. |
  | | `subsample` | The horizontal subsampling factor. |

- `void setVerticalSubsampling(int component, int subsample)`

  sets the vertical subsampling to be applied to an image band. Defaults to 1 for grayscale and (1,2,2) for RGB.

- `int getHorizontalSubsampling(int component)`

  returns the horizontal subsampling factor for a band.

- `int getVerticalSubsampling(int component)`

  returns the vertical subsampling factor for a band.

### 13.5.5 Compression Quality

Compression quality specifies a factor that relates to the desired tradeoff between image quality and the image data compression ratio. The quality value is a `float` between 0.0 and 1.0. A setting of 1.0 produces the highest quality image at a lower compression ratio. A setting of 0.0 produces the highest compression ratio, with a sacrifice to image quality. The quality value is typically set to 0.75.

The compression quality value controls image quality and compression ratio by determining a scale factor the encoder will use in creating scaled versions of the quantization tables. Some guidelines:

| Quality Value | Meaning |
|---|---|
| 1.0 | Highest quality, no compression |
| 0.75 | High quality, good compression ratio |

| **Quality Value** | **Meaning** |
|---|---|
| 0.5 | Medium quality, medium compression ratio |
| 0.25 | Low quality, high compression ratio |

**Note:** The values stored in the quantization table also affect image quality and compression ratio. See also Section 13.5.3, “Quantization Table.”

**API:** `com.sun.media.jai.codec.JPEGEncodeParam`

- `void setQuality(float quality)`

  sets the compression quality factor. Creates new quantization tables that replace the currently-installed quantization tables.

  | | | |
  |---|---|---|
  | *Parameter*: | `quality` | The desired quality level; a value of 0.0 to 1.0. The default value is 0.75. |

- `float getQuality()`

  returns the quality setting for this encoding. This is a number between 0.0 and 1.0.

- `boolean isQualitySet()`

  tests if the quality parameter has been set in this `JPEGEncodeParam`.

### 13.5.6 Restart Interval

JPEG images use restart markers to define multiple strips or tiles. The restart markers are inserted periodically into the image data to delineate image segments known as *restart intervals*. To limit the effect of bitstream errors to a single restart interval, JAI provides methods to set the restart interval in JPEG Minimum Coded Units (MCUs). The default is zero (no restart interval markers).

**API:** `com.sun.media.jai.codec.JPEGEncodeParam`

- `void setRestartInterval(int restartInterval)`

  sets the restart interval in Minimum Coded Units (MCUs).

  | | | |
  |---|---|---|
  | *Parameter*: | `restartInterval` | Number of MCUs between restart markers. |

- `int getRestartInterval()`

  returns the restart interval.

## 13.5.7 Writing an Abbreviated JPEG Stream

Normally, both the JPEG table data and compressed (or uncompressed) image data is written to the output stream. However, it is possible to write just the table data or just the image data. The `setWriteTablesOnly` method instructs the encoder to write only the table data to the output stream. The `setWriteImageOnly` method instructs the encoder to write only the compressed image data to the output stream.

**API:** `com.sun.media.jai.codec.JPEGEncodeParam`

- `void setWriteTablesOnly(boolean tablesOnly)`

  instructs the encoder to write only the table data to the output stream.

| | | |
|---|---|---|
| *Parameter*: | `tablesOnly` | If true, only the tables will be written. |

- `void setWriteImageOnly(boolean imageOnly)`

  instructs the encoder to write only the image data to the output stream.

| | | |
|---|---|---|
| *Parameter*: | `imageOnly` | If true, only the compressed image will be written. |

## 13.5.8 Example Code

Listing 13-3 shows a code sample for encoding a JPEG image.

**Listing 13-3 Encoding a JPEG Image (Sheet 1 of 5)**

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.awt.image.renderable.*;
import java.io.*;
import javax.media.jai.*;
import javax.media.jai.widget.*;
import com.sun.media.jai.codec.*;

public class JPEGWriterTest extends WindowContainer {

private ImageEncoder encoder = null;
private JPEGEncodeParam encodeParam = null;
```

**Listing 13-3 Encoding a JPEG Image (Sheet 2 of 5)**

```
   // Create some Quantization tables.
       private static int[] qtable1 = {
           1,1,1,1,1,1,1,1,
           1,1,1,1,1,1,1,1,
           1,1,1,1,1,1,1,1,
           1,1,1,1,1,1,1,1,
           1,1,1,1,1,1,1,1,
           1,1,1,1,1,1,1,1,
           1,1,1,1,1,1,1,1,
           1,1,1,1,1,1,1,1
       };

       private static int[] qtable2 = {
           2,2,2,2,2,2,2,2,
           2,2,2,2,2,2,2,2,
           2,2,2,2,2,2,2,2,
           2,2,2,2,2,2,2,2,
           2,2,2,2,2,2,2,2,
           2,2,2,2,2,2,2,2,
           2,2,2,2,2,2,2,2,
           2,2,2,2,2,2,2,2
       };

       private static int[] qtable3 = {
           3,3,3,3,3,3,3,3,
           3,3,3,3,3,3,3,3,
           3,3,3,3,3,3,3,3,
           3,3,3,3,3,3,3,3,
           3,3,3,3,3,3,3,3,
           3,3,3,3,3,3,3,3,
           3,3,3,3,3,3,3,3,
           3,3,3,3,3,3,3,3
       };

       // Really rotten quality Q Table
       private static int[] qtable4 = {
           200,200,200,200,200,200,200,200,
           200,200,200,200,200,200,200,200,
           200,200,200,200,200,200,200,200,
           200,200,200,200,200,200,200,200,
           200,200,200,200,200,200,200,200,
           200,200,200,200,200,200,200,200,
           200,200,200,200,200,200,200,200,
           200,200,200,200,200,200,200,200
       };
```

**Listing 13-3  Encoding a JPEG Image (Sheet 3 of 5)**

```
   public static void main(String args[]) {
       JPEGWriterTest jtest = new JPEGWriterTest(args);
       }

   // Load the source image.
   private PlanarImage loadImage(String imageName) {
   ParameterBlock pb = (new
           ParameterBlock()).add(imageName);
   PlanarImage src = JAI.create("fileload", pb);
           if (src == null) {
         System.out.println("Error in loading image " + imageName);
               System.exit(1);
           }
           return src;
       }

   // Create the image encoder.
   private void encodeImage(PlanarImage img, FileOutputStream out)
       {
   encoder = ImageCodec.createImageEncoder("JPEG", out,
                                           encodeParam);
           try {
               encoder.encode(img);
               out.close();
           } catch (IOException e) {
               System.out.println("IOException at encoding..");
               System.exit(1);
           }
       }

   private FileOutputStream createOutputStream(String outFile) {
           FileOutputStream out = null;
           try {
               out = new FileOutputStream(outFile);
           } catch(IOException e) {
               System.out.println("IOException.");
               System.exit(1);
           }

           return out;
       }

   public JPEGWriterTest(String args[]) {
   // Set parameters from command line arguments.
   String inFile = "images/Parrots.tif";
```

**Listing 13-3 Encoding a JPEG Image (Sheet 4 of 5)**

```
FileOutputStream out1 = createOutputStream(“out1.jpg”);
FileOutputStream out2 = createOutputStream(“out2.jpg”);
FileOutputStream out3 = createOutputStream(“out3.jpg”);

// Create the source op image.
PlanarImage src = loadImage(inFile);

   double[] constants = new double[3];
   constants[0] = 0.0;
   constants[1] = 0.0;
   constants[2] = 0.0;
   ParameterBlock pb = new ParameterBlock();
   pb.addSource(src);
   pb.add(constants);

// Create a new src image with weird tile sizes
ImageLayout layout = new ImageLayout();
layout.setTileWidth(57);
layout.setTileHeight(57);
RenderingHints hints = new RenderingHints(JAI.KEY_IMAGE_LAYOUT,
                                                 layout);
PlanarImage src1 = JAI.create("addconst", pb, hints);

// ----- End src loading ------

// Set the encoding parameters if necessary.
encodeParam = new JPEGEncodeParam();

encodeParam.setQuality(0.1F);

encodeParam.setHorizontalSubsampling(0, 1);
encodeParam.setHorizontalSubsampling(1, 2);
encodeParam.setHorizontalSubsampling(2, 2);

encodeParam.setVerticalSubsampling(0, 1);
encodeParam.setVerticalSubsampling(1, 1);
encodeParam.setVerticalSubsampling(2, 1);

encodeParam.setRestartInterval(64);
//encodeParam.setWriteImageOnly(false);
//encodeParam.setWriteTablesOnly(true);
//encodeParam.setWriteJFIFHeader(true);

// Create the encoder.
encodeImage(src, out1);
PlanarImage dst1 = loadImage(“out1.jpg”);

//   ----- End first encode ---------
```

**Listing 13-3 Encoding a JPEG Image (Sheet 5 of 5)**

```
    encodeParam.setLumaQTable(qtable1);
    encodeParam.setChromaQTable(qtable2);

    encodeImage(src, out2);
    PlanarImage dst2 = loadImage("out2.jpg");

    //   ----- End second encode ---------

    encodeParam = new JPEGEncodeParam();
    encodeImage(loadImage("images/BlackCat.tif"), out3);
    PlanarImage dst3 = loadImage("out3.jpg");

    //   ----- End third encode ---------

    setTitle ("JPEGWriter Test");
    setLayout(new GridLayout(2, 2));
    ScrollingImagePanel panel1 = new ScrollingImagePanel(src, 512,
                                                         400);
    ScrollingImagePanel panel2 = new ScrollingImagePanel(dst1, 512,
                                                         400);
    ScrollingImagePanel panel3 = new ScrollingImagePanel(dst2, 512,
                                                         400);
    ScrollingImagePanel panel4 = new ScrollingImagePanel(dst3, 512,
                                                         400);
       add(panel1);
       add(panel2);
       add(panel3);
       add(panel4);
       pack();
       show();    }
    }
```

## 13.6 Writing PNG Image Files

The Portable Network Graphics (PNG) format is a file standard for compressed lossless bitmapped image files. A PNG file consists of an eight-byte PNG *signature* followed by several *chunks*. The signature identifies the file as a PNG file. The chunks provide additional information about the image. The JAI codec architecture supports PNG 1.1 and provides control over several of the chunks as described in this section.

### 13.6.1 PNG Image Layout

PNG images can be encoded in one of three pixel types, as defined by the subclass of `PNGEncodeParam`, as follows:

| Pixel Type | Description |
|---|---|
| `PNGEncodeParam.Palette` | Also known as *indexed-color*, where each pixel is represented by a single sample that is an index into a supplied color palette. The `com.sun.media.jai.codec.PNGEncodeParam.Palette` class supports the encoding of palette pixel images. |
| `PNGEncodeParam.Gray` | Each pixel is represented by a single sample that is a grayscale level. The `com.sun.media.jai.codec.PNGEncodeParam.Gray` class supports the encoding of grayscale pixel images. |
| `PNGEncodeParam.RGB` | Also known as *truecolor*, where each pixel is represented by three samples: red, green, and blue. The `com.sun.media.jai.codec.PNGEncodeParam.RGB` class supports the encoding of RGB pixel images. |

Optionally, grayscale and RGB pixels can also include an alpha sample (see Section 13.6.6.12, "Transparency (tRNS Chunk)").

A call to the `getDefaultEncodeParam` method returns an instance of:

- `PNGEncodeParam.Palette` for an image with an `IndexColorModel`.
- `PNGEncodeParam.Gray` for an image with only one or two bands.
- `PNGEncodeParam.RGB` for all other images.

This method provides no guarantee that the image can be successfully encoded by the PNG encoder, since the encoder only performs a superficial analysis of the image structure.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `static PNGEncodeParam getDefaultEncodeParam(RenderedImage im)`
  returns an instance of `PNGEncodeParam.Palette`, `PNGEncodeParam.Gray`, or `PNGEncodeParam.RGB` appropriate for encoding the given image.

### 13.6.2 PNG Filtering

The PNG file definition allows the image data to be filtered before it is compressed, which can improve the compressibility of the data. PNG encoding supports five filtering algorithms, including “none,” which indicates no filtering. The filtering algorithms are described below.

**Table 13-3 PNG Filtering Algorithms**

| Parameter | Description |
|---|---|
| `PNG_FILTER_NONE` | No filtering – the scanline is transmitted unaltered. |
| `PNG_FILTER_SUB` | The filter transmits the difference between each byte and the value of the corresponding byte of the prior pixel. |
| `PNG_FILTER_UP` | Similar to the Sub filter, except that the pixel immediately above the current pixel, rather than just to its left, is used as the predictor. |
| `PNG_FILTER_AVERAGE` | The filter uses the average of the two neighboring pixels (left and above) to predict the value of a pixel. |
| `PNG_FILTER_PAETH` | The filter computes a simple linear function of the three neighboring pixels (left, above, upper left), then chooses as predictor the neighboring pixel closest to the computed value. |

The filtering can be different for each row of an image by using the `filterRow` method. The method can be overridden to provide a custom algorithm for choosing the filter type for a given row.

The `filterRow` method is supplied with the current and previous rows of the image. For the first row of the image, or of an interlacing pass, the previous row array will be filled with zeros as required by the PNG specification.

The method is also supplied with five scratch arrays. These arrays may be used within the method for any purpose. At method exit, the array at the index given by the return value of the method should contain the filtered data. The return value will also be used as the filter type.

The default implementation of the method performs a trial encoding with each of the filter types, and computes the sum of absolute values of the differences between the raw bytes of the current row and the predicted values. The index of the filter producing the smallest result is returned.

As an example, to perform only “sub” filtering, this method could be implemented (non-optimally) as follows:

```
for (int i = bytesPerPixel; i < bytesPerRow + bytesPerPixel; i++)
{
     int curr = currRow[i] & 0xff;
     int left = currRow[i - bytesPerPixel] & 0xff;
     scratchRow[PNG_FILTER_SUB][i] = (byte)(curr - left);
}
return PNG_FILTER_SUB;
```

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `int filterRow(byte[] currRow, byte[] prevRow,`
  `byte[][] scratchRows, int bytesPerRow, int bytesPerPixel)`

  returns the type of filtering to be used on a row of an image.

| *Parameters*: | | |
|---|---|---|
| | `currRow` | The current row as an array of `bytes` of length at least `bytesPerRow` + `bytesPerPixel`. The pixel data starts at index `bytesPerPixel`; the initial `bytesPerPixel` bytes are zero. |
| | `prevRow` | The current row as an array of `bytes`. The pixel data starts at index `bytesPerPixel`; the initial `bytesPerPixel` bytes are zero. |
| | `scratchRows` | An array of 5 `byte` arrays of length at least `bytesPerRow` + `bytesPerPixel`, usable to hold temporary results. The filtered row will be returned as one of the entries of this array. The returned filtered data should start at index `bytesPerPixel`; The initial `bytesPerPixel` bytes are not used. |
| | `bytesPerRow` | The number of bytes in the image row. This value will always be greater than 0. |
| | `bytesPerPixel` | The number of bytes representing a single pixel, rounded up to an integer. This is the `bpp` parameter described in the PNG specification. |

## 13.6.3 Bit Depth

The PNG specification identifies the following bit depth restrictions for each of the color types:

**Table 13-4 PNG Bit Depth Restrictions**

| Color Type | Allowed Bit Depths | Description |
|---|---|---|
| 0 | 1, 2, 4, 8, 16 | Grayscale. Each pixel is a grayscale sample. |
| 2 | 8, 16 | Truecolor (RGB) without alpha. Each pixel is an RGB triple. |

**Table 13-4 PNG Bit Depth Restrictions (Continued)**

| Color Type | Allowed Bit Depths | Description |
|---|---|---|
| 3 | 1, 2, 4, 8 | Indexed color (Palette). Each pixel is a palette index. |
| 4 | 8, 16 | Grayscale with alpha. Each pixel is a grayscale sample followed by an alpha sample. |
| 6 | 8, 16 | Truecolor (RGB) with alpha. Each pixel is an RGB triple followed by an alpha sample. |

The bit depth is specified by the `setBithDepth` method in the class type.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.Palette`

- `void setBitDepth(int bitDepth)`

  sets the desired bit depth for a palette image. The bit depth must be 1, 2, 4, or 8.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.Gray`

- `public void setBitDepth(int bitDepth)`

  sets the desired bit depth for a grayscale image. The bit depth must be 1, 2, 4, 8, or 16.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.RGB`

- `void setBitDepth(int bitDepth)`

  sets the desired bit depth for an RGB image. The bit depth must be 8 or 16.

### 13.6.4 Interlaced Data Order

The interlaced data order indicates the transmission order of the image data. Two settings are currently allowed: no interlace and Adam7 interlace. With interlacing turned off, pixels are stored sequentially from left to right, and scanlines sequentially from top to bottom. Adam7 interlacing (named after its author, Adam M. Costello), consists of seven distinct passes over the image; each pass transmits a subset of the pixels in the image.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `void setInterlacing(boolean useInterlacing)`

  turns Adam7 interlacing on or off.

- `boolean getInterlacing()`
  returns `true` if Adam7 interlacing will be used.

### 13.6.5 PLTE Chunk for Palette Images

The PLTE chunk provides the palette information palette or indexed-color images. The PLTE chunk must be supplied for all palette (color type 3) images and is optional for RGB (color type 2 and 6) images.

The PLTE chunk contains from 1 to 256 palette entries, each a three-byte series of the alternating red, green, and blue values, as follows:

- Red: one byte (0 = black, 255 = red)
- Green: one byte (0 = black, 255 = green)
- Blue: one byte (0 = black, 255 = blue)

The number of elements in the palette must be a multiple of 3, between 3 and 768 ($3 \times 256$). The first entry in the palette is referenced by pixel value 0, the second by pixel value 1, and so on.

For RGB (color type 2 and 6) images, the PLTE chunk, if included, provides a suggested set of from 1 to 256 colors to which the RGB image can be quantized in case the viewing system cannot display RGB directly.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `void setPalette(int[] rgb)`
  sets the RGB palette of the image to be encoded.

  *Parameters*: `rgb` An array of `int`s.

- `int[] getPalette()`
  returns the current RGB palette.

- `void unsetPalette()`
  suppresses the PLTE chunk from being output.

- `boolean isPaletteSet()`
  returns true if a PLTE chunk will be output.

## 13.6.6 Ancillary Chunk Specifications

All ancillary PNG chunks are optional but are recommended. Most of the PNG chunks can be specified prior to encoding the image by `set` methods in the `PNGEncodeParam` class. The chunks that can be set and the methods used to set them are described in the following paragraphs.

### 13.6.6.1 Background Color (bKGD Chunk)

Methods are provided to set and read the suggested background color, which is encoded by the bKGD chunk.

For Palette (indexed color) images, the bKGD chunk contains a single value, which is the palette index of the color to be used as the background.

For Grayscale images, the bKGD chunk contains a single value, which is the gray level to be used as the background. The range of values is 0 to $2^{\text{bitdepth}} - 1$.

For RGB (truecolor) images, the bKGD chunk contains three values, one each for red, green, and blue. Each value has the range of 0 to $2^{\text{bitdepth}} - 1$.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.Palette`

- `void setBackgroundPaletteIndex(int index)`
  sets the palette index of the suggested background color.

- `int getBackgroundPaletteIndex()`
  returns the palette index of the suggested background color.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.Gray`

- `void setBackgroundGray(int gray)`
  sets the suggested gray level of the background.

- `int getBackgroundGray()`
  returns the suggested gray level of the background.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.RGB`

- `void setBackgroundRGB(int[] rgb)`
  sets the RGB value of the suggested background color. The `rgb` parameter should have three entries.

- `int[] getBackgroundRGB()`

  returns the RGB value of the suggested background color.

### 13.6.6.2 Chromaticity (cHRM Chunk)

Applications that need device-independent specification of colors in a PNG file can specify the 1931 CIE (*x*,*y*) chromaticities of the red, green, and blue primaries used in the image, and the referenced white point.

The chromaticity parameter should be a `float` array of length 8 containing the white point *X* and *Y*, red *X* and *Y*, green *X* and *Y*, and blue *X* and *Y* values in order.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `void setChromaticity(float[] chromaticity)`

  sets the white point and primary chromaticities in CIE (*x*,*y*) space.

-
  ```
  void setChromaticity(float whitePointX, float whitePointY,
      float redX, float redY, float greenX, float greenY,
      float blueX, float blueY)
  ```

  a convenience method that calls the array version.

- `float[] getChromaticity()`

  returns the white point and primary chromaticities in CIE (*x*,*y*) space.

### 13.6.6.3 Gamma Correction (gAMA Chunk)

The gamma value specifies the relationship between the image samples and the desired display output intensity as a power function:

$$\text{sample} = \text{light\_out}^{\text{gamma}}$$

If the image's gamma value is unknown, the gAMA chunk should be suppressed. The absence of the gAMA chunk indicates that the gamma is unknown.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `void setGamma(float gamma)`

  sets the gamma value for the image.

- `float getGamma()`

  returns the gamma value for the image.

- `void unsetGamma()`
  suppresses the gAMA chunk from being output.

#### 13.6.6.4 Palette Histogram (hIST Chunk)

The palette histogram is a value that gives the approximate usage frequency of each color in the color palette. If the viewer is unable to provide all the colors listed in the palette, the histogram may help decide how to choose a subset of colors for display. The hIST chunk is only valid with Palette images.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.Palette`

- `void setPaletteHistogram(int[] paletteHistogram)`
  sets the palette histogram for the image.
- `int[] getPaletteHistogram()`
  returns the palette histogram for the image.
- `void unsetPaletteHistogram()`
  suppresses the hIST chunk from being output.

#### 13.6.6.5 Embedded ICC Profile Data (iCCP Chunk)

You can specify that RGB image samples conform to the color space presented by the embedded International Color Consortium profile. The color space of the ICC profile must be an RGB color space.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `void setICCProfileData(byte[] ICCProfileData)`
  sets the ICC profile data.
- `byte[] getICCProfileData()`
  returns the ICC profile data.
- `void unsetICCProfileData()`
  suppresses the iCCP chunk from being output.

#### 13.6.6.6 Physical Pixel Dimensions (pHYS Chunk)

The intended pixel size or aspect ratio for display of the image may be specified in the pHYS chunk. The physical pixel dimensions information is presented as three integer values:

- Pixels per unit, *x* axis
- Pixels per unit, *y* axis
- Unit specifier

The unit specifier may have one of two values:

0 = Unit is unknown
1 = Unit is meters

When the unit specifier is 0, the pHYS chunk defines pixel aspect ratio only; the actual size of the pixels remains unspecified.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `void setPhysicalDimension(int[] physicalDimension)`

  sets the physical pixel dimension.

-
  ```
  void setPhysicalDimension(int xPixelsPerUnit,
      int yPixelsPerUnit, int unitSpecifier)
  ```
  a convenience method that calls the array version.

- `int[] getPhysicalDimension()`

  returns the physical pixel dimension.

#### 13.6.6.7 Significant Bits (sBIT Chunk)

For PNG data that has been converted from a lower sample depth, the significant bits information in the sBIT chunk stores the number of significant bits in the original image. This value allows decoders to recover the original data losslessly, even if the data had a sample depth not directly supported by PNG.

The number of entries in the `significantBits` array must be equal to the number of output bands in the image:

- 1 – for a grayscale image
- 2 – for a grayscale image with alpha
- 3 – for palette or RGB images

- 4 – for RGB images with alpha

**API:** `com.sun.media.jai.codec.PNGEncodeParam.RGB`

- `void setSignificantBits(int[] significantBits)`
  sets the significant bits.
- `int[] getSignificantBits()`
  returns the significant bits.
- `void unsetSignificantBits()`
  suppresses the sBIT chunk from being output.

### 13.6.6.8 Suggested Palette (sPLT Chunk)

A suggested palette may be specified when the display device is not capable of displaying the full range of colors in the image. This palette provides a recommended set of colors, with alpha and frequency information, that can be used to construct a reduced palette to which the image can be quantized.

The suggested palette, as defined by the `PNGSuggestedPaletteEntry` class, consists of the following:

- A palette name – a String that provides a convenient name for referring to the palette
- A `sampleDepth` parameter – must be either 8 or 16
- Red sample
- Green sample
- Blue sample
- Alpha sample
- Frequency – the value is proportional to the fraction of pixels in the image that are closest to that palette entry in RGBA space, before the image has been composited against any background

**API:** `com.sun.media.jai.codec.PNGEncodeParam.Palette`

- `void setSuggestedPalette(PNGSuggestedPaletteEntry[] palette)`
  sets the suggested palette.

- `PNGSuggestedPaletteEntry[] getSuggestedPalette()`
  returns the suggested palette.

- `void unsetSuggestedPalette()`
  suppresses the sPLT chunk from being output.

#### 13.6.6.9 PNG Rendering Intent (sRGB Chunk)

If the PNG image includes an sRGB chunk, the image samples confirm to the sRGB color space and should be displayed using the specified rendering "intent." The rendering intent specifies tradeoffs in colorimetric accuracy. There are four rendering intents:

**Table 13-5 PNG Rendering Intent**

| Parameter | Description |
|---|---|
| `INTENT_PERCEPTUAL` | The "perceptual" intent is for images that prefer good adaptation to the output device gamut at the expense of colorimetric accuracy, such as photographs. |
| `INTENT_RELATIVE` | The "relative colorimetric" intent is for images that require color appearance matching. |
| `INTENT_SATURATION` | The "saturation" intent is for images that prefer preservation of saturation at the expense of hue and lightness. |
| `INTENT_ABSOLUTE` | The "absolute colorimetric" intent is for images that require absolute colorimetry. |

**API:** `com.sun.media.jai.codec.PNGEncodeParam.RGB`

- `void setSRGBIntent(int SRGBIntent)`
  sets the PNG rendering intent.

| | | |
|---|---|---|
| *Parameter*: | `SRGBIntent` | The sRGB rendering intent to be stored with the image. The legal values are 0 = Perceptual, 1 = Relative colorimetric, 2 = Saturation, and 3 = Absolute colorimetric. |

- `int getSRGBIntent()`
  returns the rendering intent.

- `void unsetSRGBIntent()`
  suppresses the sRGB chunk from being output.

### 13.6.6.10 Textual Data (tEXt Chunk)

Textual data can be encoded along with the image in the tEXt chunk. The information stored in this chunk can be an image description or copyright notice. A keyword indicates what the text string contains. The following keywords are defined:

| | |
|---|---|
| `Title` | A title or caption for the image |
| `Author` | The name of the image's creator |
| `Description` | A description of the image |
| `Copyright` | A copyright notice |
| `Creation Time` | The time the original image was created |
| `Software` | The software used to create the image |
| `Disclaimer` | A legal disclaimer |
| `Warning` | A warning of the nature of the image content |
| `Source` | The hardware device used to create the image |
| `Comment` | Miscellaneous information |

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `void setText(String[] text)`

  sets the text string to be encoded with the image.

- `String[] getText()`

  returns the text string to be encoded with the image.

- `void unsetText()`

  suppresses the tEXt chunk from being output.

### 13.6.6.11 Image Modification Timestamp (tIME Chunk)

The tIME chunk provides information on the last time the image was modified. The tIME information is a `Date` and the internal storage format uses UTC regardless of how the `modificationTime` parameter was created.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `void setModificationTime(Date modificationTime)`

  sets the image modification time as a `Date` that will be sent in the tIME chunk.

- `Date getModificationTime()`

  returns the image modification time data that will be sent in the tIME chunk.

- `void unsetModificationTime()`

  suppresses the tIME chunk from being output.

#### 13.6.6.12 Transparency (tRNS Chunk)

The tRNS chunk specifies that the image uses simple transparency. Simple transparency means either alpha values associated with palette entries for Palette images, or a single transparent color, for Grayscale and RGB images.

For Palette images, the tRNS chunk should contain a series of one-byte alpha values, one for each RGB triple in the palette. Each entry indicates that pixels of the corresponding palette index must be treated as having the specified alpha value.

For grayscale images, the tRNS chunk should contain a single gray level value, stored as an int. Pixels of the specified gray value are treated as transparent. If the grayscale image has an alpha value, setting the gray level causes the image's alpha channel to be ignored.

For RGB images, the tRNS chunk should an RGB color value, stored as an int. Pixels of the specified gray value are treated as transparent. If the RGB image has an alpha value, setting the gray level causes the image's alpha channel to be ignored.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.Palette`

- `void setPaletteTransparency(byte[] alpha)`

  sets the alpha values associated with each palette entry. The alpha parameter should have as many entries as there are RGB triples in the palette.

- `byte[] getPaletteTransparency()`

  returns the alpha values associated with each palette entry.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.Gray`

- `void setTransparentGray(int transparentGray)`

  sets the gray value to be used to denote transparency. Setting this attribute will cause the alpha channel of the input image to be ignored.

- `int getTransparentGray()`
  returns the gray value to be used to denote transparency.

**API:** `com.sun.media.jai.codec.PNGEncodeParam.RGB`

- `void setTransparentRGB(int[] transparentRGB)`
  sets the RGB value to be used to denote transparency. Setting this attribute will cause the alpha channel of the input image to be ignored.

- `int[] getTransparentRGB()`
  returns the RGB value to be used to denote transparency.

#### 13.6.6.13 Compressed Text Data (zTXt Chunk)

Text data may be stored in the zTXt chunk, in addition to the text in the tEXt chunk. The zTXt chunk is intended for storing large blocks of text, since the text is compressed.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `void setCompressedText(String[] text)`
  sets the compressed text to be sent in the zTXt chunk.

- `String[] getCompressedText()`
  returns the compressed text to be sent in the zTXt chunk.

- `void unsetCompressedText()`
  suppresses the zTXt chunk from being output.

#### 13.6.6.14 Private Chunks

Private chunks may be added to the output file. These private chunks carry information that is not understood by most other applications. Private chunks should be given names with lowercase second letters to ensure that they do not conflict with any future public chunk information. See the PNG specification for more information on chunk naming conventions.

**API:** `com.sun.media.jai.codec.PNGEncodeParam`

- `synchronized void addPrivateChunk(String type, byte[] data)`
  adds a private chunk to the output file.

- `synchronized int getNumPrivateChunks()`

  returns the number of private chunks to be written to the output file.

- `synchronized String getPrivateChunkType(int index)`

  returns the type of the private chunk at a given index, as a four-character `String`. The index must be smaller than the return value of `getNumPrivateChunks`.

- `synchronized void removeUnsafeToCopyPrivateChunks()`

  removes all private chunks associated with this parameter instance whose “safe-to-copy” bit is not set. This may be advisable when transcoding PNG images.

- `synchronized void removeAllPrivateChunks()`

  remove all private chunks associated with this parameter instance.

## 13.7 Writing PNM Image Files

The PNM format is one of the extensions of the PBM file format (PBM, PGM, and PPM). The portable bitmap format is a lowest-common-denominator monochrome file format. It was originally designed to make it reasonable to mail bitmaps between different types of machines. It now serves as the common language of a large family of bitmap conversion filters.

The PNM format comes in six variants:

- PBM ASCII – three-banded images
- PBM raw – three-banded images
- PGM ASCII – single-banded images
- PGM raw – single-banded images
- PPM ASCII – single-banded images
- PPM raw – single-banded images

The parameter values, then are `RAW` and `ASCII`.

Listing 13-4 shows a code sample for encoding a PNM image.

**Listing 13-4 Encoding a PNM Image**

```
// Create the OutputStream.
OutputStream out = new FileOutputStream(fileToWriteTo);
```

**Listing 13-4 Encoding a PNM Image (Continued)**

```
   // Create the ParameterBlock.
   PNMEncodeParam param = new PNMEncodeParam();
   param.setRaw(true.equals("raw"));

   //Create the PNM image encoder.
   ImageEncoder encoder = ImageCodec.createImageEncoder("PNM",
                                                        out,
                                                        param);
```

**API:** `com.sun.media.jai.codec.PNMEncodeParam`

- `void setRaw(boolean raw)`

  sets the RAWBITS option flag.

- `boolean getRaw()`

  retrieves the RAWBITS option flag.

## 13.8 Writing TIFF Image Files

The TIFF file format is a tag-based file format for storing and interchanging raster images. TIFF files typically come from scanners, frame grabbers, and paint- or photo-retouching programs.

By default, TIFF images in JAI are encoded without any compression and are written out in strips rather than tiles. However, JAI does support image compression, and the writing of tiled TIFF images.

### 13.8.1 TIFF Compression

JAI currently does not support compression of TIFF images.

### 13.8.2 TIFF Tiled Images

By default, the JAI encoder organizes TIFF images into strips. For low- to medium-resolution images, this is adequate. However, for high-resolution (large) images, the images can be accessed more efficiently if the image is divided into roughly square tiles instead of strips.

Writing of tiled TIFF images can be enabled by calling the `setWriteTiled` method.

**API:** `com.sun.media.jai.codec.TIFFEncodeParam`

- `void setWriteTiled(boolean writeTiled)`
  enables writing of TIFF images in tiles rather than in strips.

  | | | |
  |---|---|---|
  | *Parameter*: | `writeTiled` | Specifies whether the image data should be written out in tiled format. |

- `boolean getWriteTiled()`
  returns the value of the `writeTiled` parameter.

CHAPTER 14

# Extending the API

THIS chapter describes how the JAI API may be extended.

## 14.1 Introduction

No image processing API can hope to capture the enormous variety of operations that can be performed on a digital image. Although the JAI API supports a large number of imaging operations, it was designed from the beginning to encourage programmers to write extensions rather than manipulating image data directly. JAI allows virtually *any* image processing algorithm to be added to the API and used as if it were a native part of the API.

The mechanism for adding functionality to the API can be presented at multiple levels of encapsulation and complexity. This allows programmers who wish to add simple things to the API to deal with simple concepts, while more complex extensions have complete control over their environment at the lowest levels of abstraction. The API also supports a variety of programming styles, including an immediate mode and a deferred mode of execution for different types of imaging applications.

## 14.2 Package Naming Convention

All extensions to JAI require the addition of new classes. All new classes are grouped into packages as a convenient means of organizing the new classes and separating the new classes from code libraries provided by others.

All new packages are given a *product name*. A product name is the accepted Java method of using your company’s reversed Internet address to name new packages. This product naming convention helps to guarantee the uniqueness of package names. Supposing that your company’s Internet address is

WebStuff.COM and you wish to create a new package named Prewitt. A good choice of package name would be

```
com.webstuff.Prewitt
```

Or, even

```
com.webstuff.media.jai.Prewitt
```

To uniquely identify the package as part of JAI.

The above new prewitt class file must now be placed into a subdirectory that matches the product name, such as:

com/webstuff/media/jai *for Solaris-based systems*

or

com\webstuff\media\jai *for Windows systems*

The Java convention for class naming is to use initial caps for the name, as in the Prewitt example above. So called multi-word class names use initial caps for each word. For example AddOpImage.

Vendors are encouraged to use unique product names (by means of the Java programming language convention of reversed internet addresses) to maximize the likelihood of a clean installation.

## 14.3 Writing New Operators

To extend the JAI API by creating new operations, you will need to write a new OpImage subclass. This may be done by subclassing one or more existing utility classes to automate some of the details of the operator you wish to implement. For most operators, you need only supply a routine that is capable of producing an arbitrary rectangle of output, given contiguous source data.

Once created, new operators may be made available to users transparently and without user source code changes using the JAI registry mechanism. Existing applications may be tuned for new hardware platforms by strategic insertion of new implementations of existing operators.

To create a new operator, you need to create the following new classes:

- A class that extends the OpImage class or any of its subclasses. This new class does the actual processing. See Section 14.3.1, “Extending the OpImage Class.”

- A class that extends the `OperationDescriptor` class. This new class describes the operation such as name, parameter list, and so on. See Section 14.3.2, "Extending the OperationDescriptor Interface."
- If the operator will function in the Rendered mode only, a class that implements `java.awt.image.renderable.RenderedImageFactory`.

## 14.3.1 Extending the OpImage Class

Every new operator being written must be a subclass of `OpImage` or one of its subclasses. The `OpImage` class currently has the following subclasses:

**Table 14-1 OpImage Subclasses**

| Class | Description |
|---|---|
| `AreaOpImage` | An abstract base class for image operators that require only a fixed rectangular source region around a source pixel in order to compute each destination pixel. |
| `NullOpImage` | Extends: `PointOpImage`<br>A trivial `OpImage` subclass that simply transmits its source unchanged. Potentially useful when an interface requires an `OpImage` but another sort of `RenderedImage` (such as a `TiledImage`) is to be used. |
| `PointOpImage` | An abstract base class for image operators that require only a single source pixel in order to compute each destination pixel. |
| `ScaleOpImage` | Extends: `WarpOpImage`<br>An abstract base class for scale-like operations that require rectilinear backwards mapping and padding by the resampling filter dimensions. |
| `SourcelessOpImage` | An abstract base class for image operators that have no image sources. |
| `StatisticsOpImage` | An abstract base class for image operators that compute statistics on a given region of an image, and with a given sampling rate. |
| `UntiledOpImage` | A general class for single-source operations in which the values of all pixels in the source image contribute to the value of each pixel in the destination image. |
| `WarpOpImage` | A general implementation of image warping, and a superclass for other geometric image operations. |

All abstract methods defined in `OpImage` must be implemented by any new `OpImage` subclass. Specifically, there are two fundamental methods that must be implemented:

| Method | Description |
|---|---|
| `getTile` | Gets a tile for reading. This method is called by the object that has the new operator name as its source with a rectangle as its parameter. The operation is responsible for returning a rectangle filled in with the correct values. |
| `computeRect` | Computes a rectangle of output, given `Raster` sources. The method is called by `getTile` to do the actual computation. The extension must override this method. |

First, you have to decide which of the `OpImage` subclasses to extend. To write a new statistics operation, you would most likely extend the `StatisticsOpImage` class. Each subclass has a specific purpose, as described in Table 14-1.

## 14.3.2 Extending the OperationDescriptor Interface

Operations that are to be created using one of the `JAI.create` methods must be defined in the `registryFile`, which is included in the `jai_core.jar`. Each operation has an OperationDescriptor (denoted as “`odesc`” in the `registryFile`), which provides a textual description of the operation and specifies the number and type of its sources and parameters. The OperationDescriptor also specifies whether the operation supports rendered mode, renderable mode, or both.

Listing 14-1 shows a sample of the `registryFile` contents. Note that this is not the entire `registryFile`, only a small sample showing two operators (absolute and addconst).

**Listing 14-1 registryFile Example**

```
odesc   javax.media.jai.operator.AbsoluteDescriptor    absolute
odesc   javax.media.jai.operator.AddConstDescriptor    addconst

rif com.sun.media.jai.opimage.AbsoluteCRIF
                com.sun.media.jai   absolute   sunabsoluterif
rif com.sun.media.jai.mlib.MlibAbsoluteRIF
                com.sun.media.jai   absolute   mlibabsoluterif
rif com.sun.media.jai.opimage.AddConstCRIF
                com.sun.media.jai   addconst   sunaddconstrif
rif com.sun.media.jai.mlib.MlibAddConstRIF
                com.sun.media.jai   addconst   mlibaddconstrif

crif    com.sun.media.jai.opimage.AbsoluteCRIF      absolute
crif    com.sun.media.jai.opimage.AddConstCRIF      addconst
```

All high-level operation names in JAI (such as `Rotate`, `Convolve`, and `AddConst`) are mapped to instances of `RenderedImageFactory` (RIF) and/or `ContextualRenderedImageFactory` (CRIF) that are capable of instantiating `OpImage` chains to perform the named operation. The RIF is for rendered mode operations only; the CRIF is for operations that can handle renderable mode or both rendered and renderable modes.

To avoid the problems associated with directly editing the `registryFile` and then repackaging it, you can register OperationDescriptors and RIFs and CRIFs using the OperationRegistry's `registerOperationDescription`, and `registerRIF` and `registerCRIF` methods. The only drawback to this method of registration is that the new operator will not be automatically reloaded every time a JAI program is executed., since the operation is not actually present in the `registryFile`. This means that to use the new operation, the operation will always have to be invoked beforehand.

To temporarily register a new operation:

1. **Register the operation name**.

   The high-level operation name, called an *operation descriptor*, is registered by calling the `registerOperationByName()` method or the `registerOperationDescriptor()` method. The operation descriptor name must be unique.

   Once an operation descriptor is registered, it may be obtained by name by calling the `getOperationDescriptor()` method.

2. **Register the set of rendered image factory objects.**

   The rendered image factory (RIF) is registered using the `registerRIF` method. Each RIF is registered with a specific operation name and is given a product name. Similar methods exist for registering a contextual image factory (CRIF).

The `OperationDescriptor` interface provides a comprehensive description of a specific image operation. All of the information regarding the operation, such as the operation name, version, input, and property, should be listed. Any conditions placed on the operation, such as its input format and legal parameter range, should also be included, and the methods to enforce these conditions should be implemented. A set of `PropertyGenerators` may be specified to be used as a basis for the operation's property management.

Each family of the image operation in JAI must have a descriptor that implements this interface. The following basic resource data must be provided:

- GlobalName – a global operation name that is visible to all and is the same in all `Locales`
- LocalName – a localized operation name that may be used as a synonym for the global operation name
- Vendor – the name of the vendor (company name) defining this operation
- Description – a brief description of this operation
- DocURL – a URL where additional documentation on this operation may be found (the javadoc for the operation)
- Version – the version of the operation
- arg0Desc, arg1Desc, etc. – descriptions of the arguments. There must be a property for each argument.
- hint0Desc, hint1Desc, etc. – descriptions of the rendering hints. There must be a property for each hint.

Additional information about the operation must be provided when appropriate. It is also good idea to provide a detailed description of the operation's functionality in the class comment. When all of the above data is provided, the operation can be added to an `OperationRegistry`.

Listing 14-2 shows an example of an operation descriptor for the Clamp operation. Note that the descriptor also contains descriptions of the two required operation parameters, but no hints as these aren't required for the operation.

**Listing 14-2 Operation Descriptor for Clamp Operation**

```
public class ClampDescriptor extends OperationDescriptorImpl {
/**
* The resource strings that provide the general documentation
* and specify the parameter list for this operation.
*/
private static final String[][] resources = {
     {"GlobalName",  "Clamp"},
     {"LocalName",   "Clamp"},
     {"Vendor",      "com.sun.javax.media.jai"},
    {"Description", “Clamps the pixel values of a rendered image”},
     {"DocURL",      "http://java.sun.com/products/java-media/jai/
  forDevelopers/jaiapi/
  javax.media.jai.operator.ClampDescriptor.html"},
     {"Version",     “Beta”)},
     {"arg0Desc",    “The lower boundary for each band.”},
     {"arg1Desc",    “The upper boundary for each band.”}
};
```

As described in Section 3.3, “Processing Graphs,” JAI has two image modes: Rendered and Renderable. An operation supporting the Rendered mode takes `RenderedImages` as its sources, can only be used in a Rendered op chain, and produces a `RenderedImage`. An operation supporting the Renderable mode takes `RenderableImages` as its sources, can only be used in a Renderable op chain, and produces a `RenderableImage`. Therefore, the class types of the sources and the destination of an operation are different between the two modes, but the parameters must be the same for both modes.

All operations must support the rendered mode and implement those methods that supply the information for this mode. Those operations that support the renderable mode must specify this feature using the `isRenderableSupported` method and implement those methods that supply the additional information for the Renderable mode.

Table 14-2 lists the Rendered mode methods. Table 14-3 lists the Renderable mode methods. Table 14-4 lists the methods relative to operation parameters.

**Table 14-2 Rendered Mode Methods**

| Method | Description |
|---|---|
| `isRenderedSupported` | Returns `true` if the operation supports the Rendered image mode. This must be `true` for all operations. |
| `isImmediate` | Returns `true` if the operation should be rendered immediately during the call to `JAI.create`; that is, the operation is placed in immediate mode. |

**Table 14-2 Rendered Mode Methods (Continued)**

| Method | Description |
|---|---|
| `getSourceClasses` | Returns an array of `Class`es that describe the types of sources required by this operation in the Rendered image mode. |
| `getDestClass` | Returns a `Class` that describes the type of destination this operation produces in the Rendered image mode. |
| `validateArguments` | Returns `true` if this operation is capable of handling the input rendered source(s) and/or parameter(s) specified in the `ParameterBlock`. |

**Table 14-3 Renderable Mode Methods**

| Method | Description |
|---|---|
| `isRenderableSupported` | Returns `true` if the operation supports the Renderable image mode. |
| `getRenderableSourceClasses` | Returns an array of `Class`es that describe the types of sources required by this operation in the Renderable image mode. |
| `getRenderableDestClass` | Returns a `Class` that describes the type of destination this operation produces in the Renderable image mode. |
| `validateRenderableArguments` | Returns `true` if this operation is capable of handling the input Renderable source(s) and/or parameter(s) specified in the `ParameterBlock`. |

**Table 14-4 Parameter Methods**

| Method | Description |
|---|---|
| `getNumParameters` | Returns the number of parameters (not including the sources) required by this operation. |
| `getParamClasses` | Returns an array of `Class`es that describe the types of parameters required by this operation. |
| `getParamNames` | Returns an array of `String`s that are the localized parameter names of this operation. |
| `getParamDefaults` | Returns an array of `Object`s that define the default values of the parameters for this operation. |
| `getParamDefaultValue` | Returns the default value of a specified parameter. |
| `getParamMinValue` | Returns the minimum legal value of a specified numeric parameter for this operation. |
| `getParamMaxValue` | Returns the maximum legal value of a specified numeric parameter for this operation. |

**API:** `javax.media.jai.OperationRegistry`

- `void registerOperationDescriptor(OperationDescriptor odesc,`
  `    String operationName)`

  registers an `OperationDescriptor` with the registry. Each operation must have an `OperationDescriptor` before `registerRIF()` may be called to add RIFs to the operation.

| | | |
|---|---|---|
| *Parameter*: | `odesc` | An `OperationDescriptor` containing information about the operation. |
| | `operationName` | The operation name as a `String`. |

  A `OperationDescriptor` cannot be registered under an operation name under which another `OperationDescriptor` was registered previously. If such an attempt is made, an Error will be thrown.

- `void registerOperationByName(String odescClassName,`
  `    String operationName)`

  registers an `OperationDescriptor` by its class name.

| | | |
|---|---|---|
| *Parameter*: | `odescClassName` | The fully-qualified class name of the `OperationDescriptor`. |
| | `operationName` | The operation name as a `String`. |

- `void unregisterOperationDescriptor(String operationName)`

  unregisters an `OperationDescriptor` from the registry.

- `void registerRIF(String operationName, String productName,`
  `    RenderedImageFactory RIF)`

  registers a `RIF` with a particular product and operation.

| | | |
|---|---|---|
| *Parameter*: | `operationName` | The operation name as a `String`. |
| | `productName` | The product name, as a `String`. |
| | `RIF` | The `RenderedImageFactory` to be registered. |

-
  ```
  void registerRIFByClassName(String operationName,
      String productName, String RIFClassName)
  ```

  registers a `RIF` with a particular product and operation, constructing an instance using its class name.

| | | |
|---|---|---|
| *Parameter*: | `operationName` | The operation name as a `String`. |
| | `productName` | The product name, as a `String`. |
| | `RIFClassName` | The fully-qualified class name of a `RenderedImageFactory`. |

## 14.4 Iterators

Iterators are provided to help the programmer who writes extensions to the JAI API and does not want to use any of the existing API methods for traversing pixels. Iterators define the manner in which the source image pixels are traversed for processing. Iterators may be used both in the implementation of `computeRect` methods or `getTile` methods of OpImage subclasses, and for ad-hoc pixel-by-pixel image manipulation.

Iterators provide a mechanism for avoiding the need to cobble sources, as well as to abstract away the details of source pixel formatting. An iterator is instantiated to iterate over a specified rectangular area of a source `RenderedImage` or `Raster`. The iterator returns pixel values in `int`, `float`, or `double` format, automatically promoting integral values smaller than 32 bits to `int` when reading, and performing the corresponding packing when writing.

JAI offers three different types of iterator, which should cover nearly all of a programmer's needs. However, extenders may wish to build others for more specialized needs.

The most basic iterator is `RectIter`, which provides the ability to move one line or pixel at a time to the right or downwards, and to step forward in the list of bands. `RookIter` offers slightly more functionality than `RectIter`, allowing leftward and upward movement and backwards motion through the set of bands. Both `RectIter` and `RookIter` allow jumping to an arbitrary line or pixel, and reading and writing of a random band of the current pixel. The `RookIter` also allows jumping back to the first line or pixel, and to the last line or pixel.

`RandomIter` allows an unrelated set of samples to be read by specifying their *x* and *y* coordinates and band offset. The `RandomIter` will generally be slower than either the `RectIter` or `RookIter`, but remains useful for its ability to hide pixel formats and tile boundaries.

Figure 14-1 shows the Iterator package hierarchy. The classes are described in the following paragraphs.

### 14.4.1 RectIter

The `RectIter` interface represents an iterator for traversing a read-only image in top-to-bottom, left-to-right order (Figure 14-2). The RectIter traversal will generally be the fastest style of iterator, since it does not need to perform bounds checks against the top or left edges of tiles. The `WritableRectIter` interface traverses a read/write image in the same manner as the RectIter.

The iterator is initialized with a particular rectangle as its bounds. The initialization takes place in a factory method (the `RectIterFactory` class) and is not a part of the iterator interface itself. Once initialized, the iterator may be reset to its initial state by means of the `startLines()`, `startPixels()`, and `startBands()` methods. Its position may be advanced using the `nextLine()`, `jumpLines()`, `nextPixel()`, `jumpPixels()`, and `nextBand()` methods.

**Class Hierarchy**

```
Object
 ├─RandomIterFactory
 ├─RectIterFactory
 └─RookIterFactory
```

**Interface Hierarchy**

```
RandomIter
 └─WritableRandomIter

RectIter
 ├─WritableRectIter
 └─RookIter
    └─WritableRookIter
```

**Figure 14-1 Iterator Hierarchy**

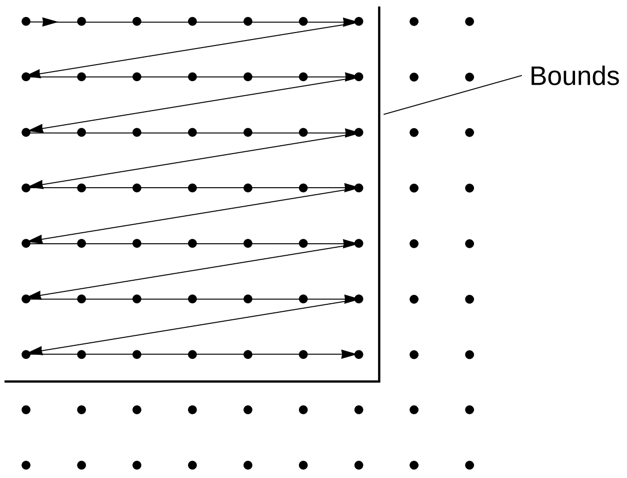


**Figure 14-2 RectIter Traversal Pattern**

The `WritableRookIter` interface adds the ability to alter the source pixel values using the various `setSample()` and `setPixel()` methods.

An instance of `RectIter` may be obtained by means of the `RectIterFactory.create()` method, which returns an opaque object implementing this interface.

**API:** `javax.media.jai.iterator.RectIterFactory`

- `static RectIter create(RenderedImage im, Rectangle bounds)`

  constructs and returns an instance of `RectIter` suitable for iterating over the given bounding rectangle within the given `RenderedImage` source. If the `bounds` parameter is null, the entire image will be used.

| *Parameters*: | | |
|---|---|---|
| | `im` | A read-only `RenderedImage` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

- `static RectIter create(Raster ras, Rectangle bounds)`

  constructs and returns an instance of `RectIter` suitable for iterating over the given bounding rectangle within the given `Raster` source. If the `bounds` parameter is null, the entire Raster will be used.

| | | |
|---|---|---|
| *Parameters*: | `ras` | A read-only `Raster` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

- `static WritableRectIter createWritable(WritableRenderedImage im, Rectangle bounds)`

  constructs and returns an instance of `WritableRectIter` suitable for iterating over the given bounding rectangle within the given `WritableRenderedImage` source. If the `bounds` parameter is null, the entire image will be used.

| | | |
|---|---|---|
| *Parameters*: | `im` | A `WritableRenderedImage` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

- `static WritableRectIter createWritable(WritableRaster ras, Rectangle bounds)`

  constructs and returns an instance of `WritableRectIter` suitable for iterating over the given bounding rectangle within the given `WritableRaster` source. If the `bounds` parameter is null, the entire `Raster` will be used.

| | | |
|---|---|---|
| *Parameters*: | `ras` | A `WritableRaster` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

**API:** `javax.media.jai.iterator.RectIter`

- `void startLines()`

  sets the iterator to the first line of its bounding rectangle. The pixel and band offsets are unchanged.

- `void startPixels()`

  sets the iterator to the leftmost pixel of its bounding rectangle. The line and band offsets are unchanged.

- `void startBands()`

  sets the iterator to the first band of the image. The pixel column and line are unchanged.

- `void nextLine()`

  sets the iterator to the next line of the image. The pixel and band offsets are unchanged. If the iterator passes the bottom line of the rectangles, calls to `get()` methods are not valid.

- `void jumpLines(int num)`

  jumps downward `num` lines from the current position. The `num` parameter may be negative. The pixel and band offsets are unchanged.

- `void nextPixel()`

  sets the iterator to the next pixel in the image (that is, move rightward). The line and band offsets are unchanged.

- `void jumpPixels(int num)`

  jumps rightward `num` pixels from the current position. The `num` parameter may be negative. The line and band offsets are unchanged.

- `void nextBand()`

  sets the iterator to the next band in the image. The pixel column and line are unchanged.

### 14.4.2 RookIter

The `RookIter` interface represents an iterator for traversing a read-only image using arbitrary up-down and left-right moves (Figure 14-3 shows two of the possibilities for traversing the pixels). The RookIter traversal will generally be somewhat slower than a corresponding instance of `RectIter`, since it must perform bounds checks against the top and left edges of tiles in addition to their bottom and right edges. The `WritableRookIter` interface traverses a read/write image in the same manner as the RookIter.

An instance of RookIter may be obtained by means of the `RookIterFactory.create()` or `RookIterFactory.createWritable()` methods, which return an opaque object implementing this interface. The iterator is initialized with a particular rectangle as its bounds. This initialization takes place in a factory method (the `RookIterFactory` class) and is not a part of the iterator interface itself.

Once initialized, the iterator may be reset to its initial state by means of the `startLines()`, `startPixels()`, and `startBands()` methods. As with `RectIter`, its position may be advanced using the `nextLine()`, `jumpLines()`, `nextPixel()`, `jumpPixels()`, and `nextBand()` methods.

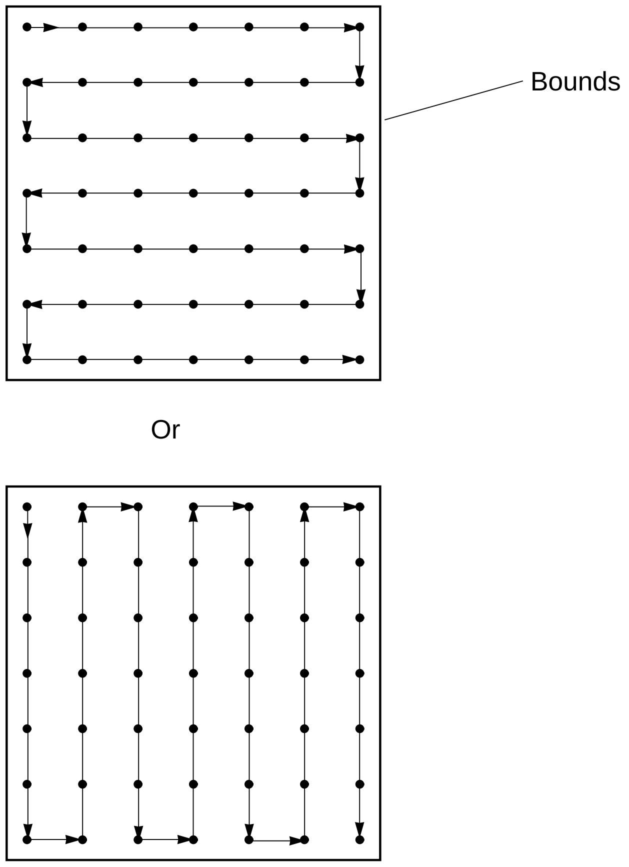


**Figure 14-3 RookIter Traversal Patterns**

**API:** `avax.media.jai.iterator.RookIterFactory`

- `static RookIter create(RenderedImage im, Rectangle bounds)`

  constructs and returns an instance of `RookIter` suitable for iterating over the given bounding rectangle within the given `RenderedImage` source. If the `bounds` parameter is null, the entire image will be used.

| | | |
|---|---|---|
| *Parameters*: | `im` | A read-only `RenderedImage` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

- `static RookIter create(Raster ras, Rectangle bounds)`

  constructs and returns an instance of `RookIter` suitable for iterating over the given bounding rectangle within the given `Raster` source. If the bounds parameter is null, the entire `Raster` will be used.

| | | |
|---|---|---|
| *Parameters*: | `ras` | A read-only `Raster` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

-
  ```
  static WritableRookIter createWritable(WritableRenderedImage
      im, Rectangle bounds)
  ```

  constructs and returns an instance of `WritableRookIter` suitable for iterating over the given bounding rectangle within the given `WritableRenderedImage` source. If the `bounds` parameter is null, the entire image will be used.

| | | |
|---|---|---|
| *Parameters*: | `im` | A `WritableRenderedImage` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

-
  ```
  static WritableRookIter createWritable(WritableRaster ras,
      Rectangle bounds)
  ```

  constructs and returns an instance of `WritableRookIter` suitable for iterating over the given bounding rectangle within the given `WritableRaster` source. If the `bounds` parameter is null, the entire `Raster` will be used.

| | | |
|---|---|---|
| *Parameters*: | `ras` | A `WritableRaster` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

### 14.4.3 RandomIter

The `RandomIter` interface represents an iterator that allows random access to any sample within its bounding rectangle. The flexibility afforded by this class will generally exact a corresponding price in speed and setup overhead.

The iterator is initialized with a particular rectangle as its bounds. This initialization takes place in a factory method (the `RandomIterFactory` class) and is not a part of the iterator interface itself. An instance of `RandomIter` may be obtained by means of the `RandomIterFactory.create()` method, which returns an opaque object implementing this interface.

The `getSample()`, `getSampleFloat()`, and `getSampleDouble()` methods are provided to allow read-only access to the source data. The `getPixel()` methods allow retrieval of all bands simultaneously.

**API:** `javax.media.jai.iterator.RandomIterFactory`

- `static RandomIter create(RenderedImage im, Rectangle bounds)`

  constructs and returns an instance of `RandomIter` suitable for iterating over the given bounding rectangle within the given `RenderedImage` source. If the `bounds` parameter is null, the entire image will be used.

| | | |
|---|---|---|
| *Parameters*: | `im` | A read-only `RenderedImage` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

- `static RandomIter create(Raster ras, Rectangle bounds)`

  constructs and returns an instance of `RandomIter` suitable for iterating over the given bounding rectangle within the given `Raster` source. If the `bounds` parameter is null, the entire `Raster` will be used.

| | | |
|---|---|---|
| *Parameters*: | `ras` | A read-only `Raster` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

-
  ```
  static WritableRandomIter createWritable(WritableRenderedImage
      im, Rectangle bounds)
  ```

  constructs and returns an instance of `WritableRandomIter` suitable for iterating over the given bounding rectangle within the given

`WritableRenderedImage` source. If the `bounds` parameter is null, the entire image will be used.

| | | |
|---|---|---|
| *Parameters*: | `im` | A `WritableRenderedImage` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

- `static WritableRandomIter createWritable(WritableRaster ras, Rectangle bounds)`

  constructs and returns an instance of `WritableRandomIter` suitable for iterating over the given bounding rectangle within the given `WritableRaster` source. If the `bounds` parameter is null, the entire Raster will be used.

| | | |
|---|---|---|
| *Parameters*: | `ras` | A read-only `Raster` source. |
| | `bounds` | The bounding `Rectangle` for the iterator, or null. |

### 14.4.4 Example RectIter

Listing 14-3 shows an example of the construction of a new `RectIter`.

**Listing 14-3 Example RectIter (Sheet 1 of 4)**

```
import java.awt.Rectangle;
import java.awt.image.ColorModel;
import java.awt.image.DataBuffer;
import java.awt.image.PixelInterleavedSampleModel;
import java.awt.image.SampleModel;
import java.util.Random;
import javax.media.jai.*;
import javax.media.jai.iterator.*;

class RectIterTest {

    int width = 10;
    int height = 10;
    int tileWidth = 4;
    int tileHeight = 4;

    public static void main(String[] args) {
        new RectIterTest();
    }

    public RectIterTest() {
```

**Listing 14-3 Example RectIter (Sheet 2 of 4)**

```
            Random rand = new Random(1L);
            Rectangle rect = new Rectangle();

            int[] bandOffsets = { 2, 1, 0 };
            SampleModel sampleModel =
              new PixelInterleavedSampleModel(DataBuffer.TYPE_BYTE,
                                              tileWidth, tileHeight,
                                                 3, 3*tileWidth,
                                                 bandOffsets);
            ColorModel colorModel = null;

           TiledImage im = new TiledImage(0, 0, width, height, 0, 0,
                                          sampleModel,
                                          colorModel);

            int[][][] check = new int[width][height][3];
            int x, y, b;

            for (int i = 0; i < 10; i++) {
                rect.x = rand.nextInt(width);
                rect.width = rand.nextInt(width - rect.x) + 1;

                rect.y = rand.nextInt(height);
                rect.height = rand.nextInt(height - rect.y) + 1;

    System.out.println("Filling rect " + rect + " with " + i);

    WritableRectIter witer = RectIterFactory.createWritable(im,
                                                           rect);

                b = 0;
                witer.startBands();
                while (!witer.finishedBands()) {
                    y = rect.y;
                    witer.startLines();
                    while (!witer.finishedLines()) {
                        x = rect.x;
                        witer.startPixels();
                        while (!witer.finishedPixels()) {
                            witer.setSample(i);
                            check[x][y][b] = i;

                            ++x;
                            witer.nextPixel();
                        }
```

**Listing 14-3 Example RectIter (Sheet 3 of 4)**

```
                    ++y;
                    witer.nextLine();
                }

                ++b;
                witer.nextBand();
            }
        }

        rect.x = 0;
        rect.y = 0;
        rect.width = width;
        rect.height = height;
      RectIter iter = RectIterFactory.createWritable(im, rect);

        b = 0;
        iter.startBands();
        while (!iter.finishedBands()) {
            System.out.println();

            y = 0;
            iter.startLines();
            while (!iter.finishedLines()) {

                x = 0;
                iter.startPixels();
                while (!iter.finishedPixels()) {
                    int val = iter.getSample();
                    System.out.print(val);

                    if (val != check[x][y][b]) {
                 System.out.print("(" + check[x][y][b] + ")  ");
                    } else {
                        System.out.print("     ");
                    }

                    ++x;
                    iter.nextPixel();
                }

                ++y;
                iter.nextLine();
                System.out.println();
            }
```

**Listing 14-3 Example RectIter (Sheet 4 of 4)**

```
                ++b;
                iter.nextBand();
            }
        }
    }
```

## 14.5 Writing New Image Decoders and Encoders

The `sample` directory contains an example of how to create a new image codec. The example is of a PNM codec, but can be used as a basis for creating any codec. The PNM codec consists of three files:

| File Name | Description |
|---|---|
| `SamplePNMCodec.java` | Defines a subclass of `ImageCodec` for handling the PNM family of image files. |
| `SamplePNMImageDecoder.java` | Defines an `ImageDecoder` for the PNM family of image files. Necessary for reading PNM files. |
| `SamplePNMImageEncoder.java` | Defines an `ImageEncoder` for the PNM family of image files. Necessary for writing PNM files. |

### 14.5.1 Image Codecs

**Note:** The codec classes are provided for the developer as a convenience for file IO. These classes are not part of the official Java Advanced Imaging API and are subject to change as a result of the near future File IO extension API. Until the File IO extension API is defined, these classes and functions will be supported for JAI use.

The `ImageCodec` class allows the creation of image decoders and encoders. Instances of `ImageCodec` may be registered by name. The `registerCodec` method associates an `ImageCodec` with the given name. Any codec previously associated with the name is discarded. Once a codec has been registered, the name associated with it may be used as the `name` parameter in the `createImageEncoder` and `createImageDecoder` methods.

The `ImageCodec` class maintains a registry of `FormatRecognizer` objects that examine an `InputStream` and determine whether it adheres to the format handled by a particular `ImageCodec`. A `FormatRecognizer` is added to the registry with the `registerFormatRecognizer` method. The `unregisterFormatRecognizer` method removes a previously registered `FormatRecognizer` from the registry.

The `getCodec` method returns the `ImageCodec` associated with a given name. If no codec is registered with the given name, `null` is returned.

**API:** `com.sun.media.jai.codec.ImageCodec`

-
  ```
  static ImageEncoder createImageEncoder(String name,
      OutputStream dst, ImageEncodeParam param)
  ```
  returns an `ImageEncoder` object suitable for encoding to the supplied `OutputStream`, using the supplied `ImageEncodeParam` object.

| | | |
|---|---|---|
| *Parameter*: | `name` | The name associated with the codec. |
| | `dst` | An `OutputStream` to write to. |
| | `param` | An instance of ImageEncodeParam suitable for use with the named codec, or null. |

-
  ```
  static ImageEncoder createImageEncoder(String name,
      OutputStream dst)
  ```
  returns an `ImageEncoder` object suitable for encoding to the supplied `OutputStream` object. A null `ImageEncodeParam` is used.

-
  ```
  static ImageDecoder createImageDecoder(String name,
      InputStream src, ImageDecodeParam param)
  ```
  returns an `ImageDecoder` object suitable for decoding from the supplied `InputStream`, using the supplied `ImageDecodeParam` object.

| | | |
|---|---|---|
| *Parameter*: | `name` | The name associated with the codec. |
| | `src` | An `InputStream` to read from. |
| | `param` | An instance of ImageEncodeParam suitable for use with the named codec, or null. |

-
  ```
  static ImageDecoder createImageDecoder(String name,
      InputStream src)
  ```
  returns an `ImageDecoder` object suitable for decoding from the supplied `InputStream`. A null `ImageDecodeParam` is used.

- `static void registerCodec(String name, ImageCodec codec)`

  associates an `ImageCodec` with the given name. Case is not significant. Any codec previously associated with the name is discarded.

  | | | |
  |---|---|---|
  | *Parameter*: | `name` | The name associated with the codec. |
  | | `codec` | The `ImageCodec` object to be associated with the given name. |

- `static void unregisterCodec(String name)`

  removes the association between a given `name` and an `ImageCodec` object. Case is not significant.

- `static ImageCodec getCodec(String name)`

  returns the `ImageCodec` associated with the given name. If no codec is registered with the given name, null is returned. Case is not significant.

  | | | |
  |---|---|---|
  | *Parameter*: | `name` | The name associated with the codec. |

APPENDIX A

# Program Examples

THIS appendix contains fully-operational JAI program examples.

The examples in this appendix are provided to demonstrate how to create simple programs using JAI. Although these examples can be compiled and run, they are not intended to be used that way since they are pretty simple and would not be particularly interesting, visually.

## A.1 Lookup Operation Example

Listing A-1 shows an example of the Lookup operation. This example program decodes a TIFF image file into a RenderedImage. If the TIFF image is an unsigned short type image, the program performs a Lookup operation to convert the image into a byte type image. Finally, the program displays the byte image.

**Listing A-1 Example Lookup Program (Sheet 1 of 3)**

```
import java.awt.Frame;
import java.awt.RenderingHints;
import java.awt.image.DataBuffer;
import java.awt.image.renderable.ParameterBlock;
import java.io.IOException;
import javax.media.jai.JAI;
import javax.media.jai.LookupTableJAI;
import javax.media.jai.RenderedOp;
import com.sun.media.jai.codec.FileSeekableStream;
import com.sun.media.jai.codec.TIFFDecodeParam;
import javax.media.jai.widget.ScrollingImagePanel;

public class LookupSampleProgram {

    // The main method.
    public static void main(String[] args) {
```

**Listing A-1 Example Lookup Program (Sheet 2 of 3)**

```
   // Validate input.
           if (args.length != 1) {
            System.out.println(“Usage: java LookupSampleProgram “ +
                                 “TIFF_image_filename”);
               System.exit(-1);
           }

   // Create an input stream from the specified file name to be
   // used with the TIFF decoder.
           FileSeekableStream stream = null;
           try {
               stream = new FileSeekableStream(args[0]);
           } catch (IOException e) {
               e.printStackTrace();
               System.exit(0);
           }

   // Store the input stream in a ParameterBlock to be sent to
   // the operation registry, and eventually to the TIFF
   // decoder.
           ParameterBlock params = new ParameterBlock();
           params.add(stream);

   // Specify to TIFF decoder to decode images as they are and
   // not to convert unsigned short images to byte images.
           TIFFDecodeParam decodeParam = new TIFFDecodeParam();
           decodeParam.setDecodePaletteAsShorts(true);

   // Create an operator to decode the TIFF file.
           RenderedOp image1 = JAI.create(“tiff”, params);

   // Find out the first image’s data type.
           int dataType = image1.getSampleModel().getDataType();
           RenderedOp image2 = null;
           if (dataType == DataBuffer.TYPE_BYTE) {
   // Display the byte image as it is.
               System.out.println(“TIFF image is type byte.”);
               image2 = image1;
           } else if (dataType == DataBuffer.TYPE_USHORT) {

   // Convert the unsigned short image to byte image.
               System.out.println(“TIFF image is type ushort.”);
```

**Listing A-1 Example Lookup Program (Sheet 3 of 3)**

```
// Setup a standard window-level lookup table. */
            byte[] tableData = new byte[0x10000];
            for (int i = 0; i < 0x10000; i++) {
                tableData[i] = (byte)(i >> 8);
            }

// Create a LookupTableJAI object to be used with the
// “lookup” operator.
          LookupTableJAI table = new LookupTableJAI(tableData);

// Create an operator to lookup image1.
            image2 = JAI.create(“lookup”, image1, table);

        } else {
           System.out.println(“TIFF image is type “ + dataType +
                                “, and will not be displayed.”);
            System.exit(0);
        }

// Get the width and height of image2.
        int width = image2.getWidth();
        int height = image2.getHeight();

// Attach image2 to a scrolling panel to be displayed.
        ScrollingImagePanel panel = new ScrollingImagePanel(
                                        image2, width, height);

// Create a frame to contain the panel.
        Frame window = new Frame(“Lookup Sample Program”);
        window.add(panel);
        window.pack();
        window.show();
    }
}
```

## A.2 Adding an OperationDescriptor Example

**Chapter 14, “Extending the API**,” describes how to extend the API by writing custom OperationDescriptors. Listing A-2 shows the construction of an `OperationDescriptor`, called `SampleDescriptor`, that is both an

`OperationDescriptor` and a `RenderedImageFactory`. The operation created here is called `Sample` and takes two parameters for the operation.

**Listing A-2 Example OperationDescriptor (Sheet 1 of 8)**

```
   import java.awt.Rectangle;
   import java.awt.RenderingHints;
   import java.awt.image.ComponentSampleModel;
   import java.awt.image.DataBuffer;
   import java.awt.image.DataBufferByte;
   import java.awt.image.Raster;
   import java.awt.image.RenderedImage;
   import java.awt.image.SampleModel;
   import java.awt.image.WritableRaster;
   import java.awt.image.renderable.ParameterBlock;
   import java.awt.image.renderable.RenderedImageFactory;
   import javax.media.jai.ImageLayout;
   import javax.media.jai.OperationDescriptorImpl;
   import javax.media.jai.OpImage;
   import javax.media.jai.PointOpImage;
   import javax.media.jai.RasterAccessor;

   // A single class that is both an OperationDescriptor and
   // a RenderedImageFactory along with the one OpImage it is
   // capable of creating.  The operation implemented is a variation
   // on threshold, although the code may be used as a template for
   // a variety of other point operations.
   public class SampleDescriptor extends OperationDescriptorImpl
                                 implements RenderedImageFactory {

   // The resource strings that provide the general documentation
   // and specify the parameter list for the “Sample” operation.
   private static final String[][] resources = {
       {“GlobalName”,  “Sample”},
       {“LocalName”,   “Sample”},
       {“Vendor”,      “com.mycompany”},
       {“Description”, “A sample operation that thresholds source
                        pixels”},
       {“DocURL”,      “http://www.mycompany.com/
                        SampleDescriptor.html”},
       {“Version”,     “1.0”},
       {“arg0Desc”,    “param1”},
       {“arg1Desc”,    “param2”}
       };
```

**Listing A-2 Example OperationDescriptor (Sheet 2 of 8)**

```
// The parameter names for the “Sample” operation. Extenders may
// want to rename them to something more meaningful.
    private static final String[] paramNames = {
        “param1”, “param2”
    };

// The class types for the parameters of the “Sample” operation.
// User defined classes can be used here as long as the fully
// qualified name is used and the classes can be loaded.
    private static final Class[] paramClasses = {
        java.lang.Integer.class, java.lang.Integer.class
    };

// The default parameter values for the “Sample” operation
// when using a ParameterBlockJAI.
    private static final Object[] paramDefaults = {
        new Integer(0), new Integer(255)
    };

// Constructor.
public SampleDescriptor() {
   super(resources, 1, paramClasses, paramNames, paramDefaults);
    }

// Creates a SampleOpImage with the given ParameterBlock if the
// SampleOpImage can handle the particular ParameterBlock.
    public RenderedImage create(ParameterBlock paramBlock,
                                RenderingHints renderHints) {
        if (!validateParameters(paramBlock)) {
            return null;
        }
      return new SampleOpImage(paramBlock.getRenderedSource(0),
                          new ImageLayout(),
                       (Integer)paramBlock.getObjectParameter(0),
                      (Integer)paramBlock.getObjectParameter(1));
    }
```

**Listing A-2 Example OperationDescriptor (Sheet 3 of 8)**

```
// Checks that all parameters in the ParameterBlock have the
// correct type before constructing the SampleOpImage
public boolean validateParameters(ParameterBlock paramBlock) {
        for (int i = 0; i < this.getNumParameters(); i++) {
            Object arg = paramBlock.getObjectParameter(i);
            if (arg == null) {
                return false;
            }
            if (!(arg instanceof Integer)) {
                return false;
            }
        }
        return true;
    }
}

// SampleOpImage is an extension of PointOpImage that takes two
// integer parameters and one source and performs a modified
// threshold operation on the given source.
class SampleOpImage extends PointOpImage {

    private int param1;
    private int param2;

// A dummy constructor used by the class loader. */
    public SampleOpImage() {}

/** Constructs an SampleOpImage. The image dimensions are copied
*  from the source image.  The tile grid layout, SampleModel, and
*  ColorModel may optionally be specified by an ImageLayout
*  object.
*
* @param source    a RenderedImage.
* @param layout    an ImageLayout optionally containing the tile
*                  grid layout, SampleModel, and ColorModel, or
*                  null.
*/
    public SampleOpImage(RenderedImage source,
                         ImageLayout layout,
                         Integer param1,
                         Integer param2) {
        super(source, null, layout, true);
        this.param1 = param1.intValue();
        this.param2 = param2.intValue();
    }
```

**Listing A-2 Example OperationDescriptor (Sheet 4 of 8)**

```
/**
* Performs a modified threshold operation on the pixels in a
* given rectangle.  Sample values below a lower limit are clamped
* to 0, while those above an upper limit are clamped to 255.  The
* results are returned in the input WritableRaster dest.  The
* sources are cobbled.
*
* @param sources   an array of sources, guarantee to provide all
*              necessary source data for computing the rectangle.
* @param dest  a tile that contains the rectangle to be computed.
* @param destRect the rectangle within this OpImage to be
*                  processed.
*/
    protected void computeRect(Raster[] sources,
                               WritableRaster dest,
                               Rectangle destRect) {
        Raster source = sources[0];
        Rectangle srcRect = mapDestRect(destRect, 0);

// RasterAccessor is a convienient way to represent any given
// Raster in a usable format.  It has very little overhead if
// the underlying Raster is in a common format (PixelSequential
// for this release) and allows generic code to process
// a Raster with an exotic format.  Essentially, it allows the
// common case to processed quickly and the rare case to be
// processed easily.

// This “best case” formatTag is used to create a pair of
// RasterAccessors for processing the source and dest rasters

RasterFormatTag[] formatTags = getFormatTags();
RasterAccessor srcAccessor =
   new RasterAccessor(sources[0], srcRect, formatTags[0],
                      getSource(0).getColorModel());
RasterAccessor dstAccessor =
   new RasterAccessor(dest, destRect, formatTags[1],
                      getColorModel());
```

**Listing A-2 Example OperationDescriptor (Sheet 5 of 8)**

```
    // Depending on the base dataType of the RasterAccessors,
    // either the byteLoop or intLoop method is called.  The two
    // functions are virtually the same, except for the data type
    // of the underlying arrays.
            switch (dstAccessor.getDataType()) {
            case DataBuffer.TYPE_BYTE:
                byteLoop(srcAccessor,dstAccessor);
                break;
            case DataBuffer.TYPE_INT:
                intLoop(srcAccessor,dstAccessor);
                break;
            default:
                String className = this.getClass().getName();
                throw new RuntimeException(className +
                                " does not implement computeRect" +
                                   " for short/float/double data");
            }

    // If the RasterAccessor object set up a temporary buffer for the
    // op to write to, tell the RasterAccessor to write that data
    // to the raster now that we're done with it.
            if (dstAccessor.isDataCopy()) {
                dstAccessor.clampDataArrays();
                dstAccessor.copyDataToRaster();
            }
        }

    /**
    * Computes an area of a given byte-based destination Raster using
    * a souce RasterAccessor and a destination RasterAccesor.
    * Processing is done as if the bytes are unsigned, even though
    * the Java language has support only for signed bytes as a
    * primitive datatype.
    */
    private void byteLoop(RasterAccessor src, RasterAccessor dst) {
            int dwidth = dst.getWidth();
            int dheight = dst.getHeight();
            int dnumBands = dst.getNumBands();

            byte dstDataArrays[][] = dst.getByteDataArrays();
            int dstBandOffsets[] = dst.getBandOffsets();
            int dstPixelStride = dst.getPixelStride();
            int dstScanlineStride = dst.getScanlineStride();
```

**Listing A-2 Example OperationDescriptor (Sheet 6 of 8)**

```
            byte srcDataArrays[][] = src.getByteDataArrays();
            int srcBandOffsets[] = src.getBandOffsets();
            int srcPixelStride = src.getPixelStride();
            int srcScanlineStride = src.getScanlineStride();

            byte bp1 = (byte)(param1 & 0xff);
            byte bp2 = (byte)(param2 & 0xff);

            // A standard imaging loop
            for (int k = 0; k < dnumBands; k++)  {
                byte dstData[] = dstDataArrays[k];
                byte srcData[] = srcDataArrays[k];
                int srcScanlineOffset = srcBandOffsets[k];
                int dstScanlineOffset = dstBandOffsets[k];
                for (int j = 0; j < dheight; j++)  {
                    int srcPixelOffset = srcScanlineOffset;
                    int dstPixelOffset = dstScanlineOffset;
                    for (int i = 0; i < dwidth; i++)  {

   // This code can be specialized by rewriting the
   // following block of code to do some other
   // operation.
   //
   //  Some examples:
   //   InvertOp:
   //      dstData[dstPixelOffset] =
   //        (byte)(0xff & ~srcData[srcPixelOffset]);
   //
   //   AddConst:
   //      dstData[dstPixelOffset] =
   //        (byte)(0xff & (srcData[srcPixelOffset]+param1));
   //
   //   Currently, the operation performs a threshold.

                int pixel = srcData[srcPixelOffset] & 0xff;
                if (pixel < param1) {
                    dstData[dstPixelOffset] = 0; // bp1;
                } else if (pixel > param2) {
                    dstData[dstPixelOffset] = (byte)255; // bp2;
                } else {
                 dstData[dstPixelOffset] = srcData[srcPixelOffset];
                }
```

**Listing A-2 Example OperationDescriptor (Sheet 7 of 8)**

```
                        srcPixelOffset += srcPixelStride;
                        dstPixelOffset += dstPixelStride;
                    }
                    srcScanlineOffset += srcScanlineStride;
                    dstScanlineOffset += dstScanlineStride;
                }
            }
        }

    /**
    * Computes an area of a given int-based destination Raster using
    * a source RasterAccessor and a destination RasterAccesor.
    */
    private void intLoop(RasterAccessor src, RasterAccessor dst) {
            int dwidth = dst.getWidth();
            int dheight = dst.getHeight();
            int dnumBands = dst.getNumBands();

            int dstDataArrays[][] = dst.getIntDataArrays();
            int dstBandOffsets[] = dst.getBandOffsets();
            int dstPixelStride = dst.getPixelStride();
            int dstScanlineStride = dst.getScanlineStride();

            int srcDataArrays[][] = src.getIntDataArrays();
            int srcBandOffsets[] = src.getBandOffsets();
            int srcPixelStride = src.getPixelStride();
            int srcScanlineStride = src.getScanlineStride();
```

**Listing A-2 Example OperationDescriptor (Sheet 8 of 8)**

```
            for (int k = 0; k < dnumBands; k++)  {
                int dstData[] = dstDataArrays[k];
                int srcData[] = srcDataArrays[k];
                int srcScanlineOffset = srcBandOffsets[k];
                int dstScanlineOffset = dstBandOffsets[k];
                for (int j = 0; j < dheight; j++)  {
                    int srcPixelOffset = srcScanlineOffset;
                    int dstPixelOffset = dstScanlineOffset;
                    for (int i = 0; i < dwidth; i++)  {
                        int pixel = srcData[srcPixelOffset];
                        if (pixel < param1) {
                           dstData[dstPixelOffset] = 0;
                        } else if (pixel > param2) {
                           dstData[dstPixelOffset] = 255;
                        } else {
                  dstData[dstPixelOffset] = srcData[srcPixelOffset];
                        }
                        srcPixelOffset += srcPixelStride;
                        dstPixelOffset += dstPixelStride;
                    }
                    srcScanlineOffset += srcScanlineStride;
                    dstScanlineOffset += dstScanlineStride;
                }
            }
        }
    }
```

APPENDIX B

# Java Advanced Imaging API Summary

THIS appendix summarizes the imaging interfaces and classes for Java AWT, Java 2D, and Java Advanced Imaging API.

## B.1 Java AWT Imaging

Table B-1 lists and describes the `java.awt` imaging classes.

**Table B-1 `java.awt` Imaging Classes**

| Class | Description |
|---|---|
| `Image` | Extends: `Object`<br>The superclass of all classes that represent graphical images. |

## B.2 Java 2D Imaging

The Java 2D API is a set of classes for advanced 2D graphics and imaging. It encompasses line art, text, and images in a single comprehensive model. The API provides extensive support for image compositing and alpha channel images, a set of classes to provide accurate color space definition and conversion, and a rich set of display-oriented imaging operators.

The Java 2D classes are provided as additions to the `java.awt` and `java.awt.image` packages (rather than as a separate package).

## B.2.1 Java 2D Imaging Interfaces

Table B-2 lists and briefly describes the imaging interfaces defined in the `java.awt.image` (Java 2D) API.

**Table B-2 `java.awt.image` Interfaces**

| Interface | Description |
|---|---|
| `BufferedImageOp` | Describes single-input/single-output operations performed on BufferedImage objects. This is implemented by such classes as `AffineTransformOp`, `ConvolveOp`, `BandCombineOp`, and `LookupOp`. |
| `ImageConsumer` | Used for objects expressing interest in image data through the `ImageProducer` interfaces. |
| `ImageObserver` | Receives notifications about `Image` information as the `Image` is constructed. |
| `ImageProducer` | Used for objects that can produce the image data for `Images`. Each image contains an `ImageProducer` that is used to reconstruct the image whenever it is needed, for example, when a new size of the Image is scaled, or when the width or height of the Image is being requested. |
| `ImagingLib` | Provides a hook to access platform-specific imaging code. |
| `RasterImageConsumer` | Extends: `ImageConsumer`<br>The interface for objects expressing interest in image data through the `ImageProducer` interfaces. When a consumer is added to an image producer, the producer delivers all of the data about the image using the method calls defined in this interface. |
| `RasterOp` | Describes single-input/single-output operations performed on `Raster` objects. This is implemented by such classes as `AffineTransformOp`, `ConvolveOp`, and `LookupOp`. |
| `RenderedImage` | A common interface for objects that contain or can produce image data in the form of `Rasters`. |
| `TileChangeListener` | An interface for objects that wish to be informed when tiles of a `WritableRenderedImage` become modifiable by some writer via a call to `getWritableTile`, and when they become unmodifiable via the last call to `releaseWritableTile`. |
| `WriteableRenderedImage` | Extends: `RenderedImage`<br>A common interface for objects that contain or can produce image data that can be modified and/or written over. |

## B.2.2 Java 2D Imaging Classes

Table B-3 lists and briefly describes the imaging classes defined in the `java.awt.image` (Java 2D) API.

**Table B-3 `java.awt.image` Classes**

| Class | Description |
|---|---|
| `AffineTransformOp` | Extends: `Object`<br>Implements: `BufferedImageOp`, `RasterOp`<br>An abstract class that uses an affine transform to perform a linear mapping from 2D coordinates in the source image or Raster to 2D coordinates in the destination image or Raster. |
| `AreaAveragingScaleFilter` | Extends: `ReplicateScaleFilter`<br>An `ImageFilter` class for scaling images using a simple area averaging algorithm that produces smoother results than the nearest-neighbor algorithm. |
| `BandCombineOp` | Extends: `Object`<br>Implements: `RasterOp`<br>Performs an arbitrary linear combination of bands in a Raster, using a specified matrix. |
| `BandedSampleModel` | Extends: `SampleModel`<br>Provides more efficent implementations for accessing image data than are provided in `SampleModel`. Used when working with images that store sample data for each band in a different bank of the DataBuffer. |
| `BilinearAffineTransformOp` | Extends: `AffineTransformOp`<br>Uses an affine transformation with bilinear interpolation to transform an image or Raster. |
| `BufferedImage` | Extends: `Image`<br>Implements: `WritableRenderedImage`<br>Describes an Image with an accessible buffer of image data. |
| `BufferedImageFilter` | Extends: `ImageFilter`<br>Implements: `RasterImageConsumer`, `Cloneable`<br>Provides a simple means of using a single-source/single-destination image operator (`BufferedImageOp`) to filter a `BufferedImage` or `Raster` in the Image Producer/Consumer/Observer paradigm. |

**Table B-3 `java.awt.image` Classes (Continued)**

| Class | Description |
|---|---|
| `ByteLookupTable` | Extends: `LookupTable`<br>Defines a lookup table object. The lookup table contains byte data for one or more tile channels or image components (for example, separate arrays for R, G, and B), and it contains an offset that will be subtracted from the input value before indexing the array. |
| `ColorConvertOp` | Extends: `Object`<br>Implements: `BufferedImageOp`, `RasterOp`<br>Performs a pixel-by-pixel color conversion of the data in the source image. The resulting color values are scaled to the precision of the destination image data type. |
| `ColorModel` | Extends: `Object`<br>Implements: `Transparency`<br>An abstract class that encapsulates the methods for translating from pixel values to color components (e.g., red, green, blue) for an image. |
| `ComponentColorModel` | Extends: `ColorModel`<br>A ColorModel class that can handle an arbitrary `ColorSpace` and an array of color components to match the `ColorSpace`. |
| `ComponentSampleModel` | Extends: `SampleModel`<br>Stores the N samples that make up a pixel in N separate data array elements all of which are in the same bank of a dataBuffer. |
| `ConvolveOp` | Extends: `Object`<br>Implements: `BufferedImageOp`, `RasterOp`<br>Implements a convolution from the source to the destination. Convolution using a convolution kernel is a spatial operation that computes the output pixel from an input pixel by multiplying the kernel with the surround of the input pixel. |
| `CropImageFilter` | Extends: `ImageFilter`<br>An `ImageFilter` class for cropping images. |
| `DataBuffer` | Extends: `Object`<br>Wraps one or more data arrays. Each data array in the `DataBuffer` is referred to as a bank. Accessor methods for getting and setting elements of the `DataBuffer`'s banks exist with and without a bank specifier. |
| `DataBufferByte` | Extends: `DataBuffer`<br>Stores data internally as bytes. |

**Table B-3 `java.awt.image` Classes (Continued)**

| Class | Description |
|---|---|
| `DataBufferInt` | Extends: `DataBuffer`<br>Stores data internally as ints. |
| `DataBufferShort` | Extends: `DataBuffer`<br>Stores data internally as shorts. |
| `DirectColorModel` | Extends: `PackedColorModel`<br>Represents pixel values that have RGB color components embedded directly in the bits of the pixel itself. |
| `FilteredImageSource` | Extends: `Object`<br>Implements: `ImageProducer`<br>An implementation of the `ImageProducer` interface which takes an existing image and a filter object and uses them to produce image data for a new filtered version of the original image. |
| `ImageFilter` | Extends: `Object`<br>Implements: `ImageConsumer`, `Cloneable`<br>Implements a filter for the set of interface methods that are used to deliver data from an `ImageProducer` to an `ImageConsumer`. |
| `IndexColorModel` | Extends: `ColorModel`<br>Represents pixel values that are indices into a fixed colormap in the `ColorModel`'s color space. |
| `Kernel` | Extends: `Object`<br>Defines a Kernel object – a matrix describing how a given pixel and its surrounding pixels affect the value of the given pixel in a filtering operation. |
| `LookupOp` | Extends: `Object`<br>Implements: `BufferedImageOp`, `RasterOp`<br>Implements a lookup operation from the source to the destination. |
| `LookupTable` | Extends: `Object`<br>Defines a lookup table object. The subclasses are `ByteLookupTable` and `ShortLookupTable`, which contain byte and short data, respectively. |
| `MemoryImageSource` | Extends: `Object`<br>Implements: `ImageProducer`<br>An implementation of the `ImageProducer` interface, which uses an array to produce pixel values for an Image. |

**Table B-3** **`java.awt.image` Classes (Continued)**

| Class | Description |
|---|---|
| `MultiPixelPackedSampleModel` | Extends: `SampleModel`<br>Stores one-banded images, but can pack multiple one-sample pixels into one data element. |
| `NearestNeighborAffine-TransformOp` | Extends: `AffineTransformOp`<br>Uses an affine transformation with nearest neighbor interpolation to transform an image or Raster. |
| `PackedColorModel` | Extends: `ColorModel`<br>An abstract ColorModel class that represents pixel values that have the color components embedded directly in the bits of an integer pixel. |
| `PixelGrabber` | Extends: `Object`<br>Implements: `ImageConsumer`<br>Implements an `ImageConsumer` which can be attached to an `Image` or `ImageProducer` object to retrieve a subset of the pixels in that image. |
| `RGBImageFilter` | Extends: `ImageFilter`<br>Provides an easy way to create an `ImageFilter` that modifies the pixels of an image in the default RGB ColorModel. It is meant to be used in conjunction with a `FilteredImageSource` object to produce filtered versions of existing images. |
| `Raster` | Extends: `Object`<br>Represents a rectanglular array of pixels and provides methods for retrieving image data. It contains a `DataBuffer` object that holds a buffer of image data in some format, a `SampleModel` that describes the format is capable of storing and retrieving Samples from the DataBuffer, and a `Rect` that defines the coordinate space of the raster (upper left corner, width and height). |
| `ReplicateScaleFilter` | Extends: `ImageFilter`<br>Scales images using the simplest algorithm. |
| `RescaleOp` | Extends: `Object`<br>Implements: `BufferedImageOp`, `RasterOp`<br>Performs a pixel-by-pixel rescaling of the data in the source image by multiplying each pixel value by a scale factor and then adding an offset. |

**Table B-3 `java.awt.image` Classes (Continued)**

| Class | Description |
|---|---|
| `SampleModel` | Extends: `Object`<br>Defines an interface for extracting samples of an image without knowing how the underlying data is stored in a DataBuffer. |
| `ShortLookupTable` | Extends: `LookupTable`<br>Defines a lookup table object. The lookup table contains short data for one or more tile channels or image components (for example, separate arrays for R, G, and B), and it contains an offset that will be subtracted from the input value before indexing the array. |
| `SinglePixelPackedSample-Model` | Extends: `SampleModel`<br>Stores (packs) the N samples that make up a single pixel in one data array element. All data array elements reside in the first bank of a DataBuffer. |
| `ThresholdOp` | Extends: `Object`<br>Implements: `BufferedImageOp`, `RasterOp`<br>Performs thresholding on the source image by mapping the value of each image component (for `BufferedImages`) or channel element (for `Rasters`) that falls between a low and a high value, to a constant. |
| `TileChangeMulticaster` | Extends: `Object`<br>A convenience class that takes care of the details of implementing the `TileChangeListener` interface. |
| `WritableRaster` | Extends: `Raster`<br>Provides methods for storing image data and inherits methods for retrieving image data from it's parent class `Raster`. |

## B.3 Java Advanced Imaging

The Java Advanced Imaging API consists of the following packages:

- `javax.media.jai` – contains the "core" JAI interfaces and classes
- `javax.media.jai.iterator` – contains special iterator interfaces and classes, which are useful for writing extension operations
- `javax.media.jai.operator` – contains classes that describe all of the image operators
- `javax.media.jai.widget` – contains interfaces and classes for creating simple image canvases and scrolling windows for image display

### B.3.1 JAI Interfaces

Table B-4 lists and briefly describes the interfaces defined in the Java Advanced Imaging API (`javax.media.jai`).

**Table B-4 Summary of jai Interfaces**

| Interface | Description |
|---|---|
| `CollectionImageFactory` | Abbreviated CIF, this interface is intended to be implemented by classes that wish to act as factories to produce different collection image operators. |
| `ImageFunction` | A common interface for vector-valued functions that are to be evaluated at positions in the X-Y coordinate system. |
| `ImageJAI` | The top-level JAI image type, implemented by all JAI image classes. |
| `OperationDescriptor` | Describes a family of implementations of a high-level operation (RIF) that are to be added to an `OperationRegistry`. |
| `PropertyGenerator` | An interface through which properties may be computed dynamically with respect to an environment of pre-existing properties. |
| `PropertySource` | Encapsulates the set of operations involved in identifying and reading properties. |
| `TileCache` | Implements a caching mechanism for image tiles. The TileCache is a central place for OpImages to cache tiles they have computed. The tile cache is created with a given capacity, measured in tiles. |
| `TileScheduler` | Implements a mechanism for scheduling tile calculation. |

## B.3.2 JAI Classes

Table B-5 lists and briefly describes the classes defined in the Java Advanced Imaging API (`javax.media.jai`).

**Table B-5 Summary of jai Classes**

| Class | Description |
|---|---|
| `AreaOpImage` | Extends: `OpImage`<br>An abstract base class for image operators that require only a fixed rectangular source region around a source pixel in order to compute each each destination pixel. |
| `BorderExtender` | An abstract superclass for classes that extend a `WritableRaster` with additional pixel data taken from a `PlanarImage`. |
| `BorderExtenderConstant` | Extends: `BorderExtender`<br>Implements border extension by filling all pixels outside of the image bounds with constant values. |
| `BorderExtenderCopy` | Extends: `BorderExtender`<br>Implements border extension by filling all pixels outside of the image bounds with copies of the edge pixels. |
| `BorderExtenderReflect` | Extends: `BorderExtender`<br>Implements border extension by filling all pixels outside of the image bounds with copies of the whole image. |
| `BorderExtenderWrap` | Extends: `BorderExtender`<br>Implements border extension by filling all pixels outside of the image bounds with copies of the whole image. |
| `BorderExtenderZero` | Extends: `BorderExtender`<br>Implements border extension by filling all pixels outside of the image bounds with zeros. |
| `CanvasJAI` | Extends: `java.awt.Canvas`<br>Automatically returns an instance of `GraphicsJAI` from its `getGraphics` method. |
| `CollectionImage` | Extends: `ImageJAI`<br>Implements: `java.util.Collection`<br>An abstract superclass for classes representing a collection of objects. |
| `CollectionOp` | Extends: `CollectionImage`<br>A node in a rendered imaging chain representing a `CollectionImage`. |

**Table B-5 Summary of jai Classes (Continued)**

| Class | Description |
|---|---|
| `ColorCube` | Extends: `LookupTableJAI`<br>Represents a color cube lookup table that provides a fixed, invertible mapping between tables indices and sample values. |
| `ComponentSampleModelJAI` | Extends: `ComponentSampleModel`<br>Represents image data that is stored such that each sample of a pixel occupies one data element of the `DataBuffer`. |
| `CoordinateImage` | Extends: `java.lang.Object`<br>Represents an image that is associated with a coordinate. This class is used with `ImageStack`. |
| `DataBufferDouble` | Extends: `java.awt.image.DataBuffer`<br>Stores `DataBuffer` data internally in double form. |
| `DataBufferFloat` | Extends: `java.awt.image.DataBuffer`<br>Stores `DataBuffer` data internally in float form. |
| `DisplayOpImage` | Extends: `OpImage`<br>A placeholder for display functionality. |
| `FloatDoubleColorModel` | Extends: `ComponentColorModel`<br>A `ColorModel` class that works with pixel values that represent color and alpha information as separate samples, using float or double elements, and that store each sample in a separate data element. |
| `GraphicsJAI` | Extends: `java.awt.Graphics2D`<br>An extension of `java.awt.Graphics` and `java.awt.Graphics2D` that will support new drawing operations. |
| `Histogram` | Extends: `java.lang.Object`<br>Accumulates histogram information on an image. A histogram counts the number of image samples whose values lie within a given range of values, or *bin*. |
| `ImageLayout` | Extends: `java.lang.Object`<br>Implements: `java.lang.Clonable`<br>Describes the desired layout of an `OpImage`. |
| `ImageMIPMap` | Extends: `ImageCollection`<br>Represents a stack of images with a fixed operational relationship between adjacent slices. |
| `ImagePyramid` | Extends: `ImageCollection`<br>Represents a stack of images with a fixed operational relationship between adjacent slices. |

**Table B-5 Summary of jai Classes (Continued)**

| Class | Description |
|---|---|
| `ImageSequence` | Extends: `ImageCollection`<br>Represents a sequence of images with associated timestamps and camera positions that can be used to represent video or time-lapse photography. |
| `ImageStack` | Extends: `ImageCollection`<br>Represents a group of images, each with a defined spatial orientation in a common coordinate system, such as CT scans or seismic volumes. |
| `IntegerSequence` | Extends: `java.lang.Object`<br>Represents an image that is associated with a coordinate. This class is used with `ImageStack`. |
| `Interpolation` | Extends: `java.lang.Object`<br>Encapsulates a particualr algorithm for performing sampling on a regular grid of pixels using a local neighborhood. It is intended to be used by operations that resample their sources, including affine mapping and warping. |
| `InterpolationBicubic` | Extends: `InterpolationTable`<br>Performs bicubic interpolation. |
| `InterpolationBicubic2` | Extends: `InterpolationTable`<br>Performs bicubic interpolation using a different polynomial than `InterpolationBicubic`. |
| `InterpolationBilinear` | Extends: `Interpolation`<br>Represents bilinear interpolation. |
| `InterpolationNearest` | Extends: `Interpolation`<br>Represents nearest-neighbor interpolation. |
| `InterpolationTable` | Extends: `Interpolation`<br>Represents nearest-neighbor interpolation. |
| JAI | Extends: `java.lang.Object`<br>A convenience class for instantiating operations. |
| `KernelJAI` | Extends: `java.lang.Object`<br>A convolution kernel, used by the `Convolve` operation. |
| `LookupTableJAI` | Extends: `java.lang.Object`<br>A lookup table object for the `Lookup` operation. |

**Table B-5 Summary of jai Classes (Continued)**

| Class | Description |
|---|---|
| `MultiResolutionRenderable-Image` | Extends: `java.lang.Object`<br>Implements: `java.awt.image.renderable, RenderableImage`<br>A `RenderableImage` that produces renderings based on a set of supplied `RenderedImages` at various resolution. |
| `NullOpImage` | Extends: `PointOpImage`<br>A trivial `OpImage` subclass that simply transmits its source unchanged. Potentially useful when an interface requires an `OpImage` but another sort of `RenderedImage` (such as a `TiledImage`) is to be used. |
| `OperationDescriptorImpl` | Extends: `java.lang.Object`<br>Implements: `OperationDescriptor`<br>A concrete implementation of the `OperationDescriptor` interface, suitable for subclassing. |
| `OperationRegistry` | Extends: `java.lang.Object`<br>Implements: `java.io.Externalizable`<br>Maps an operation name into a `RenderedImageFactory` capable of implementing the operation, given a specific set of sources and parameters. |
| `OpImage` | Extends: `PlanarImage`<br>The parent class for all imaging operations. `OpImage` centralizes a number of common functions, including connecting sources and sinks during construction of `OpImage` chains, and tile cache management. |
| `ParameterBlockJAI` | Extends: `java.awt.image.renderable, ParameterBlock`<br>A convenience subclass of `ParameterBlock` that allows the use of default parameter values and getting/setting parameters by name. |
| `PerspectiveTransform` | Extends: `java.lang.Object`<br>Implements: `java.lang.Cloneable, java.io.Serializable`<br>A 2D perspective (or projective) transform, used by various OpImages. |
| `PlanarImage` | Extends: `java.awt.Image`<br>Implements: `java.awt.image.RenderedImage`<br>A fundamental base class representing two-dimensional images. |

**Table B-5 Summary of jai Classes (Continued)**

| Class | Description |
|---|---|
| `PointOpImage` | Extends: `OpImage`<br>An abstract base class for image operators that require only a single source pixel to compute each destination pixel. |
| `PropertyGeneratorImpl` | Extends: `java.lang.Object`<br>A utility class to simplify the writing of property generators. |
| `RasterAccessor` | Extends: `java.lang.Object`<br>An adapter class for presenting image data in a `ComponentSampleModel` format, even if the data is not stored that way. |
| `RasterFactory` | A convenience class for the construction of various types of `WritableRaster` and `SampleModel` objects. |
| `RasterFormatTag` | Encapsulates some of the information needed for `RasterAccessor` to understand how a `Raster` is laid out. |
| `RemoteImage` | Extends: `PlanarImage`<br>An implementation of `RenderedImage` that uses a `RMIImage` as its source. |
| `RenderableGraphics` | Extends: `Graphics2D`<br>Implements: `RenderableImage`, `Serializable`<br>An implementation of `Graphics2D` with `RenderableImage` semantics. |
| `RenderableImageAdapter` | Extends: `java.lang.Object`<br>Implements: `java.awt.image.renderable.RenderableImage`, `PropertySource`<br>An adapter class for externally-generated `RenderableImages`. |
| `RenderableOp` | Extends: `java.awt.image.renderable.RenderableImageOp`<br>Implements: `PropertySource`<br>A JAI version of `RenderableImageOp`. |
| `RenderedImageAdapter` | Extends: `PlanarImage`<br>A `PlanarImage` wrapper for a non-writable `RenderedImage`. |
| `RenderedOp` | Extends: `PlanarImage`<br>A node in a rendered imaging chain. |
| `ROI` | Extends: `java.lang.Object`<br>Represents a region of interest of an image. |

**Table B-5 Summary of jai Classes (Continued)**

| Class | Description |
|---|---|
| `ROIShape` | Extends: `ROI`<br>Represents a region of interest within an image as a `Shape`. |
| `ScaleOpImage` | Extends: `WarpOpImage`<br>Used by further extension classes that perform scale-like operations and thus require rectilinear backwards mapping and padding by the resampling filter dimensions. |
| `SequentialImage` | Extends: `java.lang.Object`<br>Represents an image that is associated with a time stamp and a camera position. Used with ImageSequence. |
| `SnapshotImage` | Extends: `PlanarImage`:<br>Implements: java.awt.image.TileObserver<br>Provides an arbitrary number of synchronous views of a possibly changing `WritableRenderedImage`. |
| `SourcelessOpImage` | Extends: `OpImage`<br>An abstract base class for image operators that have no image sources. |
| `StatisticsOpImage` | Extends: `OpImage`<br>An abstract base class for image operators that compute statistics on a given region of an image and with a given sampling rate. |
| `TiledImage` | Extends: `PlanarImage`<br>Implements: `java.awt.image.WritableRenderedImage`<br>A concrete implementation of `WritableRenderedImage`. |
| `UntiledOpImage` | Extends: `OpImage`<br>A general class for single-source operations in which the values of all pixels in the source image contribute to the value of each pixel in the destination image. |
| `Warp` | Extends: `java.lang.Object`<br>A description of an image warp. |
| `WarpAffine` | Extends: `WarpPolynomial`<br>A description of an Affine warp. |
| `WarpCubic` | Extends: `WarpPolynomial`<br>A cubic-based description of an image warp. |
| `WarpGeneralPolynomial` | Extends: `WarpPolynomial`<br>A general polynomial-based description of an image warp. |

**Table B-5 Summary of jai Classes (Continued)**

| Class | Description |
|---|---|
| `WarpGrid` | Extends: `Warp`<br>A regular grid-based description of an image warp. |
| `WarpOpImage` | Extends: `OpImage`<br>A general implementation of image warping, and a superclass for other geometric image operations. |
| `WarpPerspective` | Extends: `Warp`<br>A description of a perspective (projective) warp. |
| `WarpPolynomial` | Extends: `Warp`<br>A polynomial-based description of an image warp. |
| `WarpQuadratic` | Extends: `WarpPolynomial`<br>A quadratic-based description of an image warp. |
| `WritableRenderedImage-Adapter` | Extends: `RenderedImageAdapter`<br>Implements: `java.awt.image.WritableRenderedImage`<br>A `PlanarImage` wrapper for a `WritableRenderedImage`. |

## B.3.3 JAI Iterator Interfaces

Table B-6 lists the JAI iterator classes (`javax.media.jai.iterator`).

**Table B-6 JAI Iterator Interfaces**

| Interface | Description |
|---|---|
| `RandomIter` | An iterator that allows random read-only access to any sample within its bounding rectangle. |
| `RectIter` | An iterator for traversing a read-only image in top-to-bottom, left-to-right order. |
| `RookIter` | An iterator for traversing a read-only image using arbitrary up-down and left-right moves. |
| `WritableRandomIter` | Extends: `RandomIter`<br>An iterator that allows random read/write access to any sample within its bounding rectangle. |
| `WritableRectIter` | Extends: `RectIter`<br>An iterator for traversing a read/write image in top-to-bottom, left-to-right order. |
| `WritableRookIter` | Extends: `RookIter`, `WritableRectIter`<br>An iterator for traversing a read/write image using arbitrary up-down and left-right moves. |

### B.3.4 JAI Iterator Classes

Table B-7 lists the JAI iterator classes (`javax.media.jai.iterator`).

**Table B-7 JAI Iterator Classes**

| Class | Description |
|---|---|
| `RandomIterFactory` | Extends: `java.lang.Object`<br>A factory class to instantiate instances of the `RandomIter` and `WritableRandomIter` interfaces on sources of type `Raster`, `RenderedImage`, and `WritableRenderedImage`. |
| `RectIterFactory` | Extends: `java.lang.Object`<br>A factory class to instantiate instances of the `RectIter` and `WritableRectIter` interfaces on sources of type `Raster`, `RenderedImage`, and `WritableRenderedImage`. |
| `RookIterFactory` | Extends: `java.lang.Object`<br>A factory class to instantiate instances of the `RookIter` and `WritableRookIter` interfaces on sources of type `Raster`, `RenderedImage`, and `WritableRenderedImage`. |

### B.3.5 JAI Operator Classes

Table B-8 lists the JAI operator classes (`javax.jai.operator`). These classes extend the JAI `OperationDescriptor` class.

**Table B-8 JAI Operator Classes**

| Class | Description |
|---|---|
| `AbsoluteDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `Absolute` operation, which gives the mathematical absolute value of the pixel values of a source image. |
| `AddCollectionDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `AddCollection` operation, which takes a collection of rendered images, and adds every set of pixels, one from each source image of the corresponding position and band. |
| `AddConstDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `AddConst` operation, which adds one of a set of constant values to the pixel values of a source image on a per-band basis. |

**Table B-8 JAI Operator Classes (Continued)**

| Class | Description |
|---|---|
| AddConstToCollection-Descriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the AddConstToCollection operation, which adds constants to a collection of rendered images. |
| AddDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Add operation, which adds the pixel values of two source images on a per-band basis. |
| AffineDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Affine operation, which performs an affine mapping between a source and a destination image. |
| AndConstDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the AndConst operation, which performs a bitwise logical AND between the pixel values of a source image with one of a set of per-band constants. |
| AndDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the And operation, which performs a bitwise logical AND between the pixel values of the two source images on a per-band basis. |
| AWTImageDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the AWTImage operation, which imports a standard AWT image into JAI. |
| BandCombineDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the BandCombine operation, which computes an arbitrary linear combination of the bands of a source image for each band of a destination image, using a specified matrix. |
| BandSelectDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the BandSelect operation, which copies the pixel data from a specified number of bands in a source image to a destination image in a specified order. |
| BMPDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the BMP operation, which reads BMP image data file from an input stream. |
| BorderDecriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Border operation, which adds a border around an image. |

**Table B-8 JAI Operator Classes (Continued)**

| Class | Description |
|---|---|
| `BoxFilterDescriptor` | Extends: `OperationDescriptorImpl` An OperationDescriptor for the `BoxFilter` operation, which determines the intensity of a pixel in an image by averaging the source pixels within a rectangular area around the pixel. |
| `ClampDescriptor` | Extends: `OperationDescriptorImpl` An OperationDescriptor for the `Clamp` operation, which sets all the pixel values below a "low" value to that low value, and sets all the pixel values above a "high" value to that high value. |
| `ColorConvertDescriptor` | Extends: `OperationDescriptorImpl` An OperationDescriptor for the `ColorConvert` operation, which performs a pixel-by-pixel color conversion of the data in a source image. |
| `CompositeDescriptor` | Extends: `OperationDescriptorImpl` An OperationDescriptor for the `Composite` operation, which combines two images based on their alpha values at each pixel. |
| `ConjugateDescriptor` | Extends: `OperationDescriptorImpl` An OperationDescriptor for the `Conjugate` operation, which negates the imaginary components of pixel values of an image containing complex data. |
| `ConstantDescriptor` | Extends: `OperationDescriptorImpl` An OperationDescriptor for the `Constant` operation, which defines a multi-banded, tiled rendered image with constant pixel values. |
| `ConvolveDescriptor` | Extends: `OperationDescriptorImpl` An OperationDescriptor for the `Convolve` operation, which computes each output sample by multiplying elements of a kernel with the samples surrounding a particular source sample. |
| `CropDescriptor` | Extends: `OperationDescriptorImpl` An OperationDescriptor for the `Crop` operation, which crops a rendered or renderable image to a specified rectangular area. |
| `DCTDescriptor` | Extends: `OperationDescriptorImpl` An operation descriptor for the `DCT` operation, which computes the even discrete cosine transform of an image. |
| `DFTDescriptor` | Extends: `OperationDescriptorImpl` An OperationDescriptor for the `DFT` operation, which computes the discrete Fourier transform of an image. |

**Table B-8 JAI Operator Classes (Continued)**

| Class | Description |
|---|---|
| `DivideByConstDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `DivideByConst` operation, which divides the pixel values of a source image by a constant. |
| `DivideComplexDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `DivideComplex` operation, which divides two images representing complex data. |
| `DivideDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `Divide` operation, which divides the pixel values of one first source image by the pixel values of another source image on a per-band basis. |
| `DivideIntoConstDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `DivideIntoConst` operation, which divides a constant by the pixel values of a source image. |
| `EncodeDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `Encode` operation, which stores an image to an `OutputStream`. |
| `ErrorDiffusionDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `ErrorDiffusion` operation, which performs color quantization by finding the nearest color to each pixel in a supplied color map. |
| `ExpDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `Exp` operation, which takes the exponential of the pixel values of an image. |
| `ExtremaDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `Extrema` operation, which scans an image and finds the image-wise maximum and minimum pixel values for each band. |
| `FileLoadDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `FileLoad` operation, which reads an image from a file. |
| `FileStoreDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `FileStore` operation, which stores an image to a file. |
| `FormatDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `Format` operation, which reformats an image. |

**Table B-8 JAI Operator Classes (Continued)**

| Class | Description |
|---|---|
| `FPXDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `FPX` operation, which reads FlashPix data from an input stream. |
| `GIFDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `GIF` operation, which reads GIF data from an input stream. |
| `GradientMagnitudeDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `Gradient` operation, which is an edge detector that computes the magnitude of the image gradient vector in two orthogonal directions. |
| `HistogramDecriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `Histogram` operation, which scans a specified region of an image and generates a histogram based on the pixel values within that region of the image. |
| `IDCTDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `IDCT` operation, which computes the inverse discrete cosine transform of an image. |
| `IDFTDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `IDFT` operation, which computes the inverse discrete Fourier transform of an image. |
| `IIPDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `IIP` operation, which reads an image from an IIP server and creates a RenderedImage or a RenderableImage based on data from the server. |
| `IIPResolutionDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `IIPResolution` operation, which reads an image at a particular resolution from an IIP server and creates a RenderedImage based on the data from the server. |
| `ImageFunctionDescriptor` | Extends: `OperationDescriptorImpl`<br>An OperationDescriptor for the `ImageFunction` operation, which generates an image on the basis of a functional description provided by an object that is an instance of a class that implements the `ImageFunction` interface. |
| `InvertDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `Invert` operation, which inverts the pixel values of an image. |

**Table B-8 JAI Operator Classes (Continued)**

| Class | Description |
|---|---|
| `JPEGDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `JPEG` operation, which reads a standard JPEG (JFIF) file. |
| `LogDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Log` operation, which takes the logarithm of the pixel values of an image. |
| `LookupDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Lookup` operation, which performs general table lookup on an image. |
| `MagnitudeDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Magnitude` operation, which computes the magnitude of each pixel of an image. |
| `MagnitudeSquaredDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `MagnitudeSquared` operation, which computes the squared magnitude of each pixel of a complex image. |
| `MatchCDFDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `MatchCDF` operation, which matches pixel values to a supplied cumulative distribution function (CDF). |
| `MaxDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Max` operation, which computes the pixelwise maximum value of two images. |
| `MeanDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Mean` operation, which scans a specified region of an image and computes the image-wise mean pixel value for each band within the region. |
| `MedianFilterDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `MedianFilter` operation, which is useful for removing isolated lines or pixels while preserving the overall appearance of an image. |
| `MinDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Min` operation, which computes the pixelwise minimum value of two images. |
| `MultiplyComplexDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `MultiplyComplex` operation, which multiplies two images representing complex data. |

**Table B-8 JAI Operator Classes (Continued)**

| Class | Description |
|---|---|
| `MultiplyConstDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `MultiplyConst` operation, which multiplies the pixel values of a source image with a constant on a per-band basis. |
| `MultiplyDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `Multiply` operation, which multiplies the pixel values of two source images on a per-band basis. |
| `NotDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `Multiply` operation, which performs a bitwise logical NOT operation on each pixel of a source image on a per-band basis. |
| `OrConstDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `OrConst` operation, which performs a bitwise logical OR between the pixel values of a source image with a constant on a per-band basis. |
| `OrderedDitherDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `OrderedDither` operation, which performs color quantization by finding the nearest color to each pixel in a supplied color cube and "shifting" the resulting index value by a pseudo-random amount determined by the values of a supplied dither mask. |
| `OrDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `Or` operation, which performs a bitwise logical OR between the pixel values of the two source images on a per-band basis. |
| `OverlayDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `Overlay` operation, which overlays one image on top of another image. |
| `PatternDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `Pattern` operation, which defines a tiled image consisting of a repeated pattern. |
| `PeriodicShiftDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `PeriodicShift` operation, which computes the periodic translation of an image. |
| `PhaseDescriptor` | Extends: `OperationDescriptorImpl`<br>An `OperationDescriptor` for the `Phase` operation, which computes the phase angle of each pixel of an image. |

**Table B-8 JAI Operator Classes (Continued)**

| Class | Description |
|---|---|
| PiecewiseDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Piecewise operation, which applies a piecewise pixel value mapping to an image. |
| PNGDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the PNG operation, which reads a PNG input stream. |
| PNMDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the PNM operation, which reads a standard PNM file, including PBM, PGM, and PPM images of both ASCII and raw formats. |
| PolarToComplexDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the PolarToComplex operation, which computes a complex image from a magnitude and a phase image. |
| RenderableDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Renderable operation, which produces a RenderableImage from a RenderedImage. |
| RescaleDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Rescale operation, which maps the pixel values of an image from one range to another range. |
| RotateDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Rotate operation, which rotates an image about a given point by a given angle. |
| ScaleDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Scale operation, which translates and resizes an image. |
| ShearDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Shear operation, which shears an image horizontally or vertically. |
| StreamDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the Stream operation, which reads java.io.InputStream files. |
| SubtractConstDescriptor | Extends: OperationDescriptorImpl<br>An OperationDescriptor for the SubtractConst operation, which subtracts one of a set of constant values from the pixel values of a source image on a per-band basis. |

**Table B-8 JAI Operator Classes (Continued)**

| Class | Description |
|---|---|
| `SubtractDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Subtract` operation, which subtracts the pixel values of the second source image from the first source image on a per-band basis. |
| `SubtractFromConstDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `SubtractFromConst` operation, which subtracts the pixel values of a source image from one of a set of constant values on a per-band basis. |
| `ThresholdDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Threshold` operation, which maps all the pixel values of an image that fall within a given range to one of a set of per-band constants. |
| `TIFFDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `TIFF` operation, which reads TIFF 6.0 data from an input stream. |
| `TranslateDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Translate` operation, which copies an image to a new location in the plane. |
| `TransposeDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Transpose` operation, which flips or rotates an image. |
| `URLDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `URL` operation, which reads an image from a file, via a URL path. |
| `WarpDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Warp` operation, which performs general warping on an image. |
| `XorConstDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `XorConst` operation, which performs a bitwise logical XOR between the pixel values of a source image with a constant. |
| `XorDescriptor` | Extends: `OperationDescriptorImpl` An `OperationDescriptor` for the `Xor` operation, which performs a bitwise logical XOR between the pixel values of two source images on a per-band basis. |

## B.3.6 JAI Widget Interfaces

Table B-9 lists the JAI widget interfaces (`javax.media.jai.widget`).

**Table B-9 JAI Widget Interfaces**

| Interface | Description |
|---|---|
| `ViewportListener` | Used by the `ScrollingImagePanel` class to inform listeners of the current viewable area of the image. |

## B.3.7 JAI Widget Classes

Table B-10 lists the JAI widget classes (`javax.media.jai.widget`).

**Table B-10 JAI Widget Classes**

| Class | Description |
|---|---|
| `ImageCanvas` | Extends: `java.awt.Canvas`<br>A simple output widget for a `RenderedImage`. This class can be used in any context that calls for a `Canvas`. |
| `ScrollingImagePanel` | Extends: `java.awt.Panel`<br>Implements: `java.awt.event.AdjustmentListener`, `java.awt.event.MouseListener`, `java.awt.event.MouseMotionListener`<br>An extension of `java.awt.Panel` that contains an `ImageCanvas` and vertical and horizontal scrollbars. |

# Glossary

THIS glossary contains descriptions of significant terms that appear in this book.

**affine transformation**
Geometric image transformation, such as translation, scaling, or rotation.

**band**
The set of all samples of one type in an image, such as all red samples or all green samples.

**box filter**
A low-pass spatial filter composed of uniformly-weighted convolution coefficients.

**bicubic interpolation**
Two-dimensional cubic interpolation of pixel values based on the 16 pixels in a $4 \times 4$ pixel neighborhood. See also *bilinear interpolation*, *nearest-neighbor interpolation*.

**bilinear interpolation**
Two-dimensional linear interpolation of pixel values based on the four pixels in a $2 \times 2$ pixel neighborhood. See also *bicubic interpolation*, *nearest-neighbor interpolation*.

**binary image**
An image that consists of only two brightness levels: black and white.

**chain code**
A pixel-by-pixel direction code that defines boundaries of objects within an image. The chain code records an object boundary as a series of direction codes.

**cobble**
To assemble multiple tile regions into a single contiguous region.

**color space conversion**
The conversion of a color using one space to another color space, such as RGB to CMYK.

**components**
Values of samples independent of color interpretation.

**compression ratio**
In image compression, the ratio of an uncompressed image data file size to its compressed counterpart.

**data element**
Primitive types used as units of storage of image data. Data elements are individual members of a `DataBuffer` array.

**directed graph (digraph)**
A graph with one-way edges. See also *directed acyclic graph (DAG)*.

**directed acyclic graph (DAG)**
A directed graph containing no cycles. This means that if there is a route from node A to node B then there is no way back.

**first-order interpolation**
See *bilinear interpolation*.

**high-pass filter**
A spatial filter that accentuates an image's high-frequency detail or attenuates the low-frequency detail. Contrast with *low-pass filter, median filter*.

**histogram**
A measure of the amplitude distribution of pixels within an image.

**Lempel-Ziv-Welch (LZW) compression**
A lossless image coding method that scans the image for repeating patterns of blocks of pixels and codes the patterns into a code list.

**low-pass filter**
A spatial filter that attenuates an image's high-frequency detail or accentuates the low-frequency detail. Contrast with *high-pass filter*, *median filter*.

**median filter**
A non-linear spatial filter used to remove noise spikes from an image.

**nearest-neighbor interpolation**
Two-dimensional interpolation of pixel values in which the amplitude of the interpolated sample is the amplitude of its nearest neighbor. See also *bicubic interpolation*, *bilinear interpolation*.

**perspective warp**
An image distortion in which objects appear trapezoidal due to foreshortening.

**projective warp**
See *perspective warp*.

**quantization**
The conversion of discrete image samples to digital quantities.

**ROI**
Abbreviation for *region of interest*. An area or pixel group within an image that has been selected for processing.

**run-length coding**
A type of lossless image data compression that scans for sequences of pixels with the same brightness level and codes them into a reduced description.

**Sobel edge detection**
A spatial edge detection filter that detects edges by finding the gradient of an image.

**square pixel**
A pixel with equal height and width.

**thresholding**
A point operation that maps all the pixel values of an image that fall within a given range to one of a set of per-band constants.

**transform coding**

A form of lossy block coding that transforms blocks of an image from the spatial domain to the frequency domain.

**trapping**

An image manipulation technique used in printing that uses dilation and erosion to compensation for misregistration of colors.

**unsharp masking**

An image enhancement technique using a high-frequency accentuating filter.

**zero-order interpolation**

See *nearest-neighbor interpolation*.

# Index