

JavaTM Advanced Imaging API

White Paper

Version 1.0
June 15, 1999



Java Software Division
Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

© 1998, 1999 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, California 94303 U.S.A.
All rights reserved.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The release described in this document may be protected by one or more U.S. patents, foreign patents, or pending applications.

Sun Microsystems, Inc. (SUN) hereby grants to you a fully paid, nonexclusive, nontransferable, perpetual, worldwide limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice this specification. This license allows and is limited to the creation and distribution of clean-room implementations of this specification that (i) are complete implementations of this specification, (ii) pass all test suites relating to this specification that are available from SUN, (iii) do not derive from SUN source code or binary materials, and (iv) do not include any SUN binary materials without an appropriate and separate license from SUN.

Sun, Sun Microsystems, the Sun logo, Java, JDK, JavaBeans, Java 3D, Write Once Run Anywhere, and VIS are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX® is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Contents

Contents	iii
Preface	v
1 Introduction	1
1.1 Design Goals	2
1.2 Concepts	3
1.3 Image Processing Operators	4
1.4 Advantages of the Java Advanced Imaging API	5
2 Java 2D Concepts	7
2.1 AWT Imaging	7
2.2 Java 2D	8
2.3 Rendering Independence	9
2.4 Rendering-Independent Imaging in the Java2D API	10
2.5 The Renderable Layer vs. the Rendered Layer	10
2.6 The Render Context	12
2.7 Renderable and Rendered Classes	12
2.7.1 The Renderable Layer	12
2.7.2 Using the Rendered Layer	13
2.8 Image Data Representation	15
3 Java Advanced Imaging API Concepts	17
3.1 Java Advanced Imaging API versus Java 2D API and the AWT	17
3.1.1 Similarities with the Java 2D API	18
3.2 Processing Graphs	18
3.2.1 Renderable Graphs	19
3.2.2 Rendered Graphs	20
3.3 The Tile Cache	21
3.4 The Extension Mechanism	21
3.5 Image Collections	22
3.6 Iterators	22
3.7 Deferred Execution	23
3.8 Client/server Imaging	23
3.9 Properties	24
3.10 Regions of Interest	25

3.11 Programming Example..... 26

Preface

THIS White Paper is a brief overview of the features of the Java Advanced Imaging API; it describes the fundamental concepts of the API and its architecture.

This document is not a complete specification of the Java Advanced Imaging API; many subsidiary methods and convenience methods are not described and some details in the code examples have been omitted for clarity. Similarly, depictions of the various object hierarchies have been simplified for illustrative purposes.

Intended Audience

It is assumed in this document that the reader is relatively sophisticated concerning the principles and practices of object-oriented programming and has a good working knowledge of image processing and the requirements of image processing applications in some problem domain. Knowledge of the Java programming language and the core Java Development Environment classes (in particular the **java.awt** and **java.awt.image** packages) is also assumed, although a brief overview of these topics is given at the beginning of Chapter 2.

Related Documentation

In addition to this document, further information about Java Advanced Imaging API and related class libraries can be found in the following documents. Most of these documents are available on-line in HTML format; for other documents, please consult the appropriate specification group.

- *Java Advanced Imaging API Specification*
URL: <http://java.sun.com/products/java-media/jai>
- *Programming in Java Advanced Imaging*
URL: <http://java.sun.com/products/java-media/jai>

- *The Java 2D API Specification and Java 2D White Paper*
URL: <http://java.sun.com/products/java-media/2D>
- *Java 3D™ API Specification*
URL: <http://java.sun.com/products/java-media/3D>

A special interest mailing list is available for the Java Advanced Imaging API; to learn how to subscribe to this list, please visit the Java Advanced Imaging web site given above.

Style Conventions

The following style conventions are used in this specification:

- `Lucida Sans` type is used to represent computer code, the names of files and directories, class names and method names.
- **Bold Lucida Sans type** is used for Java Advanced Imaging API API declarations.
- **Bold** type is used to represent variables.
- *Italic type* is used for emphasis and for equations.

Document Overview

This document describes the architecture and programming interface of the Java Advanced Imaging API.

- Chapter 1 gives an overview of the Java Advanced Imaging API design goals and concepts
- Chapter 2 discusses previous support for imaging in the Java Development Environment, particularly in the Java 2D API
- Chapter 3 describes the architecture of the Java Advanced Imaging API and its fundamental concepts.

Release Notes for the Version 1.0 White Paper

This White Paper corresponds to Version 1.0 of the Java Advanced Imaging API specification and version 1.2 of the Java Developer's Kit (JDK).

Acknowledgments

The principal partners in the development of the Java Advanced Imaging API specification are

Autometric, Inc.

Eastman Kodak, Inc.

Siemens AG

Sun Microsystems, Inc.

The partners gratefully acknowledge the substantial comments provided to us by reviewers of the specification.

Introduction

THE Java Advanced Imaging API supports advanced digital image processing. This API extends the reach of the Java platform to allow sophisticated, high-performance image processing to be incorporated into Java applets and applications. The API is designed to be

- **Cross-platform:** Implementations will run on virtually any computer where there is a Java Virtual Machine.
- **Object-oriented:** Images and image processing operations are defined as objects. The connections between these objects define the flow of processed data.
- **Highly extensible:** Nearly arbitrary image processing operations can be added to the API in such a way that they appear to be a native part of it.
- **Device-independent:** Processing of images can be specified in device-independent coordinates, with the ultimate translation to pixels being specified as needed at run time.
- **Powerful:** Complex image data formats are supported, including images with an arbitrary number of bands. Many classes of imaging algorithms are supported directly.
- **High-performance:** The API is designed so that a variety of implementations are possible, including highly-optimized implementations that can take advantage of hardware acceleration.
- **Interoperable:** The API works well with other Java APIs such as Java 3D and Java component technologies, allowing sophisticated imaging to be a part of every Java technology programmer's tool box.

The Java Advanced Imaging API is a Java Media API. It is classified as a Standard Extension to the Java platform. The API provides imaging functionality beyond that of the Java Foundation Classes, although it is designed to provide

compatibility with those classes in most cases. The API implements a set of core image processing capabilities, including image tiling, regions of interest, and deferred execution and a set of core image processing operators, including many common point, area, and frequency domain operations. It is intended to meet the needs of technical imaging (medical, seismological, remote sensing, etc.) as well as commercial imaging (such as document production and photography). It is not, however, a high-level toolkit for building imaging applications, nor is it a general image database manipulation library; it is a mid-level foundation API that implements image processing operations on images in a powerful, general way.

1.1 Design Goals

The Java Advanced Imaging API 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, it seeks to present 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. 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 API design.

The primary goals of the designers of the API were to:

- Provide a rich set of functionality for digital imaging through a Java API.
- Have the processing model used be simple and easily understood by the application writer.
- Allow programmers to extend the functionality of the API to include almost arbitrary processing capabilities.
- Minimize the complexity of the extension process so that programmers will be encouraged to write extensions rather than manipulating image data directly.
- Support a wide variety of image formats, data types, image file types, and image presentation devices.
- Accommodate a variety of programming styles including an immediate mode and a deferred mode of execution for different types of imaging applications.
- Permit the creation of truly distributed imaging applications.

- Preserve sufficient flexibility to allow for multiple implementations of the specification with different tradeoffs of memory usage, operator optimization, and hardware acceleration.
- Place as few restrictions as possible on the set of applications that can be effectively implemented.
- Take advantage of the standard Java Runtime Environment functionality (such as garbage collection, memory management, and component technology), rather than reimplementing these capabilities.

While some of these design goals may appear at first glance to be mutually contradictory, modern object-oriented design paradigms for the encapsulation of data and algorithms and the appearance of true component technologies and distributed computing mechanisms have placed them all within reach. For example, the encapsulation of image data formats and remote method invocations within a re-usable image data object allows an image file, a network image object, or a real-time data stream to be processed identically, regardless of its true location or method of production. Thus, a simple programming model can be presented to the application programmer, while the complexity of the internal mechanism is concealed.

Similarly, the mechanism for adding functionality to the API can be presented at multiple levels of encapsulation and complexity, allowing 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.

1.2 Concepts

To accomplish its design goals, the Java Advanced Imaging API supports a number of concepts:

- Arbitrary extensions, allowing programmers to extend the API by implementing arbitrary image processing algorithms and having their extension appear to be a standard part of the API.
- Rendering-independent sources and operations, so that processing operations can be specified in resolution-independent coordinates.
- Rendering contexts, that allow the deferral of the definition of some image and operation characteristics until execution.
- Editable graphs of image processing operations to be defined and instantiated as needed.

- Multi-dimensional, multi-banded image data types.
- Many common image data types (grey scale, true color, pseudocolor, and multiple color coordinate spaces) with a wide variety of pixel intensity resolutions.
- Tiled images, allowing large images to be processed effectively.
- A deferred execution model that processes only the pixels that are actually needed by an application.
- Operation chain optimization, allowing a chain of image processing operations to be replaced with a more highly optimized implementation.

It will be noted that these concepts are independent of the actual processing operations that are implemented by the API. Virtually *any* image processing algorithm can be added to the API and used as if it were a native part of the API. The API provides a very general framework for organizing and encapsulating the actual processing to be done. This is essential, since any standard set of operations that might be implemented in an image processing API cannot hope to capture the enormous variety of operations that can be performed on a digital image. Extensibility is the *sine qua non* of the the Java Advanced Imaging API.

1.3 Image Processing Operators

Nevertheless, a core set of image processing operators is supported that must be present in every implementation of the specification. This provides 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 operators supported include, among others:

- Common arithmetic point operations
- Intensity remapping operations
- Arbitrary affine image transformations
- Polynomial and table-driven geometric image warping
- Area operations in the spatial domain, such as convolution and other kernel-based linear filtering
- Image statistics operators, such as histogramming
- Fourier transforms and frequency-domain filtering

Further, abstractions for many common types of image collections are supported, such as time-sequential data and image pyramids. These are intended to simplify operations on image collections and allow the development of operators which

work directly on these abstractions. The API also supports multiple image coordinate systems, so that operators can be written which work directly in a particular coordinate system.

1.4 Advantages of the Java Advanced Imaging API

The Java Advanced Imaging API offers a number of advantages for application developers compared to other imaging solutions:

- It is a true cross-platform imaging API, providing a standard interface to the imaging capabilities of a platform.
- It uses the Java language's object-oriented paradigm, reducing development costs.
- It is well suited for client-server imaging because of the Java platform's networking architecture and the availability of complementary Java technologies such as Java RMI API and JavaBeans
- It has a flexible architecture allowing the deployment of computing systems based on the cost/performance needs of a system, ranging from thin-client/server models to power-client models.
- It provides an extensible framework which allows customized solutions for vertical markets to be provided within the standard framework.
- It is integrated with the rest of the Java Media APIs, enabling media-rich applications to be deployed on the Java platform.
- It is designed so that highly-optimized implementations using the media-capabilities of CPUs, such as MMX on Intel processors and VIS™ on UltraSparc, can be readily developed

Java 2D Concepts

DIGITAL imaging on the Java platform 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 document. In this chapter, we review briefly the imaging portions of AWT and examine in some detail the imaging features of the Java 2D API, which is new in the Java 2 Development Environment and is part of the Java Foundation Classes. The Java Advanced Imaging API uses and extends the features of the Java 2D API as the foundation for its own imaging classes. An understanding of the imaging architecture of the Java 2D API is essential to understand the Java Advanced Imaging API architecture; therefore, we discuss the Java 2D API at some length.

2.1 AWT Imaging

Briefly, AWT Imaging presents 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 AWT 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 be 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.

2.2 Java 2D

To alleviate some of the restrictions of the original AWT imaging model and to provide a higher level of abstraction, a specification called the *Java 2D API* was developed for the 1.2 version of the JDK that extends AWT's capabilities for both two-dimensional graphics and imaging. In practice, the Java 2D API package is contained in the Java Foundation Classes and is a part of the Java core technology (and thus available in all Java platform implementations). However,

in this document, the distinction between the Java 2D API and the AWT is preserved for convenience of reference.

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 quite 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 concept of rendered and renderable images contained in the Java 2D API is essential to the Java Advanced Imaging API. The next few sections explain these concepts; complete information about the classes discussed can be found in the *The Java 2D API Specification*.

2.3 Rendering Independence

Resolution independence for images is a poorly understood topic because it is poorly named. The more general problem is “rendering 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, the output device type, the color quality, the tonal quality, and the rendering speed. A rendering-independent description is concerned with none of these.

In this document, the term *rendering-independent* will be used for the more general concept instead of *resolution-independent*. The latter term will be used specifically to 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 which is capable of infinite zooming. The characteristics of this image are that it is two dimensional, has a native aspect

ratio (that of the capture device), and may have properties which 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.

These two together 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. This is the primary power of rendering independence: the same visual description can be routed to any display context with an optimal result.

2.4 Rendering-Independent Imaging in the Java2D API

The Java 2D API architecture integrates a model for 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 2D API architecture supports context-dependent adaptation, which is superior to full image production and processing. It 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 2D API architecture is essentially synchronous in nature. This has several advantages, such as a simplified programming model, and explicit controls on the type and order of results. However, it has one distinct disadvantage in that it is not well-suited to notions of progressive rendering or network sources. (These issues are addressed in the Java Advanced Imaging API architecture.)

2.5 The Renderable Layer vs. the Rendered Layer

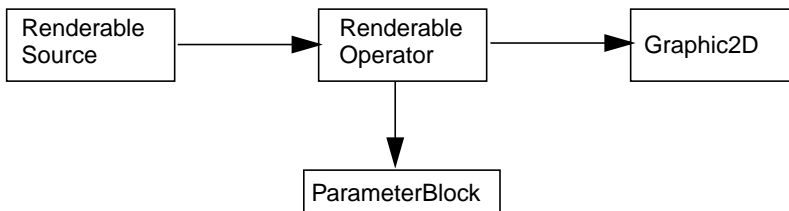
The Java 2D API architecture provides for two integrated imaging layers. There is a rendering-independent layer, called the *Renderable layer* (the interfaces and classes have `Renderable` in their names). It provides image sources which can be optimally reused multiple times in different contexts, such as screen display or printing. It also provides imaging operators which 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 (*e.g.*, to a display or file) is requested. That is, a request is made at the

sink end of the chain which is passed up the chain to the source. Such requests are context-specific (e.g., device-specific), and the chain adapts to the context. Only the data required for the context is produced.

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 which 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.

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) which 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 A Renderable chain. The chain has a sink attached (a *Graphics2D* object), but no pixels flow through the chain yet.



Programmers may use either the *Renderable* or *Rendered layer* to construct their applications. Many application 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*.

2.6 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 presents the source as modified by these operations; the end of the 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*; they 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 `Rendered` 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.

2.7 Renderable and Rendered Classes

2.7.1 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, and 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 which 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` (CRIF). 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.

The other object involved in the construction of a `RenderableImageOp` is a `ParameterBlock`. This houses references to the source(s) for the operation, plus parameters or other objects which the operator may require. The parameters are rendering-independent versions of the parameters which control the corresponding `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 `createRendering` 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.

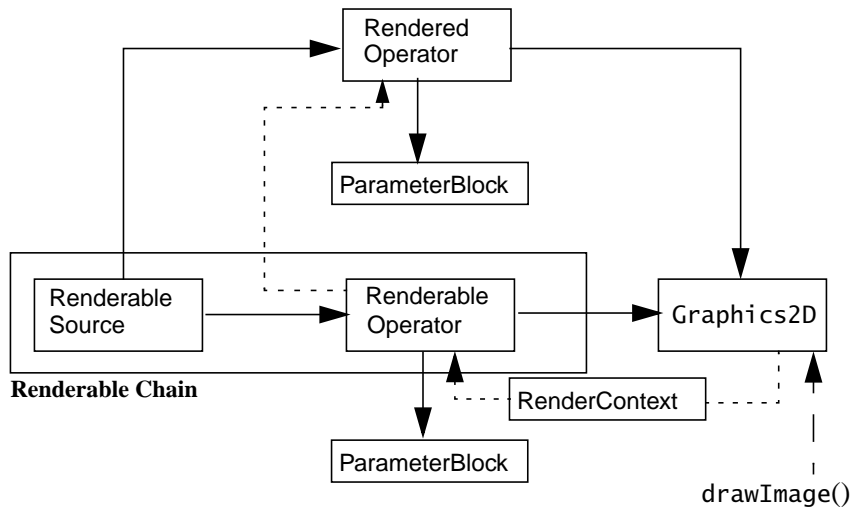
2.7.2 Using the Rendered Layer

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 can be used directly 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 below. Details concerning how the `Rendered` layer functions internally within the `Renderable` layer are covered there.

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` which 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.

Figure 2-2 Deriving a rendering from a renderable chain. When the user calls `Graphics2D.drawImage()`, a render context is constructed and used to call the `createRendering()` 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 `createRendering()` method is called.



A `Rendered` image represents a virtual image with a coordinate system that maps directly to pixels. It is not required that a `Rendered` image 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, maintain a full page buffer which can be accessed and written to. Data can be accessed in a variety of ways, each with different properties.

2.8 Image Data Representation

In the Java 2D 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, the red sample, the blue sample, and the green sample. 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.

The basic unit of image data storage is the `DataBuffer`. It is a kind of raw storage; it contains all the samples for the image data but does not maintain a notion of how those samples can be put together as pixels. This interpretation is housed in a `SampleModel`. Hence, the `SampleModel` class contains methods for deriving pixel data from a `DataBuffer`. The two together, a `DataBuffer` plus a `SampleModel`, constitute a meaningful multi-pixel image storage unit called a `Raster`. A `Raster` therefore has methods which directly return pixel data for the image data it contains. Note that there are two basic kinds of `Rasters`: `Raster` itself, which is intended to be read-only and only has “get” methods, and `WritableRaster` which can be written to and thus has a variety of “set” methods. There are separate interfaces for dealing with each raster type. The `RenderedImage` interface assumes the data is read-only, and hence does not contain methods for writing a `Raster`, while `WritableRenderedImage` assumes that the image data can be modified.

A *tile* is not a class in the architecture. Rather, it is a concept. A tile is one of a set of regular rectangular regions which span the image on a regular grid; data from a tile is returned in a `Raster` object. In the `RenderedImage` interface, many methods exist which relate to tiles and a tile grid (these methods are used by the Java Advanced Imaging 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. Thus, the `getWritableTile` method will return all the image data. Other methods which 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` 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.

Java Advanced Imaging API Concepts

THIS chapter describes the overall architecture of the Java Advanced Imaging API; the fundamental differences between it and the Java 2D API are outlined, and the concept of image graph building is described. The API embodies a few primary abstractions on which the entire library is based; the next few sections give an overview of the processing model and the main abstractions used.

3.1 Java Advanced Imaging API versus Java 2D API and the AWT

The Java Advanced Imaging API builds upon the foundation of the Java 2D API to allow more powerful and general imaging applications; it adds concepts such as multi-tiled images, deferred execution, networked images, and image property management. Unlike Java 2D, image operators with multiple sources are supported.

In the Java Advanced Imaging API, 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 may participate as first-class objects in the deferred execution model.

The API also provides for a considerable degree of compatibility with the AWT and Java 2D imaging models; its operators can work directly on Java 2D `BufferedImage` objects, or any other image objects that implement the `RenderedImage` interface. The same rendering-independent model as the Java 2D API is supported, using device-independent coordinates. Java 2D-style drawing

using the Graphics interface is supported on both Rendered and Renderable images.

The Java Advanced Imaging API does not make use of the image producer/consumer interfaces introduced in AWT and carried forward into the Java 2D API; instead, it requires that image sources participate in the “pull” imaging model by responding to requests for arbitrary areas, thus making it impossible to use an ImageProducer directly as a source. AWT Images, however, can be imported as Java Advanced Imaging objects.

3.1.1 Similarities with the Java 2D API

The Java Advanced Imaging 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 without major modifications. Here we list some of the major points of congruity:

- The `RenderableImage` and `RenderedImage` interfaces defined in the Java 2D API are used as a basis for higher-level abstractions. Further, directed acyclic graphs of objects implementing these interfaces can be created and manipulated.
- 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.
- 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 beyond that in the Java 2D API. A registry mechanism is introduced to automate the selection of operations on `RenderedImages`.
- The `SampleModel`, `DataBuffer`, and `Raster` objects from the Java 2D API are carried over without change, except that `double`s and `float`s are allowed to be used as the fundamental data types of a `DataBuffer` in addition to the `byte`, `short`, and `int` data types.

3.2 Processing Graphs

The Java Advanced Imaging API, as an object-oriented API, unifies the notions of image and operator by making both subclasses of a common parent. An operator object is instantiated with a set of image sources and other parameters,

and is itself considered to be an image. Thus, the operator object may be used in turn as an image source for further operations. This notion naturally leads to the idea of an imaging *graph* (or *chain*), since the action of attaching new objects to existing ones automatically produces directed acyclic graphs (DAGs) where each object is a node in the graph, and object references form the edges.

Although it is possible to leave the DAG structure of images and operators implicit, as in many APIs, the Java Advanced Imaging API 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, we may also directly consider the graph structure produced by the operations and the interconnections between them.

The API does place some restrictions on the graphs that may be built. When creating graphs by instantiating new nodes one at a time, cycles are not possible since a new node can only have older ones as sources. When reconfiguring graphs, however, it is the user's responsibility to avoid the creation of cycles.

3.2.1 Renderable Graphs

A *Renderable graph* is composed of nodes implementing the `RenderableImage` interface. In the Java Advanced Imaging API, these nodes will usually be instances of the `RenderableOp` class. Renderable graphs, like Renderable chains in the Java 2D API, are not evaluated at the time of their specification; rather, evaluation is deferred until there is a specific request for a rendering. In particular, if source images change between renderings the changes will be reflected in the output.

As the Renderable DAG is constructed, the sources of each node must be specified to form the topology of the graph. The sources may be arbitrarily respecified, as long as cycles are not introduced. A Renderable graph is both editable and reusable. After a request is made to a Renderable graph to render itself, the sources and parameters may be respecified and the graph rendered again.

The ultimate sources of a Renderable graph must be Renderable image objects. A rendering of a Renderable graph is produced by calling the `createRendering` method on any node of the Renderable graph. This call takes a `RenderContext` that is passed recursively to all the objects above the specified Renderable in the graph, just as in the Java 2D chain. The result is an object implementing the `RenderedImage` interface.

As discussed previously, it is easiest to think of a *Renderable* graph as a *template* that, when rendered, causes the instantiation of a parallel *Rendered* graph to accomplish the actual processing. The *Rendered* graph that is “cloned” from the *Renderable* graph is immutable and captures the precise state of the *Renderable* graph at the time of the rendering. Note that the *Rendered* graph need not actually produce pixel output until requested to do so. The *Renderable* graph remains available and may be reinstantiated with the same or different sources and parameters. A *Renderable* graph may also be serialized for future use or for transmission over a network.

3.2.2 **Rendered Graphs**

A *Rendered* graph is one composed of *Rendered object* nodes. In the Java Advanced Imaging API, these nodes will usually be instances of the *RenderedOp* class, but could belong to any subclass of *PlanarImage*, the API’s version of *RenderedImage*. *Rendered* graphs perform processing in immediate mode. That is, the image sources of a node of the graph are considered to have been evaluated at the moment it is instantiated and added to the graph. As a result, an operation node in the *Rendered* graph, once defined, will always produce the same output pixels regardless of changes to its sources.

It is possible to reconfigure and reuse *Rendered* graphs within certain limitations. Initially, a node in a *Rendered* graph is mutable; it may be assigned new sources, which are considered to be evaluated as of the moment of assignment, and its parameter values may be altered. However, once actual rendering takes place at a node it becomes *frozen* and its sources and parameters can no longer be changed. Such rendering takes place whenever pixel data or size information are extracted from the node.

A chain of *Rendered* nodes may be cloned, without freezing any of its nodes, by means of the *createInstance* method. By taking care to use only *createInstance* to perform renderings, a *Rendered* graph may be reconfigured and reused at will as well as serialized and transmitted over networks.

Since a *Renderable* graph produces a *Rendered* graph when *createRendering* is called, *Rendered* objects are the only ones that actually possess pixel data or perform any computation. When a programmer adds a new extension, a new *Rendered* object is created to perform the computation. The process of writing extensions is actually the process of writing new *Rendered* objects.

3.3 The Tile Cache

While it may seem that the creation of a new `Rendered` chain upon every invocation of `createRendering` would be an expensive process, this is not necessarily true. The API provides an efficient mechanism for reusing previously computed tiles. The `TileCache` class implements a global pool for caching recently computed tiles. If a needed tile in a `Rendered` chain is in the cache, it is not necessary to repeat previously performed tile renderings.

While the API specifies the interface to the `TileCache` object, it leaves the implementation of the API free to select efficient mechanisms for managing the tile cache.

3.4 The Extension Mechanism

Imaging operations are ultimately performed by a series of objects that subclass `OpImage`. Extenders write new `OpImage` subclasses, frequently by subclassing existing utility classes that automate some of the details of the particular type of operator being implemented. For most operators, extenders need only supply a routine that is capable of producing an arbitrary rectangle of output, given contiguous source data.

New operators may be made available to users transparently and without user source code changes using a registry mechanism. Existing applications may be tuned for new hardware platforms by strategic insertion of new implementations of existing operators. It is the API's goal to encourage all new image-to-image operators to be implemented using this mechanism, in order to provide the benefits of tiling, deferred execution, and memory management with the minimum amount of difficulty for the implementor.

Another important facet of extensibility is the ability to manage non-image data in imaging chains. This capability is provided by means of an extensible property mechanism. Image nodes may be queried for the values of various named properties, which may either be synthesized by the node itself or computed based on properties of the node's sources. By default, nodes simply copy their source properties. This mechanism allows data from tagged image file formats, such as TIFF, to be propagated through a chain and modified as needed. For example, a property indicating the spatial position of a pixel might be modified by an operation that rotates the image. Users may add their own code to the property inheritance mechanism dynamically.

3.5 Image Collections

`CollectionImage` is the abstract superclass for four classes representing collections of `PlanarImages`: `ImageStack`, `ImageMIPMap`, `ImageSequence`, and `ImagePyramid`, and one operator superclass, `CollectionOp`, which allows processing operations to be performed on image collections. `ImageStack` represents a set of two-dimensional images lying in a common three-dimensional space. The images need not lie parallel to one another. `ImageSequence` represents a sequence of images with associated times and camera positions. `ImageMIPMap` represents a stack of images with a fixed operational relationship between adjacent image slices. Given the highest resolution image slice, the lower resolution image slices may be derived in turn by performing a particular operation specified by a `RenderedOp`. Images may be extracted slice by slice or by special iterators. Finally, `ImagePyramid` is similar to `ImageMIPMap`, but the derivation operation is bidirectional, allowing images at both higher and lower resolutions to be extracted from the pyramid.

3.6 Iterators

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`. It 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. Tile boundaries in the source images are not visible to the user of the iterator. The iterator is also responsible for dealing with all of the possible combinations of `SampleModel` and `DataBuffer`.

Three styles of iterator are provided, and extenders are allowed to build others. 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, allowing leftward and upward movement and backwards motion through the set of bands. Both styles allow jumping to an arbitrary line or pixel, and reading and writing of a random band of the current pixel. It is also possible to jump back to the first line or pixel, and to the last line or pixel in `RookIter`.

`RandomIter` allows an unrelated set of samples to be read by specifying their *x* and *y* coordinates and band offset. This will generally be slower than either a `RectIter` or a `RookIter`, but remains useful for its ability to hide pixel formats and tile boundaries.

Iterators may be used both in the implementation of the `computeRect` methods or `getTile` methods of `OpImage` subclasses, and for ad-hoc, pixel-by-pixel image manipulation.

The actual implementation of iterators contains a multitude of specialized classes. However, users need only understand the `RectIter`, `RookIter`, and `RandomIter` interfaces for reading as well as their `Writable` counterparts. Iterators are created by means of `create` and `createWritable` factory methods that examine the source image and select the proper underlying implementation. The underlying implementation classes are designed to do a minimal amount of work per pixel, and to be inlineable by a sophisticated just-in-time compiler.

3.7 Deferred Execution

A `Rendered` chain has the property of seemingly immediate evaluation of sources. Beneath the hood, of course, an implementation will attempt to avoid performing work until it is absolutely necessary. A mechanism is provided to do this in the form of the `SnapshotImage` class. This class may be used to provide a synchronous view of source images that are liable to change. By implementing the `TileObserver` interface from Java 2D API, the `SnapshotImage` is informed when any tile is about to become writable, so that it can make a copy if necessary. The `SnapshotImage` also keeps track of what objects use it as a source, and at what times they requested a snapshot. Thus the `SnapshotImage` is able to store exactly the tiles necessary to provide an illusion of stability to all of its consumers.

When a `WritableRenderedImage` of any kind, such as a `TiledImage`, is used as a source for a `RenderedOp` or `OpImage`, a snapshot is taken. `RenderedImages` that do not implement `WritableRenderedImage` are immutable by definition, and it suffices to simply maintain a reference to them to prevent them from being garbage collected. Snapshotting images that are writable but that do not change will incur only the small penalty of maintaining a `SnapshotImage` with no stored tiles.

3.8 Client/server Imaging

A useful capability in an imaging system is the ability to distribute computation between a set of processing nodes. A common case involves a powerful server providing image processing services to a large number of thin clients. With the Java Advanced Imaging API, it is possible for a client to set up a complex imaging chain on a server, including references to source data situated on other

network hosts, and to request Rendered output from the server. Many other scenarios are possible as well, but for the purposes of this section we will refer to this simple client/server model.

The API can use any number of protocols for communicating between the client and the server. A single class, `RemoteImage`, implements the necessary methods for communication; subclasses of `RemoteImage` may use virtually any protocol for transmitting and receiving image data. The reference implementation uses the Java Remote Method Invocation (RMI) interfaces for this communication.

In practice, an application contacts the server and requests it to create a `RemoteImage` object from a given `RenderedImage`. The application may then pull pixels from the `RemoteImage` without consideration of where it is located on the network.

3.9 Properties

In addition to pixels, imaging applications often wish to manage many other kinds of data associated with images. The Java Advanced Imaging API provides a mechanism for attaching arbitrary data to images in the form of *properties*. Every image maintains a simple database of properties, where each property is simply an `Object` that is referenced by a unique name.

The presence of a property on a node does not normally affect pixel processing. For example, suppose an extender invents a new `PixelShape` property that can describe exotic layouts such as hexagonal grids. The value of the property will be some user-defined object that is capable of describing the various possible layouts. All of the standard operators assume that pixels are square, and are laid out in standard fashion. Such operators will continue to perform their standard functions whether or not the `PixelShape` property is set, since they are familiar with neither the property name nor the object it returns. However, an extender could write a `reshapePixels` operator (that is, a RIF) that examines the properties of its source image; if no `PixelShape` property is present, it assumes the pixels are square. If the property is present, it queries it and makes use of the associated information. The RIF instantiates a suitable `OpImage` that performs resampling to account for the difference in the source and destination pixel geometries. The extender may also extend the property mechanism so the `PixelShape` property on the `reshapePixels` operation node will have the new value appropriate to the resampled data.

As another example, consider a property that defines the mapping between pixel values and empirically measured photometric values in the original scene using a

hypothetical `ImageBrightnessCurve` object. An extender might wish to have an existing brightness adjustment operator make suitable updates to the brightness mapping, so that its `ImageBrightnessCurve` property will allow the correct original brightness to be computed at each pixel. This can be done without any changes to the implementation of the brightness adjustment operator; instead, it is only necessary to register a RIF with the `OperationRegistry`.

Every node of an image chain may be queried for its property values. There are five ways that a property may be assigned a value at a particular node:

1. It may be *copied* from the node's sources. This is the default behavior if no other behavior is specified. This allows information to flow through image chains freely, to be used farther down the chain. If multiple sources contain a property with the same name, the value will be copied from the first source to define the property.
2. It may be *produced* by the node from non-property information available to the node. For example, a file reader operation may make the value of an image tag available as a property.
3. It may be *synthesized* by the node from a rendering. In particular, `RenderedOp` nodes provide properties containing the image dimensions. Reading such a property causes the node to be rendered.
4. It may be *inherited*; that is, produced computationally from the properties of the node's sources. The `PropertyGenerator` interface, described below, allows users to add their own code for property inheritance.
5. It may be *set* explicitly by the user using the `setProperty` method. This method may be called only on nodes that are not frozen. Synthetic properties (case 3 above) may not be overridden.

Properties are always computed by following the imaging chain from the node being queried upwards, node by node. Even if optimizations are performed as part of the rendering process, resulting in `OpImages` that "skip" various nodes of the graph, the properties will be computed using all of the nodes. This mechanism ensures consistent property values under all conditions.

3.10 Regions of Interest

One common property, supported by all the standard operators, is the ROI, or region of interest property. A region of interest is simply a description of a subarea of the image, which is propagated through the imaging chain. ROIs are stored using the `ROI` and `ROIShape` classes. Most ROIs will be initially defined using a `ROIShape`, which stores its area using the `Shape` classes from the Java 2D

API. These classes generally define an area by means of a geometrical description of its outline, and provide efficient means for storage, transformation, and application of boolean operations to Shapes. The ROI class stores an area as a greyscale image. A ROIShape may be converted into a ROI, but not vice-versa.

ROI, like other properties, does not affect pixel processing. The ROI property will be carried forward through point operators, and transformed appropriately by geometric and warp operators. A ROI may be used as an argument to the set method of TiledImage in order to copy a selected area of a source into an existing TiledImage, as well as by various compositing operators.

3.11 Programming Example

Finally, as a concrete example of using the Java Advanced Imaging API, a simple example program is given below. This example is intended merely to be illustrative of the API. For more detailed examples, please consult the document *Programming in Java Advanced Imaging*, available on the web site.

The example illustrates the use of operators in the rendered domain. In the rendered domain, image sources are objects which implement the RenderedImage interface. These sources are specified as parameters in the construction of new image objects. An arbitrary directed acyclic graph (DAG) may be formed by these dependencies. This model provides immediate mode semantics - as a new operation is added to the chain, it appears to compute its results immediately. The example below is a simple, but functional, class which creates two constant images and then adds them together.

Listing 3-1 An example using rendered images

```
// A Java Advanced Imaging API based program that illustrates
// the adding of two images. A class, AddExample, is defined
// to do the work.
import javax.jai.*;
import javax.jai.widget.*;
import java.awt.Frame;

public class AddExample extends Frame {

    // ScrollingImagePanel is a utility widget that
    // contains an ImageCanvas and horizontal and vertical
    // scrollbars. It is an image sink.
    ScrollingImagePanel imagePanel1;

    // For simplicity, we just do all the work in the
    // class constructor. It is assumed that the
    // ParameterBlock classes have been constructed
    // by the caller.
    public AddExample(ParameterBlock param1,
                     ParameterBlock param2) {

        // The 'JAI' class is a convenience class for
        // instantiating image operators.

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

        // Create another constant image
        RenderedOp im1 = JAI.create("constant",param2);

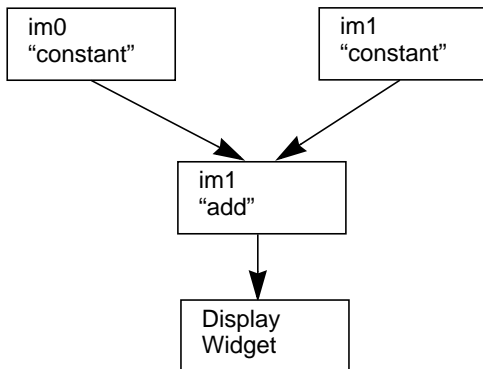
        // Add the two images together.
        RenderedOp im2 = JAI.create("add", im0, im1);

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

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

The next figure shows the processing graph in this example.

Figure 3-1 The Rendered graph given in the last example.



To illustrate what is happening here, we concentrate on just one line of code:

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

The `JAI.create` method takes three parameters:

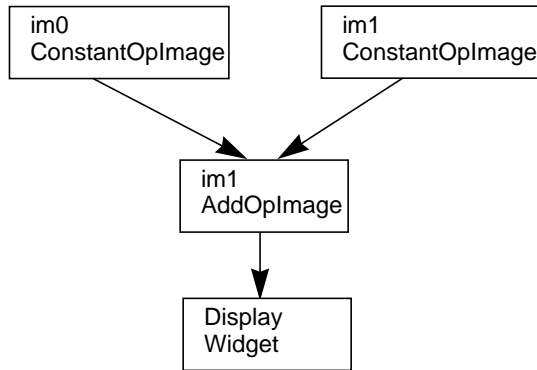
- A string, “constant”, which identifies what kind of operator is to be created.
- A parameter block (presumed here to be constructed outside the class) which in this case contains
 - Layout parameter that gives detailed information on the rendered image to be created (height, width, origin, tile size, *etc.*)
 - An array of integer values that specify the constant color of the rendered image.

The number of parameters supplied in the parameter blocks must be consistent with the parameters required by a “constant” type image. `JAI.create` does a lookup in the operation registry to find an appropriate object to create and return. In fact the code above is semantically equivalent to the following:

```
// Create a constant image
OperationRegistry registry =
    OperationRegistry.getOperationRegistry();
RenderedOp im0 = registry.create("constant", param1);
```

What is the object returned by the registry.create method? The JAI class returns instances of RenderedOp, which is a subclass of PlanarImage. PlanarImage implements the RenderedImage interface. Either now or later (when pixels are actually required from im0), a ConstantOpImage object will be instantiated to actually create the pixels. Once pixels start flowing, the graph shown above will look like this:

Figure 3-2 The ultimate form of Rendered graph given in the last example.



The display widget is the driver of the process; no pixels need be produced until the display widget wants to put them on the screen.

