

SIGGRAPH 2000

[NEW ORLEANS]



Introduction to the Java(TM) Advanced Imaging API





Abstract

The Java™ Advanced Imaging (JAI) API is a cross-platform, flexible, extensible toolkit for adding advanced image processing capabilities to applications for the Java platform. It includes features such as tiled images, lazy evaluation, multi-resolution imaging, meta-data handling, and network imaging. This course will introduce participants to the imaging capabilities of the Java platform and provide practical examples of how to make use of JAI in their applications.

The three major areas of JAI functionality will be described in detail: pixel-based, or "rendered" imaging; resolution- and rendering-independent, or "renderable" imaging; and networked, or "remote" imaging. In each of these areas, participants will learn both how to use the standard capabilities of the API as well as how to write their own extensions. An application example will be presented in depth.



Lecturer Information



Brian Burkhalter (Organizer)

Staff Engineer, Sun Microsystems, Inc.

Email: Brian.Burkhalter@eng.sun.com

Brian Burkhalter is the technical lead for core development of the Java Advanced Imaging API. He has worked since 1996 in the Imaging Software Technologies group of the Sun Microsystems Computer Systems division.

His previous work at Sun was primarily on the XIL imaging library. He has more than a decade of experience in image processing and particularly in processing of remotely sensed imagery and photogrammetry. He has represented the Java Advanced Imaging API at the JavaOne developer's conference and the Environmental Systems Research Institute (ESRI) User's Conference.

He holds a BA in Mathematics from the University of Chicago, a MSEE from the University of Washington, and the Degree of Engineer (EE) from Stanford University.



Course Outline - Part 1

- ➊ Introduction
- ➋ Part 1 - JAI and Java 2D Imaging Basics

- ➌ Introduction to JAI Features
- ➌ Imaging in Java - the AWT and Java 2D APIs
- ➌ Fundamental Java 2D image classes
- ➌ Fundamental JAI image classes
- ➌ Other fundamental JAI classes
- ➌ JAI non-image classes
- ➌ Instantiating a JAI operation

Duration: 30 minutes



Course Outline - Part 2



Part 2 - Working with Rendered images and properties

- The `RenderedImage` interface
- Rendered image semantics
- Importing an image
- Displaying an image
- Performing an operation on an image
- Writing a new Rendered operation
- Adding properties to images
- Managing image properties

Duration: 45 minutes



Course Outline - Part 3



Part 3 - Working with Renderable images

- Renderable image semantics
- The `RenderableImage` interface
- Building renderable chains
- Rendering a chain
- Editing a chain
- Writing a new Renderable operation

Duration: 40 minutes



Course Outline - Part 4



Part 4 - Working with Remote images

- Remote image sources
- Overview of networking concepts in the Java platform
- The `RemoteImage` interface

Duration: 15 minutes



Course Outline - Part 5



Part 5 - Using the JAI Image I/O Package (unofficial)

- ➊ **InputStream and extensions**
- ➋ **The `ImageCodec` class**
- ➌ **The `ImageDecoder` and `ImageEncoder` interfaces**
- ➍ **JAI I/O operations**
- ➎ **Adding new image decoders and encoders**

Duration: 15 minutes



Course Outline - Part 6



Part 6 - Writing a JAI application

- Setting up the JAI runtime environment
- Integrating with the Swing toolkit
- Integrating with browsers
- An inside look at a JAI application

Duration: 50 minutes

SIGGRAPH 2000

[NEW ORLEANS]



JAI and Java 2D Imaging Basics



Duration: 30 minutes



JAI and Java 2D Imaging Basics - Outline

- Introduction to JAI Features
- Imaging in the Java platform - the AWT and Java 2D APIs
- Fundamental Java 2D image classes
- Fundamental JAI image classes
- Other fundamental JAI classes
- JAI non-image classes
- Instantiating a JAI operation



Introduction to JAI Features

- "Rendered" imaging - large深深 images
- "Rendered" imaging - tiling/deferred execution
- "Properties" - a framework for meta-data
- "Renderable" imaging - multiresolution imaging
- "Renderable" imaging - interactive editing
- "Remote" images - imaging over the network
- Extensibility



"Rendered" Imaging - Large/Deep Images

- ➊ Large images
- ➋ Varied pixel depths
- ➌ Flexible formats
- ➍ Float and double samples



"Rendered" Imaging - Tiling/Deferred Execution

- Images broken up into tiles
- Image data computed only on demand
- Computed tiles stored in a cache



"Properties" - A Framework for Metadata

- ➊ Every image maintains a simple database of properties:
 - ➋ `String getPropertyNames()`
 - ➋ `Object getProperty(String name)`
- ➋ **PropertySource** interface contains these methods
- ➋ **PropertyGenerator** interface used to compute property values dynamically



"Renderable" Imaging - Multiresolution Imaging

- **RenderableImage == object that creates an image on demand**
- **Resolution and rendering independence**
- **Control rendering with a `RenderingContext` object**



JAVA®

"Renderable" Imaging - Interactive Editing

- Building chains of operations
- Chains may be reconfigured (change node's sources)
- Nodes may be reconfigured (change node's parameters)
- Chains are evaluated on demand



"Remote" Images - Imaging Over The Network

- Standard implementation uses Remote Method Invocation (RMI)
- Protocol-neutral interface
- Transparently distribute processing across the network



Extensibility

- ➊ Extensibility designed in from the start
- ➋ Add new operators
- ➌ Add new implementations of existing operators
- ➍ Add new image formats
- ➎ Write subclasses for interpolation, border extension, warping
- ➏ Write subclasses for custom image representations
- ➐ Write property (metadata) generators



Introduction to JAI Features (cont.)

- JAI has classes that address a number of areas of functionality:
- Imaging operations
- Image collections
- Image codecs
- Interpolation/border extension/warping
- Tile caching
- Tile scheduling



Imaging Operations

- Over 80 standard operations
- Real and Complex arithmetic ops
- Bitwise logical ops
- Area ops: Convolve, Gradient
- Geometric ops: Affine, Rotate, Scale, Shear, Translate
- Statistics ops: Mean, Extrema, Histogram
- Transform ops: DFT/IDFT, DCT/IDCT



Image Collections

- ➊ Integrate with the Java Collection API
- ➋ Operate on a **Set**, **List**, **Vector**, etc. of images
- ➌ Provide specific image-oriented collections:
 - ➍ Pyramid (decimated and differences)
 - ➎ MIPMap (decimated)
 - ➏ Sequence (timestamped)
 - ➐ Stack (spatially notated)



Image Codecs

- ➊ Read/write images in many standard formats
- ➋ Add new encoders and decoders
- ➌ Auto-detect image formats based on header bytes or full scan
- ➍ Handle tiled file formats



Interpolation/Border Extension/Warping

- Objects that perform common imaging sub-operations
- Interpolation: compute a subpixel value
- BorderExtender: produce data outside of an image for interpolators
- Warp: compute a rectangle of warped source pixel positions
- Leverage subclassing for extensibility
- Detect known subclasses for speed



Tile Caching

- ➊ When a tile is computed, cache it
- ➋ When a tile is requested, check the cache
- ➌ Provide a public interface, private implementation
- ➍ New cache implementations are possible



Tile Scheduling

- ➊ Feed tile computation to multiple threads
- ➋ Analyze tile-to-tile dependencies, compute leaves first
- ➌ Compute multiple tiles at once
- ➍ Maximize tile reuse



Imaging in the Java Platform - the AWT and Java 2D APIs

- ➊ Java 1.0/1.1; the AWT
 - ➌ "push" model
- ➋ Java 2 platform (a/k/a JDK™ 1.2); Java 2D
 - ➌ "immediate mode" model
- ➌ Java Advanced Imaging
 - ➌ "pull" model



Java 1.0/1.1 - the AWT

A "Push" Model

- ➊ `java.awt.Image`: opaque object, often native
- ➋ `ImageProducer/ImageConsumer`:
 - ➌ "Push" pixels to a consumer
 - ➍ Good for Web graphics
- ➎ `image.getGraphics`: perform drawing on an image
 - ➏ Can only draw on "off-screen" images
 - ➐ Simple raster drawing methods



Java 2 Platform (a/k/a JDK(TM) 1.2) - Java 2D *An "Immediate Mode" Model*

- `java.awt.image.*`: many new classes
- `BufferedImage`: in-memory image, single tile
- Immediate-mode operations (`BufferedImageOp`)
- Accelerated display of common formats
- `RenderedImage/RenderableImage`: expansion interfaces



		Push Model	Immediate Mode Image Buffer Model
Major Interfaces/ Classes		<ul style="list-style-type: none">• Image• ImageProducer• ImageConsumer• ImageObserver (JDK 1.0.x, 1.1.x)	<ul style="list-style-type: none">• BufferedImage• Raster• BufferedImageOp• RasterOp (Java™ 2D API)
Pros		<ul style="list-style-type: none">• Processing driven by image availability (e.g. over network)• Images processed incrementally	<ul style="list-style-type: none">• Simplest programming interface• Commonly used model
Cons		<ul style="list-style-type: none">• Requires transfer (but not processing) of complete images• More complex programming interface	<ul style="list-style-type: none">• Requires memory allocation of complete images• Requires processing of complete images



Java Advanced Imaging

A "Pull" Model

- `javax.media.jai.PlanarImage` (superclass)
- `TiledImage`: in-memory image with multiple tiles
- `OpImage`: procedurally-defined image
- Compute tiles on demand - "pull" pixels from sources



		Immediate Mode Image Buffer Model	Pull Model
Major Interfaces/ Classes	<ul style="list-style-type: none">• <code>BufferedImage</code>• <code>Raster</code>• <code>BufferedImageOp</code>• <code>RasterOp</code> (Java™ 2D API)	<ul style="list-style-type: none">• <code>RenderableImage</code>• <code>RenderableImageOp</code> (Java 2D API)	
Pros	<ul style="list-style-type: none">• Simplest programming interface• Commonly used model	<ul style="list-style-type: none">• Stores/processes only required data• Allows lazy evaluation	
Cons	<ul style="list-style-type: none">• Requires memory allocation of complete images• Requires processing of complete images	<ul style="list-style-type: none">• More complex programming interface• More complex implementation	



Fundamental Java 2D Image Classes

- `DataBuffer`
- `SampleModel`
- `Raster`
- `ColorModel`
- `BufferedImage`
- `RenderedImage`
- `ParameterBlock`
- `RenderedImageFactory (RIF)`
- `RenderableImage`



Java 2D Image Classes - *DataBuffer*

- A place to store pixels
- Arrays must have a specific type (no "void *" equivalent)
 - Java 2D currently supports only integral data types
- Can have multiple "banks" per **DataBuffer**, offset per bank

bankdata[0] offsets[0] = 0
bankdata[2] offsets[1] = 0
 offsets[2] = 3





Java 2D Image Classes - SampleModel

- ➊ Describes the layout of pixels in a DataBuffer
- ➋ Has a width and height
- ➌ `getPixels/setPixels`
 - ➍ Extract/insert data from a simple external format
- ➎ `getPixelData/setPixelData`
 - ➏ Extract/insert data in a more optimized format
- ➐ Can recognize known subclasses, access pixels directly



Java 2D Image Classes - ComponentSampleModel

- Each sample lives in its own DataBuffer array element
- Superclass for PixelInterleavedSampleModel and BandedSampleModel
- JAI prefers to work with ComponentSampleModel



Java 2D Image Classes - PixelInterleavedSampleModel

- Each sample lives in its own `DataBuffer` array element
- Samples for a given pixel are adjacent
- Get to next pixel by adding `pixelStride` to the index
- Get to next line by adding `scanlineStride` to the index





Java 2D Image Classes - *BandedSampleModel*

- Each sample lives in its own `DataBuffer` array element
- Samples for a given pixel live in separate buffers
- Get to next pixel by adding 1 to the index
- Get to next line by adding `scanlineStride` to the index





Java 2D Image Classes - MultiPixelPackedSampleModel

- ➊ Each pixel has a single sample (e.g., gray scale or palette images)
- ➋ Multiple pixels are stored in a single array element
- ➌ Get to next sample by shifting a given number of bits
- ➍ Get to next pixel by adding 1 to the index
- ➎ Get to next line by adding `scanlineStride` to the index

Gy|Gy|Gy|Gy|Gy|Gy|Gy|Gy|Gy|Gy|Gy|Gy|



Java 2D Image Classes - SinglePixelPackedSampleModel

- Each pixel lives in its own **DataBuffer** array element
- Samples for a given pixel are concatenated
- Get to next pixel by adding 1 to the index
- Get to next line by adding **scanlinestride** to the index

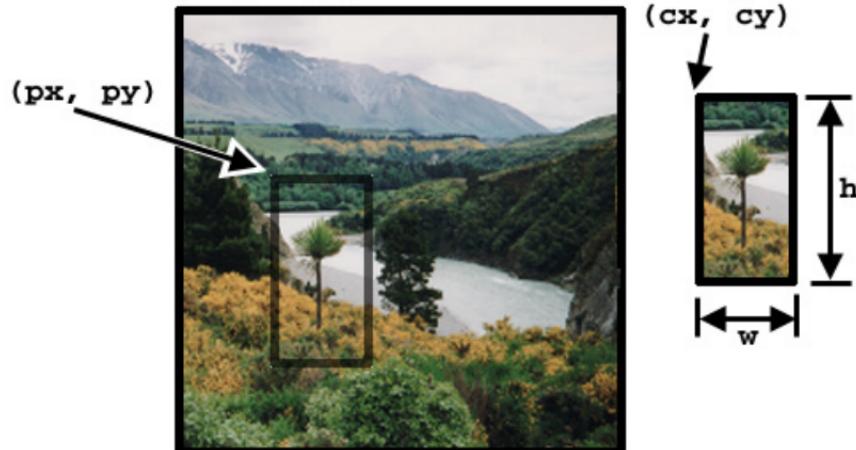




Java 2D Image Classes - Raster

- SampleModel + DataBuffer + (X, Y) Origin
- Raster is read-only, WritableRaster is read/write
- Constructed using factory methods defined in Raster
- Can create a "child" Raster
 - Access a sub-rectangle/sub-band set, translate to a new origin

```
createChild(px, py, w, h, cx, cy, null)
```





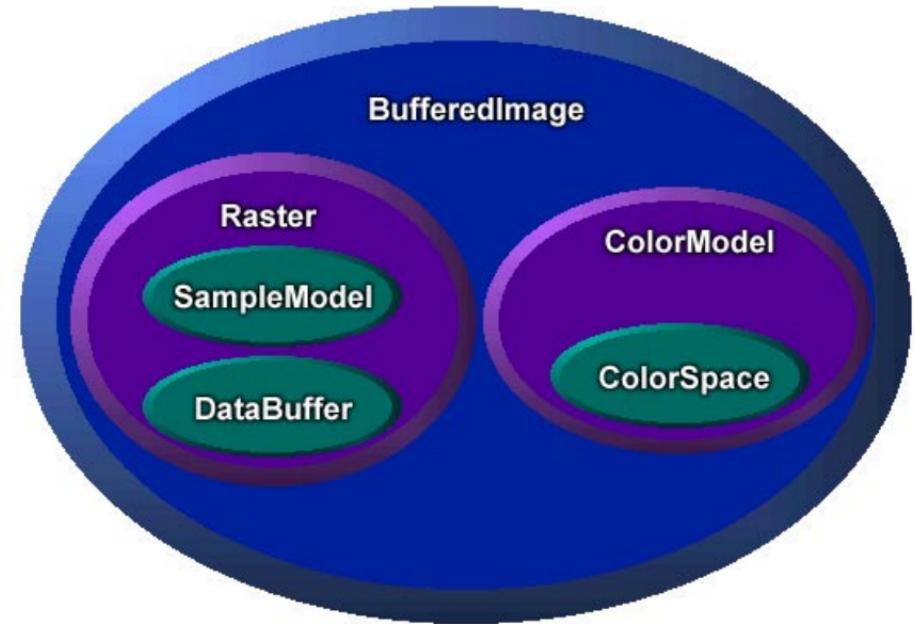
Java 2D Image Classes - ColorModel

- ➊ Provides colorimetric interpretation of a pixel
 - ➋ Encapsulates translation of a pixel value to color components and alpha
 - ➋ Requires a `java.awt.color.ColorSpace` object
- ➋ Required for rendering to screen/printer or for color conversion
- ➋ Assumes pixel components have been extracted using a `SampleModel`
- ➋ Can use shifting and masking, lookup tables, arbitrary computation



*Java 2D Image Classes - *BufferedImage**

- Java 2D's fundamental image type
- Raster + ColorModel
- A single-tile, in-memory image
- Extends `java.awt.Image` and implements
`WritableRenderedImage`





*Java 2D Image Classes - *RenderedImage**

- ➊ Expansion interface representing an image with read-only data
 - ➋ Implemented by `BufferedImage` and JAI images
- ➋ An image defined by dimensions, tile grid, `SampleModel`, `ColorModel`
 - ➌ The `ColorModel` may be null
- ➌ Tiles retrieved by a `getTile` method
 - ➍ Arbitrary data regions obtained via `getData` or `copyData`
- ➍ Provides interface for name-value style attributes (`properties`)
- ➎ Both stored and procedural implementations possible
- ➏ `WritableRenderedImage` subinterface allows editing, tracking of edits



Java 2D Image Classes - ParameterBlock

- Contains image sources and other parameters to an operation
- Used in Java 2D for renderable operations
- Used in JAI for all operations



*Java 2D Image Classes - *RenderedImageFactory**

- An interface containing a single method:

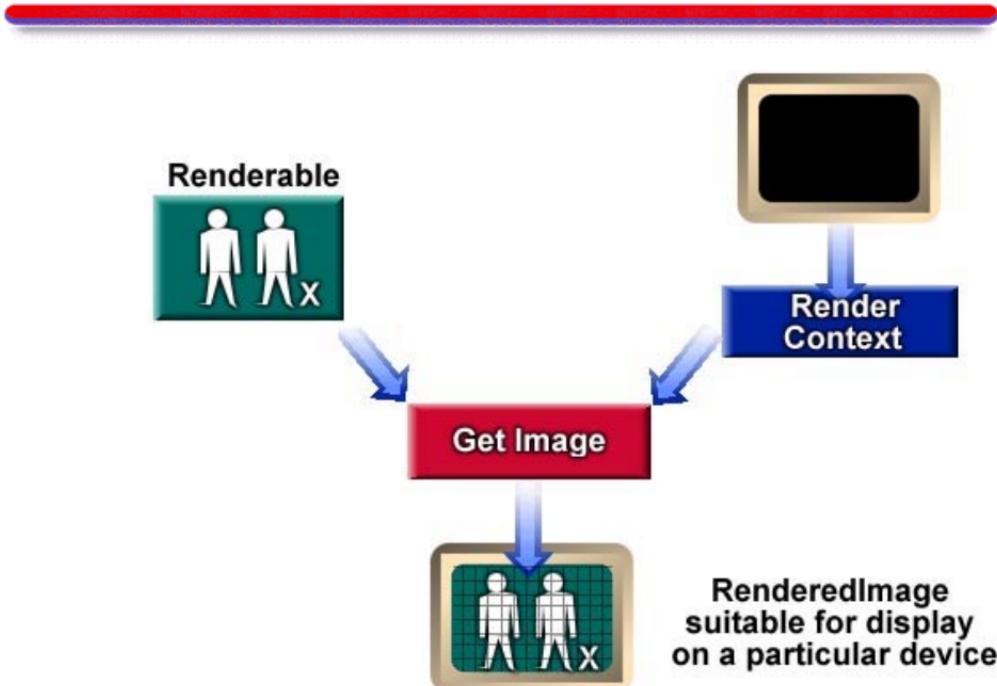
```
RenderedImage create(ParameterBlock paramBlock,  
                      RenderingHints renderHints)
```

- Used by Java 2D as part of **RenderableImage** instantiation
- Used by JAI to instantiate a potential implementation of an operation
- May return **null** if factory cannot produce result



Java 2D Image Classes - RenderableImage

- A resolution- and rendering-independent image
- Details are opaque
- Produces renderings on demand
- Will discuss in detail later





Fundamental JAI Image Classes

- `ImageJAI`
- `PlanarImage`
- `TiledImage`
- `CollectionImage`
- `OpImage`
- `PointOpImage/AreaOpImage/UntitledOpImage/WarpOpImage`



Fundamental JAI Image Classes - ImageJAI

- ➊ Top-level JAI interface
- ➋ Contains property query methods
 - ➌ `String[] getPropertyNames()`
 - ➌ `String[] getPropertyNames(String prefix)`
 - ➌ `Object getProperty(String name)`



Fundamental JAI Image Classes - PlanarImage

- Top-level JAI image class
- Implements `java.awt.image.RenderedImage` and `ImageJAI`
- Maintains source/sink information
- Adds "snapshot" capability
- Adds "cobbling" methods
- Adds "extension" methods
- Concrete subclasses must implement `getTile()`



Fundamental JAI Image Classes - TiledImage

- Straightforward in-memory tiled image
- Extends PlanarImage
- Implements `java.awt.image.WritableRenderedImage`
- Implements `createGraphics` to allow arbitrary drawing



Fundamental JAI Image Classes - CollectionImage

- ➊ Uses standard Java Collection API classes
 - ➋ Set, List, Map, Tree, Vector, ...
- ➋ Image->Image operations automatically generalize to Collection->Collection
- ➌ Collection->Image and Image->Collection ops also possible
- ➍ Nested collections are supported.



Fundamental JAI Image Classes - *OpImage*

- Extends `PlanarImage`
- Implements `getTile()` to call `computeTile()`, cache result
 - `computeTile()` depends on `computeRect()`
- Adds abstract source/dest geometric dependency methods
`mapSourceRect()` and `mapDestRect()`
- Superclass for all image processing operators



Fundamental JAI Image Classes - OpImage subclasses

- PointOpImage
- AreaOpImage
- WarpOpImage
- UntiledOpImage



Fundamental JAI Image Classes - *PointOpImage*

- ➊ Each destination pixel depends only on its corresponding source pixel
 - ➌ `mapDestRect()` is the identity mapping.
- ➋ No need to "cobble" sources
- ➌ Can support in-place operation to reduce memory requirements



Fundamental JAI Image Classes - AreaOpImage

- ➊ Each destination pixel depends on a fixed-size neighborhood around its corresponding source pixel
 - ➌ `mapDestRect()` "pads" by a fixed amount on each side
- ➋ Sources require some "cobbining" around tile edges
- ➌ Subclass only needs to describe the neighborhood shape (padding)



Fundamental JAI Image Classes - WarpOpImage

- ➊ Each destination pixels depends on a neighborhood around its corresponding source pixel
- ➋ Destination-to-source correspondence may be complex
 - ➌ Backward mapping defined by a `Warp` object
 - ➍ Required padding defined by an `Interpolation` object
- ➎ Provides a simple interface for porting, ops may need to override for performance



Fundamental JAI Image Classes - UntitledOpImage

- ➊ Each destination pixel depends on all source pixels
 - ➌ `computeTile()` implemented in terms of abstract `computeImage()`
- ➋ Best to cobble source once
- ➌ Examples: Fourier transforms



Other Fundamental JAI Classes

- `JAI`
- `OperationRegistry`
- `RenderedOp`



Other Fundamental JAI Classes - JAI

- ➊ A set of static `create()` methods for instantiating ops
- ➋ Instances contain references to:
 - ➌ An `OperationRegistry`
 - ➍ A `TileCache`
 - ➎ A `TileScheduler`
 - ➏ A set of rendering hints (`java.awt.RenderingHints`)



Other Fundamental JAI Classes - OperationRegistry

- Maps operation names to a sequence of `RenderedImageFactory` instances
- Each RIF is called in turn to produce a rendering
- First one to succeed "wins"
- New RIFs may be added for new or existing operations
- Preferences between RIFs may be modified



Other Fundamental JAI Classes - `RenderedOp`

- ➊ A lightweight object containing:
 - ➋ An operation name
 - ➋ A `ParameterBlock` with sources and parameters
 - ➋ A reference to an `OperationRegistry`
- ➋ Used to create graphs of operations
- ➋ `RenderedOp` graphs can be transferred over the network, re-rendered using classes available on the other host
- ➋ A `RenderedOp` acquires a rendering on demand (just-in-time)



JAI Non-image Classes

- Histogram
- KernelJAI
- LookupTableJAI
 - ColorCube
- PerspectiveTransform
- ROI
 - ROIShape

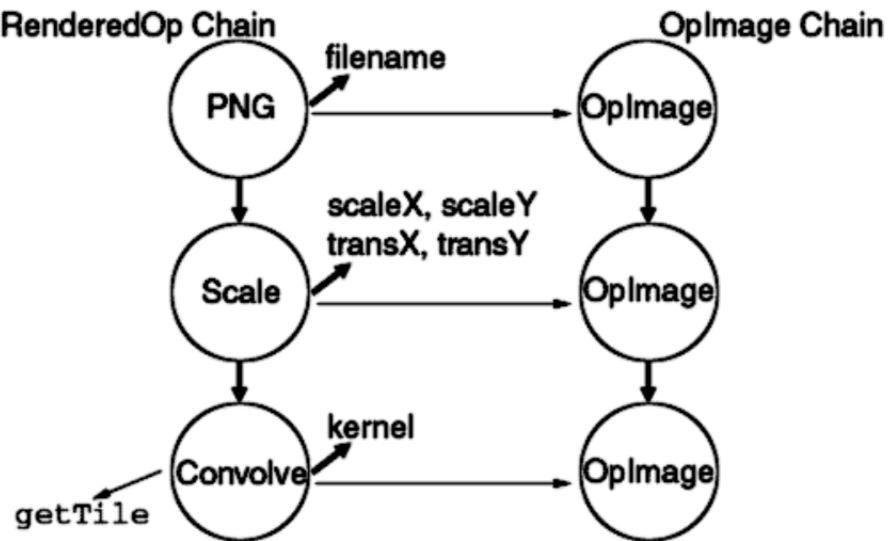


Instantiating a JAI Operation

- ➊ User calls `JAI.create(name, params, hints)`
 - ➋ Parameters checked for validity
 - ⌃ Returns a `RenderedOp` storing the name, parameters, hints, registry
- ➋ Pixel data or other information requested from the `RenderedOp`
 - ➌ `RenderedOp` calls `OperationRegistry.create`
 - ➍ `OperationRegistry` walks the prioritized list of RIFs, calls `create()` on each one
 - ➎ RIF `create` parses arguments
 - > returns `null` if it cannot handle the case
 - > returns a new `OpImage` subclass (or any `RenderedImage`)



Instantiating a JAI Rendered Chain



SIGGRAPH 2000

[NEW ORLEANS]



JAVA®

Working With Rendered Images and Properties



Duration: 45 minutes



Working With Rendered Images and Properties - Outline

- The `RenderedImage` interface
- Rendered image semantics
- Importing an image
- Displaying an image
- Performing an operation on an image
- Writing a new Rendered operation
- Adding properties to images
- Managing image properties



The RenderedImage Interface

- ➊ Implemented by `BufferedImage` in Java 2D and `PlanarImage` in JAI
- ➋ `getMinX()`/`getMinY()`/`getWidth()`/`getHeight()`
- ⌃ `getTileGrid{X,Y}Offset()`/`getTileWidth()`/`getTileHeight()`
- ⌄ `getSampleModel()`
- ⌅ `getColorModel()`
- ⌆ `getTile()` returns a single tile as a `Raster`
 - ➊ `getData()` returns a rectangular region as a `Raster`
 - ⌂ `copyData()` fills or returns a `WritableRaster`
- ⌇ Property accessor methods exist

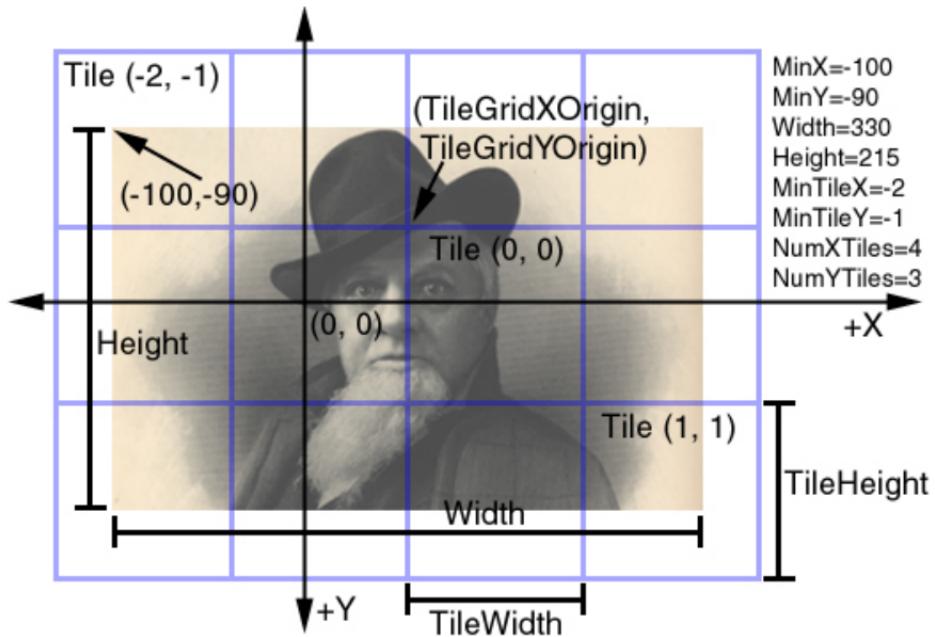


RenderedImage Semantics

- ➊ Has fixed bounds, pixel resolution
- ➋ May contain multiple tiles
- ➌ Each tile has a common `SampleModel`, `ColorModel`
 - ➍ The `ColorModel` may be null
- ➎ Tiles may extend past the image bounds
- ➏ Tiles are accessed procedurally, may be computed on demand



RenderedImage Layout Example





Importing an Image

- ➊ AWTImage operation brings old-style images into JAI
- ➋ Can use a BufferedImage or other RenderedImage directly
- ➌ JAI automatically "wraps" an external RenderedImage with a PlanarImage
- ➍ Standard set of image readers:
 - ➎ BMP, GIF, FlashPix, JFIF, PNG, PNM, TIFF
- ➏ Simple interface:

```
String filename = "image.png";
JAI.create("fileload", filename);
```



Displaying an Image

- ➊ Java 2D handles basic image display via **Graphics2D**:
`drawRenderedImage(RenderedImage, AffineTransform)`
- ➋ Future acceleration possible via **GraphicsJAI** wrapper class
- ➌ **CanvasJAI** class automates use of **GraphicsJAI**



Performing an Operation on an Image



Syntax:

```
JAI.create(String name,  
          ParameterBlock pb,  
          RenderingHints rh)
```



Operation name is located using the `OperationRegistry`



`ParameterBlock` references the sources and parameters



Rendering hints contain "optional" info:

- `ImageLayout` - output image bounds, tile grid, `SampleModel` (incl. data type), `ColorModel`
- Which `TileCache` instance to use
- Which `BorderExtender` instance to use



Code Example



Perform a Rendered operation, set output **SampleModel**:

```
RenderedImage src1, src2;
ParameterBlock pb = new ParameterBlock();
pb.addSource(src1);
pb.addSource(src2);
pb.add(param1);
pb.add(param2);
pb.add(param3);

ImageLayout layout = new ImageLayout();
layout.setSampleModel(sampleModel);
RenderingHints rh =
    new RenderingHints(JAI.KEY_IMAGE_LAYOUT,
                      layout);

RenderedImage dst = JAI.create("opname", pb, rh);
```



Writing a new Rendered Operation

- Subclass `OpImage`
- Implement one of: `getTile`/`computeTile`/`computeRect`
- `getTile`: handle all processing
- `computeTile`
 - Allow `OpImage` to cache the output
- `computeRect(Raster[], WritableRaster, Rectangle)`
 - Allow `OpImage` to cobble sources
- `computeRect(PlanarImage[], WritableRaster, Rectangle)`
 - Use uncobbled sources
- Implement `RIF` or `CRIF` as appropriate
- Implement `OperationDescriptor`
- Register `OperationDescriptor`, `(C)RIF` with `OperationRegistry`



Dealing With Tiled Sources

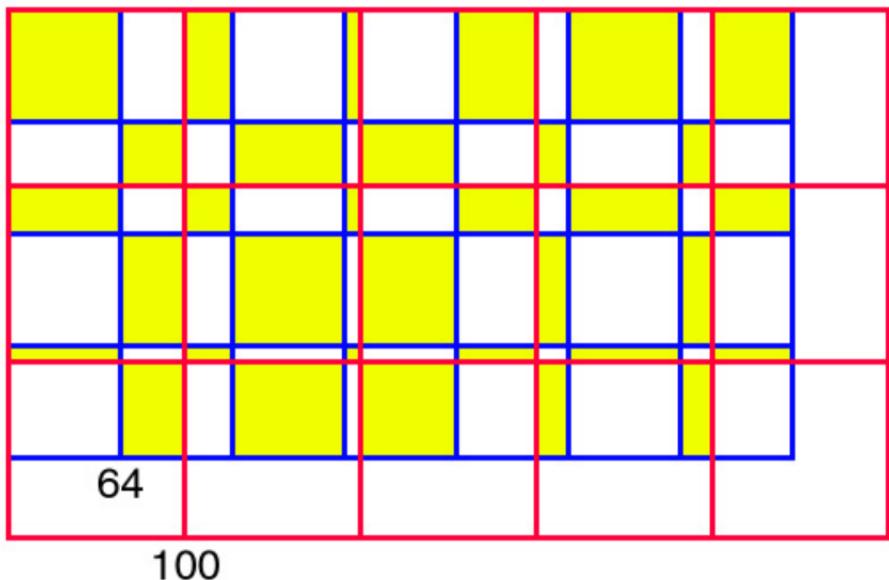
- ➊ Cobbling (copying regions that cross tiles)
- ➋ Iterators



Cobbling Example (Point Operator)



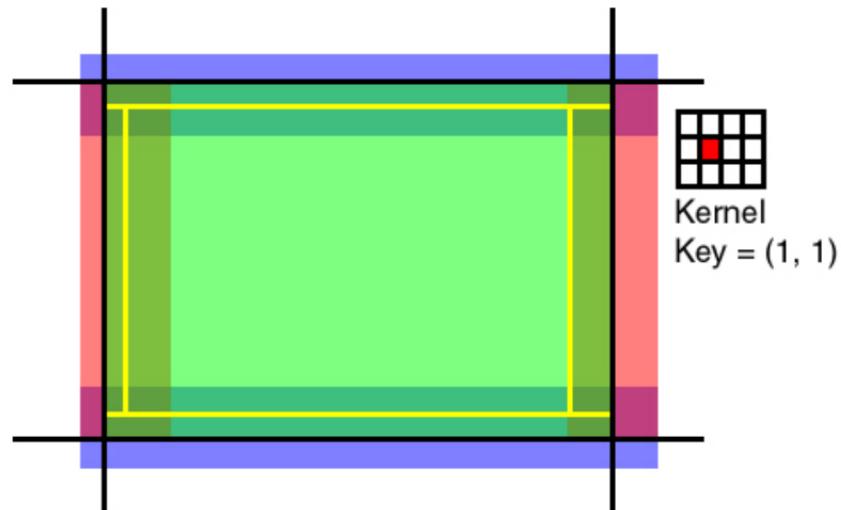
- Source has X tile splits every 100th pixel, destination has X tile splits every 64th pixel





Cobbling Example (Area Operator)

- Each dest pixel X requires source pixels $[X - lpad, X + rpad]$
- Compute dest in four pieces
 - Large central area does not require cobbling





Iterator Example

- ➊ Construct an iterator for each source image
- ➋ Iterator will handle source tile crossings automatically
- ➌ Iterators currently have lower performance



JAVA®

Adding Properties to Images

- File formats may contain meta-information that is decoded as properties
- Properties may be set on a node prior to rendering

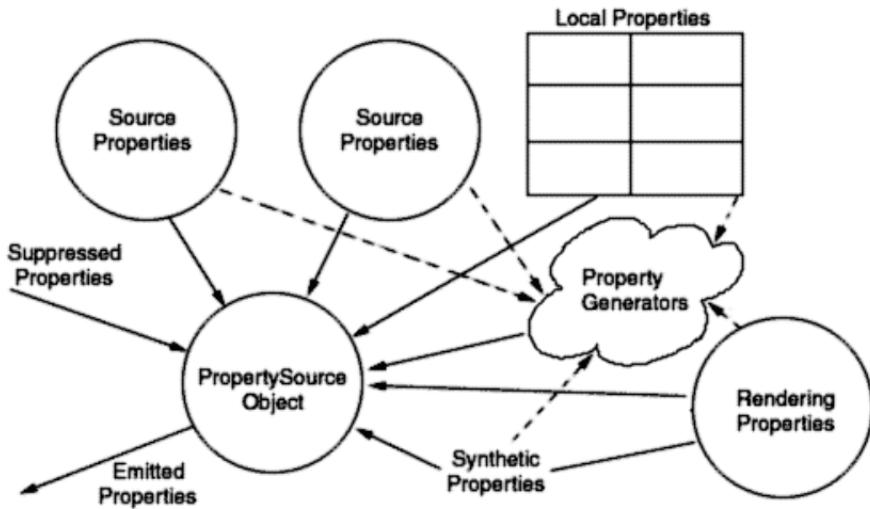


Managing Image Properties

- ➊ Image properties on a RenderedOp node come from five sources:
- ➋ Properties of the node's sources
- ➌ Properties of the node's rendering
- ➍ Synthetic properties based on the rendering (width, height, ...)
- ➎ Locally-set properties (set before rendering the node)
- ➏ `PropertyGenerator` objects attached to the node
 - ➐ Can access the other four types of sources



Managing Image Properties (cont.)





Writing a PropertyGenerator

- A **PropertyGenerator** computes a named property on behalf of a Rendered or Renderable operation node
- May be set to automatically attach to nodes performing a given operation
- Implement `getPropertyNames`, `getProperty`
- Example: DFT op has a property generator that tells whether the output image is Real or Complex
- Computation is based on same property in source, parameter of the op

SIGGRAPH 2000

[NEW ORLEANS]



JAVA®

Working With Renderable Images



Duration: 40 minutes



Working With Renderable Images - Outline

- Renderable image semantics
- The `RenderableImage` interface
- Building renderable chains
- Rendering a chain
- Editing a chain
- Writing a new Renderable operation



Renderable Image Semantics

- ➊ Abstract description of a scene
- ➋ Floating point coordinate system
 - ➌ Default of 1.0 high by aspect ratio wide
- ➍ Two way flow of information
- ➎ Lightweight layer



The RenderableImage Interface

- ➊ Implemented by `RenderableImageOp` (Java 2D) and `RenderableOp` (JAI)
- ➋ `getMinX()`, `getMinY()`, `getWidth()`, `getHeight()` (float)
- ➌ `create{Scaled,Default}Rendering()`
- ➍ `getParameterBlock()`
- ➎ `getSources()`
- ➏ `isDynamic()`
- ➐ `setParameterBlock()` (`RenderableImageOp` & `RenderableOp`)



RenderContext

- ➊ Defined in Java 2D package `java.awt.image.renderable`
- ➋ Provides context in which rendering will be used:
 - ➌ User to device transform (Renderable -> Rendered Coordinates) as a `java.awt.geom.AffineTransform`
 - ➍ Area of interest (in Renderable coordinates) as a `java.awt.Shape`
 - ➎ Rendering hints (tweak rendering)



The ContextualRenderedImageFactory (CRIF) Interface

- ➊ Defined in Java 2D package `java.awt.image.renderable`
- ➋ Plays role analogous to that of `RenderedImageFactory`
- ➌ `mapRenderContext()` transforms `RenderContext` for inputs
- ➍ `create()` Same role as `RIF.create`
- ➎ `isDynamic()` indicates if host can cache previous results.



Building Renderable Chains

- ➊ Similar approach as for `RenderedOps`
- ➋ Sources and parameters put in `ParameterBlock`
- ➌ `RenderableOp` uses `OperationRegistry` to get CRIF
 - ➍ CRIFs do not have preferences
 - ➎ `JAI.createRenderable(...)`
 - ➏ Custom implementation of `RenderableImage` - do whatever...



JAVA®

Rendering a Chain

- ➊ createRendering() Methods.
- ➋ Mechanics



createRendering Calls

- ➊ `createRendering(RenderContext renderContext)`
 - ➌ Primary interface, uses RenderContext to generate a RenderedImage
- ➋ `createScaledRendering(int w, int h, RenderingHints hints)`
 - ➌ Computes appropriate user to device transform for requested image size
- ➌ `createDefaultRendering()`
 - ➌ Computes a 'default' size image

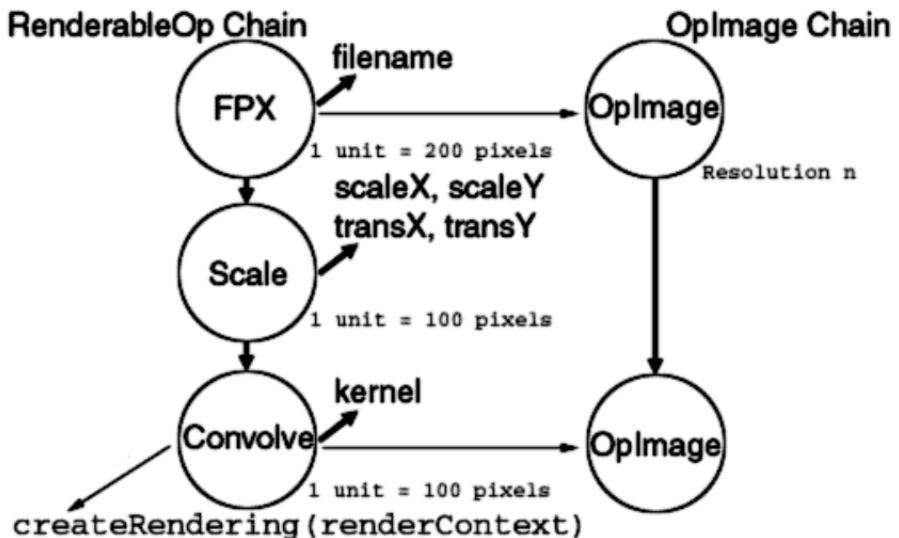


Renderable Rendering Mechanics

- ➊ Update `RenderingContext` for each source
 - ➋ `mapRenderingContext` method in **CRIF**
- ➋ Call `createRendering()` for each source
- ➌ Apply your operation, if any, to resultant `RenderedImage`s
 - ➍ `create()` method in **CRIF**
- ➎ Return your operation



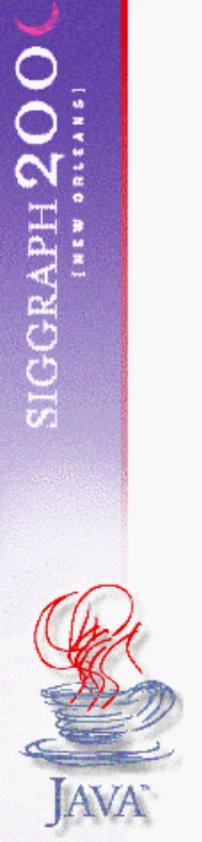
Instantiating a JAI Renderable Chain





Editing a Chain

- ➊ Edits to a `RenderableImage` chain are seen in the result of the next `createRendering` call
- ➋ `setParameterBlock`
 - ➌ Change sources
 - ➌ Change parameters
- ➌ `isDynamic`:
 - ➌ Subsequent `createRendering` calls may differ, even if `ParameterBlock` has not changed



Writing a new Renderable Operation

- `RenderableImageOp`
- `ContextualRenderedImageFactory (CRIF)`
- Implement `isDynamic()`
- Implement `Rectangle2D getBounds2D(ParameterBlock paramBlock)`
 - see `Rectangle2D.Float` and `Rectangle2D.Double`
- Implement `mapRenderContext()`
- Implement `create()`



Implementing mapRenderContext

- ➊ Decode ParameterBlock
- ➋ Decode Hints (from dest)
- ➌ Check Properties (from source)
- ➍ Create RenderContext for input.



mapRenderContext Example

```
public RenderContext  
mapRenderContext(int i, RenderContext rc,  
                  ParameterBlock pb,  
                  RenderableImage parent) {  
    if (i > 1) {  
        throw new IllegalArgumentException  
            ("Composite only takes two inputs");  
    }  
  
    RenderContext ret      = (RenderContext)rc.clone();  
    Shape         rcShape = rc.getAreaOfInterest();  
    Shape         inShape = (Shape)pb.getObjectParameter(i);  
  
    ret.setAreaOfInterest(intersect(inShape, rcShape));  
    return ret;  
}
```



Implementing create

- ➊ Figure out what you were going for in `mapRenderingContext`
- ➋ Construct `RenderedImage` based on inputs
- ➌ Often passes off work to RIF



create Example

```
public RenderedImage create(RenderContext rc,
                            ParameterBlock srcPB) {
    AffineTransform at = rc.getTransform();
    Shape aoi = rc.getAreaOfInterest();

    aoi = at.createTransformedShape(aoi);

    ParameterBlock pb = new ParameterBlock();
    for (int i=0; i < srcPB.getNumSources(); i++) {
        RenderedImage ri = srcPB.getRenderedSource(0);
        Shape clip = (Shape)srcPB.getObjectParameter(0);

        clip = at.createTransformedShape(clip);
        clip = intersect(clip, aoi);

        pb.addSource(ri);
        pb.add(clip);
    }
    return create(pb, rc.getRenderingHints());
}
```



Renderable Summary

- Framework for resolution independent imaging.
- Promotes two way flow of information
- Lightweight layer

SIGGRAPH 2000

[NEW ORLEANS]



JAVA®

Working With Remote Images



Duration: 15 minutes



Working With Remote Images - Outline

- Remote image sources
- Overview of networking concepts in the Java platform
- The `RemoteImage` interface



Remote Image Sources

- ➊ Images from network file systems, URLs, sockets, etc.
 - ➋ Make connection, use `InputStream` abstraction
- ➋ Images computed by a remote JAI server
 - ➌ Use `Remote Method Invocation` to make image appear local



Overview of Networking Concepts in the Java Platform

- ➊ **java.io package:** input and output streams, filters
 - ➋ Streams may be chained together to add functionality
- ➋ **java.net package:** sockets, URLs
- ➌ **java.rmi package:** create Java technology objects on another host
 - ➋ Use serialization to define the wire protocol
 - ➋ Transfer deep copies to the remote host
 - ➋ Maintain a local stub object



The RemoteImage Interface

- ➊ Extends PlanarImage
- ➋ Constructor specifies the server where it will live
- ➌ Evaluates a RenderedOp or RenderableOp chain
- ➍ Chain sent to server using Serialization, RMI
- ➎ Multiple remote servers may be used
 - ➏ Data may pass between servers without returning to client



Example of Using RemoteImage



Using `RemoteImage` is straightforward:

```
PlanarImage source0, source1;  
RenderedOp op = JAI.create("add", source0, source1);  
RemoteImage im = new RemoteImage(serverName, op);
```



Now `im` can be used like any other image



Using RemoteImage Efficiently

- ➊ Ensure sources can be copied to the server efficiently
 - ➋ URL and FileLoad ops are good candidates
- ➋ Ensure output can be sent back to the local host efficiently
 - ➌ Use RenderableOp chains to get only the necessary resolution

SIGGRAPH 2000

[NEW ORLEANS]



JAVATM

Using the JAI Image I/O Package (unofficial)



Duration: 15 minutes



Using the JAI Image I/O Package (unofficial) - Outline

- **InputStream and extensions**
- **The ImageCodec class**
- **The ImageDecoder and ImageEncoder interfaces**
- **JAI I/O operations**
- **Adding new image decoders and encoders**



The JAI Image I/O Package (unofficial)

- ➊ JAI provides an unofficial image I/O package
 - ➋ This is to be superseded by a separate image I/O package
 - ➋ New image I/O package initially separate, later in JRE (JDK)
- ➋ Supports reading and writing common image formats
- ➋ New formats may be added
- ➋ Codecs are based around standard `InputStream/OutputStream` classes
- ➋ Package may be used separately from the rest of JAI
- ➋ A new standard extension will subsume this package



The InputStream Abstraction

- ➊ The `InputStream` class represents a source of bytes
- ➋ Some `InputStream` methods:
 - ➌ `read`: read bytes
 - ➌ `skip`: skip bytes
 - ➌ `mark/reset/markSupported`: return to a previous point
- ➌ File sources: `stream = new FileInputStream(filename);`
- ➌ Network sources: `stream = Socket.getInputStream();`



Adding Functionality to InputStream

- ➊ `ByteArrayInputStream`: read from data already in memory
- ➋ `FileInputStream`: read from a local file on disk
- ➌ `DataInputStream`: read from another stream, adding methods:
 - ➍ `readChar`, `readInt`, `readFloat`, `readDouble`, etc.
- ➎ `BufferedInputStream`: read from another stream, perform buffering



The ImageCodec Classes (unofficial)

- `ImageCodec`: registry of classes supporting a given format
- `ImageDecoder`: abstract superclass for image decoders
- `ImageDecoderParam`: abstract superclass for decoding parameters
- `ImageEncoder`: abstract superclass for image encoders
- `ImageEncoderParam`: abstract superclass for encoding parameters



The ImageCodec Class (unofficial)

- ➊ One subclass of `ImageCodec` for each supported format
- ➋ Has a set of static factory methods:
 - ➌ Query codecs
 - ➍ Register/unregister codecs
 - ➎ Locate codecs that support a particular format
- ➏ Each format implements instance methods for its format:
 - ➐ Get information about the format
 - ➑ Recognize its format in an `InputStream`
 - ➒ Create `ImageDecoder` and `ImageEncoder` instances



The ImageCodec Class (unofficial)



ImageCodec is generally used statically:

```
public ImageDecoder
getDecoder(InputStream src) {
    String[] names =
        ImageCodec.getDecoderNames(src);
    System.out.println("Image is in " +
                       names[0] + " format.");
    return ImageCodec.createImageDecoder(names[0],
                                         src,
                                         null);
}
```



The ImageDecoder and ImageEncoder Interfaces (unofficial)

- ➊ No concrete public implementations
- ➋ Set parameters, perform decoding or encoding
- ➌ Each codec has its own unique parameter class
 - ➍ "Do the right thing" if no parameter supplied
 - ➎ Set some basic parameters without knowing about the specific implementation



Working With ImageCodecs (unofficial)



Decoding a stream is simple:

```
public RenderedImage  
decodeImage(InputStream src) {  
    ImageDecoder dec = getDecoder(src);  
    return dec.decodeAsRenderedImage();  
}  
  
public void  
encodeImage(RenderedImage im,  
            OutputStream dst,  
            String format) {  
    ImageEncoder enc =  
        ImageCodec.createImageEncoder(format,  
                                       dst,  
                                       null);  
    enc.encode(im);  
    dst.flush();  
}
```



The JAI "fileload," "url," and "stream" Operations



JAI has standard operators to simplify the image loading process:

```
String filename;  
im = JAI.create("fileload", filename, hints);  
  
URL url;  
im = JAI.create("url", url, hints);  
  
SeekableStream stream; // unofficial  
im = JAI.create("stream", stream, hints);
```



Writing a new Codec (unofficial)

- ➊ Write an `ImageDecoder` subclass
- ➋ Write an `ImageEncoder` subclass
- ➌ Write an `ImageCodec` subclass
- ➍ Register the new `ImageCodec`



Write an ImageDecoder Subclass (unofficial)

- ➊ Implement a `decode` method
- ➋ Typically, return a new object that implements `RenderedImage`
- ➌ Simple case: return value is a `BufferedImage`
- ➍ More sophisticated: object implements `RenderedImage`, responds to `getTile()` requests
- ➎ This approach allows use of sources such as tiled TIFF, FlashPIX, NITF



Write an ImageEncoder Subclass (unofficial)

- Implement an encode method
- Encoder uses image format (bit depth, ColorModel) and encoder param to determine output format



Write an ImageCodec Subclass (unofficial)



Implement instance methods:

- `getFormatName`
- `getNumHeaderBytes`
- `isFormatRecognized`
- `createImageDecoder`
- `createImageEncoder`
- `canEncodeImage(RenderedImage, ImageEncodeParam)`



*Register the new **ImageCodec** (unofficial)*

- `ImageCodec.registerCodec(new XXXCodec());`



Writing an Application With JAI



Duration: 50 minutes



Writing an Application With JAI - Outline

- Setting up the JAI runtime environment
- Integrating with the Swing toolkit
- Integrating with browsers
- An inside look at a JAI application



Setting up the JAI Runtime Environment

- ➊ Add JAI .jar files to classpath when compiling and running
 - ➋ `jai_core.jar`
 - ➋ `jai_codec.jar` (name will change)
 - ➋ `mlibwrapper_jai.jar` (optional)
- ➋ Place the directory containing native libraries on the library search path
 - ➋ WindowsTM: `PATH` variable
 - ➋ SolarisTM: `LD_LIBRARY_PATH` variable
- ➋ `jai_core.jar` contains a registry initialization file
- ➋ `mlibwrapper_jai.jar` contains "glue" for native acceleration, may be deleted
- ➋ `jai_codec.jar` may be used without `jai_core.jar` for image I/O



Integrating With the Swing Toolkit

- ➊ Can draw to a Swing canvas using normal **Graphics2D** methods:
 - ➋ `drawRenderedImage()`
 - ➋ `drawRenderableImage()`
- ➋ Swing has an **Icon** interface that draws an image on demand
 - ➌ `IconJAI` class demonstrates how to integrate with JAI
- ➌ **BufferedImage** is a **WritableRenderedImage**
 - ➍ Can be constructed from a **Raster**
 - ➍ Can be populated with one of the **CopyData** methods or drawing.
- ➍ Example `DisplayJAI` to be discussed



Integrating With Browsers

- ➊ JAI requires the Java 2 platform.
 - ➋ Sun's Java Plug-in supports the Java 2 platform
- ➋ Security issues (Native code)



An Inside Look at an Application Using JAI

- ➊ Describe application
- ➋ Describe operator
- ➌ Operator implementation
 - ➍ `OpImage` subclass
 - ➎ Image factory classes: `RIF` and `CRIF`
 - ➏ `OperationDescriptor`
- ➐ Swing display component
- ➑ Example Application
 - ➒ `OperationRegistry` configuration
 - ➓ Use of Swing display component
 - ➔ Use of new operator



Describe Application

- ➊ Simple image viewer and subsampler
- ➋ Loads image from a URL
 - ➌ URL entered in a `JTextField`
- ➍ Subsamples image using new operator
 - ➎ Subsampling factor set via a `JSlider`
- ➏ Uses Rendered layer



Describe Operator

- ➊ Subsampling operator "Subsample"
 - ➋ specific implementation for `byte` data only
 - ⌃ integral subsampling factors
 - ⌂ averages over adaptive source window
- ➋ Supports rendered and renderable layers
- ➌ Preferences can be set for interplay with "Scale" operator
 - ⌃ "Scale" can backup "Subsample"
 - ⌃ "Subsample" can be used for "Scale" in some cases



Operator Implementation

- **OpImage subclass**
- **Image factory classes: RIF and CRIF**
- **OperationDescriptor**



OpImage subclass

- Performs actual computation
- Derives from **OpImage** hierarchy in `javax.media.jai`



OpImage subclass - constructor

```
public class SubsampleOpImage extends WarpOpImage {  
    // [...]  
  
    SubsampleOpImage(RenderedImage source,  
                     BorderExtender extender,  
                     TileCache cache,  
                     ImageLayout layout,  
                     int scaleX,  
                     int scaleY) {  
        super(source, extender, cache,                      // fwd to superclass  
              setLayout(layout, source, scaleX, scaleY), // set bounds  
              createWarp(scaleX, scaleY),                // Warp object  
              new InterpolationAverage(scaleX, scaleY), // Interpolation obj.  
              true);                                    // cobble sources
```



OpImage subclass - constructor (continued)

```
private static final ImageLayout setLayout(ImageLayout layout,
                                           RenderedImage source,
                                           int scaleX,
                                           int scaleY) {
    if(scaleX <= 0 || scaleY <= 0) {
        throw new IllegalArgumentException("Non-positive scale factor.");
    }

    if(layout != null &&
       (layout.getValidMask() &
        (ImageLayout.MIN_X_MASK | ImageLayout.MIN_Y_MASK |
         ImageLayout.WIDTH_MASK | ImageLayout.HEIGHT_MASK)) != 0) {
        return layout; // bounds set in layout parameter
    }

    ImageLayout il = layout == null ?
        new ImageLayout() : (ImageLayout)layout.clone();

    // Set the bounds to the scaled source bounds.
    Rectangle bounds =
        forwardMapRect(source.getMinX(), source.getMinY(),
                      source.getWidth(), source.getHeight(),
                      scaleX, scaleY);
    il.setMinX(bounds.x);
    il.setMinY(bounds.y);
    il.setWidth(bounds.width);
    il.setHeight(bounds.height);

    return il;
}
```



OpImage subclass - forward mapping method

```
public Rectangle mapSourceRect(Rectangle sourceRect,
                                int sourceIndex) {
    if (sourceIndex != 0) { // this image only has one source
        throw new IllegalArgumentException("Non-zero source index.");
    }

    return forwardMapRect(sourceRect.x + interp.getRightPadding(),
                          sourceRect.y + interp.getBottomPadding(),
                          sourceRect.width - interp.getWidth(),
                          sourceRect.height - interp.getHeight(),
                          scaleX, scaleY);
}

private static final Rectangle forwardMapRect(int x, int y, int w, int h,
                                              int scaleX, int scaleY) {
    float sx = 1.0F/scaleX;
    float sy = 1.0F/scaleY;

    x = Math.round(x*sx);
    y = Math.round(y*sy);

    return new Rectangle(x, y,
                        Math.round((x + w)*sx) - x,
                        Math.round((y + h)*sy) - y);
}
```



JAVA®

OpImage subclass - backward mapping method

```
public Rectangle mapDestRect(Rectangle destRect,
                               int sourceIndex) {
    if (sourceIndex != 0) { // this image only has one source
        throw new IllegalArgumentException("Non-zero source index.");
    }

    return new Rectangle(destRect.x*scaleX - interp.getLeftPadding(),
                        destRect.y*scaleY - interp.getTopPadding(),
                        destRect.width*scaleX + interp.getWidth(),
                        destRect.height*scaleY + interp.getHeight());
}
```



***OpImage* subclass - computation method**



OpImage subclass - computation method (continued)

```
// Get dimensions.  
int dwidth = dst.getWidth();  
int dheight = dst.getHeight();  
int dnumBands = dst.getNumBands();  
  
// Get destination data array references and strides.  
byte dstDataArrays[][] = dst.getByteDataArrays();  
int dstBandOffsets[] = dst.getBandOffsets();  
int dstPixelStride = dst.getPixelStride();  
int dstScanlineStride = dst.getScanlineStride();  
  
// Get source data array references and strides.  
byte srcDataArrays[][] = src.getByteDataArrays();  
int srcBandOffsets[] = src.getBandOffsets();  
int srcPixelStride = src.getPixelStride();  
int srcScanlineStride = src.getScanlineStride();  
  
// Compute scaled source strides.  
int srcPixelStrideScaled = scaleX * srcPixelStride;  
int srcScanlineStrideScaled = scaleY * srcScanlineStride;  
  
// Get the dimensions of the averaging area and cache their product.  
int numCols = interp.getWidth();  
int numRows = interp.getHeight();  
float denom = numCols*numRows;
```



OpImage subclass - computation method (continued)

```
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++)  {
            int imageVerticalOffset = srcPixelOffset;

            // Average the source over the scale-dependent window.
            int sum = 0;
            for (int u = 0; u < numRows; u++)  {
                int imageOffset = imageVerticalOffset;
                for (int v = 0; v < numCols; v++)  {
                    sum += (int)(srcData[imageOffset]&0xff);
                    imageOffset += srcPixelStride;
                }
                imageVerticalOffset += srcScanlineStride;
            }

            dstData[dstPixelOffset] = (byte)(sum / denom + 0.5F);
        }
    }
}
```



OpImage subclass - computation method (continued)

```
        srcPixelOffset += srcPixelStrideScaled;
        dstPixelOffset += dstPixelStride;
    }
    srcScanlineOffset += srcScanlineStrideScaled;
    dstScanlineOffset += dstScanlineStride;
}
}

// If the RasterAccessor set up a temporary write buffer for the
// operator, tell it to copy that data to the destination Raster.
if (dst.isDataCopy()) {
    dst.clampDataArrays();
    dst.copyDataToRaster();
}
}
```



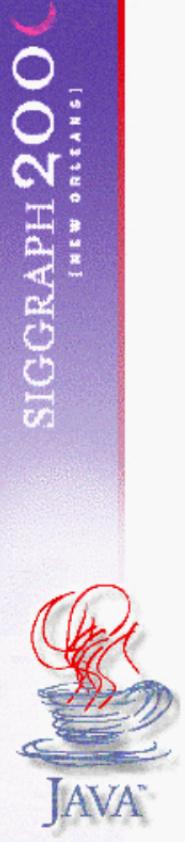
Image factory classes: RIF and CRIF

- Registered with OperationRegistry
- RIF creates an instance of the OpImage subclass



RIF Implementation

```
public class SubsampleCRIF implements ContextualRenderedImageFactory {  
    /** Default constructor. */  
    public SubsampleCRIF() {}  
  
    public RenderedImage create(ParameterBlock paramBlock,  
                                RenderingHints renderHints) {  
        RenderedImage source = paramBlock.getRenderedSource(0);  
  
        BorderExtender extender = renderHints == null ? null :  
            (BorderExtender)renderHints.get(JAI.KEY_BORDER_EXTENDER);  
        TileCache cache = renderHints == null ? null :  
            (TileCache)renderHints.get(JAI.KEY_TILE_CACHE);  
        ImageLayout layout = renderHints == null ? null :  
            (ImageLayout)renderHints.get(JAI.KEY_IMAGE_LAYOUT);  
  
        int scaleX = paramBlock.getIntParameter(0);  
        int scaleY = paramBlock.getIntParameter(1);  
  
        return new SubsampleOpImage(source, extender, cache, layout,  
                                    scaleX, scaleY);  
    }  
    // [...]  
}
```



OperationDescriptor

- Specifies:

- the behavior of the operator
- what layers it supports (rendered/renderable)
- number and types of sources
- parameters

- Provides:

- default initialization of some parameters
- validation of parameters
- property generation

- Registered with *OperationRegistry*



OperationDescriptor Implementation

```
/*
 * An OperationDescriptor describing the "Subsample" operation.
 *
 * The "Subsample" operation subsamples an image by integral factors.
 * The value of each destination pixel is calculated by averaging all source
 * pixel values in a scale-dependent rectangular window around the backward
 * mapped position. The furnished scale factors express the ratio of the
 * source dimensions to the destination dimensions.
 */
public class SubsampleDescriptor extends OperationDescriptorImpl {

    /**
     * The resource strings that provide the general documentation
     * and specify the parameter list for this operation.
     */
    private static final String[][] resources = {
        {"GlobalName", "Subsample"},  

        {"LocalName", "Subsample"},  

        {"Vendor", "org.s2000.courses.jai"},  

        {"Description", "Subsamples an image."},  

        {"DocURL", "http://org.s2000.courses/jai/SubsampleDescriptor.html"},  

        {"Version", "0.0"},  

        {"arg0Desc", "The X scale factor."},  

        {"arg1Desc", "The Y scale factor."}
    };
}
```



OperationDescriptor Implementation (continued)

```
/** The parameter class list for this operation. */
private static final Class[] paramClasses = {
    java.lang.Integer.class, java.lang.Integer.class
};

/** The parameter name list for this operation. */
private static final String[] paramNames = {
    "scaleX", "scaleY"
};

/** The parameter default value list for this operation. */
private static final Object[] paramDefaults = {
    new Integer(2), new Integer(2)
};

/** Constructor. */
public SubsampleDescriptor() {
    super(resources, 1, paramClasses, paramNames, paramDefaults);
}
```



JAVA®

OperationDescriptor Implementation (continued)

```
/** Returns true since renderable operation is supported. */
public boolean isRenderableSupported() {
    return true;
}

/**
 * Returns an array of PropertyGenerators implementing
 * property inheritance for the "Subsample" operation.
 *
 * @return An array of property generators.
 */
public PropertyGenerator[] getPropertyGenerators() {
    PropertyGenerator[] pg = new PropertyGenerator[1];
    pg[0] = new SubsamplePropertyGenerator();
    return pg;
}
```



OperationDescriptor Implementation (continued)

```
protected boolean validateParameters(ParameterBlock args,
                                    StringBuffer msg) {
    if (!super.validateParameters(args, msg)) {
        return false;
    }

    int scaleX = args.getIntParameter(0);
    int scaleY = args.getIntParameter(1);
    if (scaleX <= 0 || scaleY <= 0) {
        msg.append(getName() +
                   " operation requires positive scale factors.");
        return false;
    }

    return true;
}

public Number getParamMinValue(int index) {
    if (index == 0 || index == 1) {
        return new Integer(1);
    } else {
        throw new ArrayIndexOutOfBoundsException();
    }
}
```



Swing image display component

- ➊ Swing component able to contain an image
- ➋ The size of the image and size of the container can be different
- ➌ The image can be positioned within the container
- ➍ Extends JPanel in order to support layout management



Swing image display component implementation



Swing image display component implementation (continued)

```
public synchronized void paintComponent(Graphics g) {  
  
    Graphics2D g2d = (Graphics2D)g;  
  
    // empty component (no image)  
    if ( source == null ) {  
        g2d.setColor( getBackground() );  
        g2d.fillRect(0, 0, getWidth(), getHeight());  
        return;  
    }  
  
    // account for borders  
    Insets insets = getInsets();  
    int tx = insets.left + originX;  
    int ty = insets.top + originY;  
  
    // clear damaged component area  
    Rectangle clipBounds = g2d.getClipBounds();  
    g2d.setColor( getBackground() );  
    g2d.fillRect(clipBounds.x,  
                clipBounds.y,  
                clipBounds.width,  
                clipBounds.height);  
  
    // Translation moves the entire image within the container  
    g2d.drawRenderedImage(source,  
                          AffineTransform.getTranslateInstance(tx, ty));  
}
```



JAVA®

Example Application

- OperationRegistry configuration
- Use of Swing display component
- Use of new operator



Class Definition

```
public class JAIExample extends JFrame
    implements ActionListener, ChangeListener {

    private static Dimension screenSize =
        Toolkit.getDefaultToolkit().getScreenSize();

    private static String DEFAULT_URL_SPEC =
        "http://istserver.eng/images/tiff/Barn_DoorKnob.tiff";

    private static RenderingHints renderHints =
        new RenderingHints(JAI.KEY_BORDER_EXTENDER,
                           BorderExtender.createInstance(
                               BorderExtender.BORDER_COPY));

    // Instance variables.
    JTextField textField = null;
    JSlider slider = null;
    RenderedImage source = null;
    DisplayJAI display = null;
```



OperationRegistry configuration



Class construction

```
public JAIExample(String title) {
    super(title);

    // Add anonymous inner class to listen for windowClosing event.
    // [...]

    // Set layout and add URL text field and scale factor slider.
    Container contentPane = getContentPane();
    contentPane.setLayout(new BorderLayout());
    contentPane.add(textField = new JTextField(DEFAULT_URL_SPEC),
                   BorderLayout.NORTH);
    contentPane.add(slider = new JSlider(1, 10, 1),
                   BorderLayout.CENTER);

    // Load default image if possible.
    try {
        source = JAI.create("url", new URL(textField.getText()));
    } catch(Exception e) {
    }

    // Create image display panel and add it to the frame.
    display = source != null ? new DisplayJAI(source) : new DisplayJAI();
    contentPane.add(new JScrollPane(display), BorderLayout.SOUTH);

    // Listen for events from the slider and the text field.
    slider.addChangeListener(this);
    textField.addActionListener(this);

    pack();
    show();
}
```



ActionListener: actionPerformed()

```
URL url = new URL(textField.getText());  
  
Dimension size = getSize();  
int wOld = source.getWidth();  
int hOld = source.getHeight();  
  
source = JAI.create("url", url);  
  
if(source == null) return;  
  
int wNew = source.getWidth();  
int hNew = source.getHeight();  
  
Dimension d = getSize();  
d.setSize(Math.min(d.width += wNew - wOld,  
                   screenSize.width),  
          Math.min(d.height += hNew - hOld,  
                   screenSize.height));  
setSize(d);  
  
display.setPreferredSize(new Dimension(wNew, hNew));  
display.set(source);  
  
slider.setValue(1); // reset scale factor  
  
pack();  
show();
```



ChangeListener: stateChanged()

```
/** Handle JSlider event: subsample source according to slider valer. */
public void stateChanged(ChangeEvent e) {
    Object o = e.getSource();
    if(o.equals(slider)) {
        int scale = slider.getValue();

        ParameterBlock paramBlock =
            (new ParameterBlock()).addSource(source).add(scale).add(scale);

        RenderedImage image =
            JAI.create("subsample", paramBlock, renderHints);

        display.set(image);
    }
}
```



Conclusion

- **JAI 1.0 released June 1999**
 - Latest update is JAI 1.0.2 released December 1999
- Visit the **JAI home page for the latest information and downloads**
<http://java.sun.com/products/java-media/jai/index.html>
- Send questions to the **jai-interest@java.sun.com** mailing list
 - See home page for details on how to subscribe
- Send comments (private) to **jai-comments@sun.com**



Trademark Notices

- Sun, the Sun logo, Sun Microsystems, Solaris, Java, JDK, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.
- Windows is a trademark of Microsoft, Inc.

